

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/220168963>

Why do users like video?

Article in Computer Supported Cooperative Work · September 1992

DOI: 10.1007/BF00752437 · Source: DBLP

CITATIONS

80

READS

1,532

2 authors, including:



Ellen Isaacs

Google Inc.

65 PUBLICATIONS **2,715** CITATIONS

SEE PROFILE

Sun Microsystems Laboratories

The First Ten Years 1991–2001

*Jeanie Treichel & Mary Holzer, Editors
Susan Banta, Website Design & Production*

Sun Microsystems Laboratories

The First Ten Years ***1991–2001***

Jeanie Treichel & Mary Holzer, Editors
Susan Banta, Website Design & Production

Perspectives 2001–5

In an Essay Series Published by Sun Labs

October 2001

COPYRIGHT INFORMATION

©2001 Sun Microsystems, Inc. All rights reserved. Perspectives, a parallel series to the Sun Microsystems Laboratories Technical Report Series, is published by Sun Microsystems Laboratories, Inc. Printed in U.S.A.

Unlimited copyright without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

"SPARC Version 9: A Robust 64-Bit RISC" originally appeared in SPARC-LINE, June 1992, and is reprinted with permission from SPARC International, Inc.

"An Overview of the Spring System" originally appeared in the Proceedings of Compcon Spring 1994, February 1994. Copyright © 1994 IEEE.

"Solaris MC: A Multi-Computer OS" originally appeared in the proceedings of the 1996 USENIX Conference, San Diego, California, January, 1996.

"Towards Accessible Human-Computer Interaction" is excerpted from Advances in Human-Computer Interaction. Volume 5, Jakob Nielsen, Editor, Copyright 1995 and reprinted with permission from Ablex Publishing Corporation, 355 Chestnut Street, Norwood, New Jersey 07648.

"SpeechActs: A Spoken-Language Framework" originally appeared in IEEE, July 1996 (0018-9162/96). Copyright © 1996 Sun Microsystems and IEEE. Reprinted, with permission, from Computer, Vol. 29, Number 7, July 1996 (0018-9162/96)

"Growing a Language" was an invited talk at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 1998.

"The interactive performance of SLIM: a stateless, thin-client architecture," originally appeared in the 17th ACM Symposium on Operating Systems Principles (SOSP'99). Published as Operating Systems Review, 34(5):32-47, December 1999.

"Linguistic Knowledge Can Improve Information Retrieval" was also published in the Proceedings of the Applied Natural Language Processing Conference (ANLP-2000) in Seattle, Washington, May 1-3, 2000. Also Sun Microsystems Laboratories Technical Report SMLI TR-99-83.

"Designing Fast Asynchronous Circuits," Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA, 11-14 March 2001. pp. 184-193. Copyright ©2001 by IEEE. Used by permission. Also Copyright ©2000, Sun Microsystems, Inc.

"High-Performance, Space-efficient, Automated Object Locking." Copyright ©2001 Sun Microsystems, Inc. and IEEE. All Rights Reserved.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, Java Card, Java HotSpot, Java IDL, Java 2 SDK, J2ME, JavaOne, Java Platform, Java Virtual Machine, JDK, Jini, JRMS, JVM, ShowMe, Solaris Operating Environment, Sun Cluster 3.0, Sun Ray, SunSolutions, and SunTest are trademarks or registered trademarks of Sun Microsystems, Inc., in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc., in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Table of Contents

Notes from the CTO by Greg Papadopoulos	i
Foreword by Director Emeritus William R. (Bert) Sutherland	ii
Introduction and Overview by James G. Mitchell, Director	iii

The following papers were selected to give a broad overview of projects and technologies developed at Sun Labs during its first ten years. The accompanying introductions describe project vision, evolution, and results.

David R. Ditzel	
Introduction by David R. Ditzel	1–1
"Hindsights on SPARC™ Version 9, The Transition from 32 to 64 bits"	
SPARC Version 9: A Robust 64–bit RISC	
 John C. Tang and Ellen A. Isaacs	 2–1
Introduction by John C. Tang	
Why Do Users Like Video?	
Studies of Multimedia–Supported Collaboration (TR–92–5)	
 James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia	 3–1
Introduction by Jim Mitchell	
An Overview of the Spring System	
 Sriram Sankar and Roger Hayes	
Introduction by Alberto Savoia	4–1
Specifying and Testing Software Components Using ADL (TR–94–23)	

Table of Contents (continued)

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall	
Introduction by Jim Waldo	5–1
A Note on Distributed Computing (TR–94–29)	
 Dave Ungar and Randall B. Smith	
Introduction by David Ungar	6–1
Programming as an Experience: The Inspiration for Self	
 James Gosling	
Introduction by James Gosling	7–1
Java™: an Overview	
 James G. Hanko	
Introduction by James G. Hanko	8–1
The Design, Analysis, and Implementation of a Media Server Architecture	
 Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani	
Introduction by Yousef A. Khalidi	9–1
Solaris MC: A Multi–Computer OS (TR–95–48)	
 Paul Martin, Frederick Crabbe, Stuart Adams, Eric Baatz, and Nicole Yankelovich	
Introduction by Paul Martin and Nicole Yankelovich	10–1
SpeechActs: A Spoken–Language Framework	
 Mick Jordan	
Introduction by Mick Jordan	11–1
Early Experiences with Persistent Java™ in First International Workshop on Persistence and Java (TR–96–58)	
 Eric Bergman and Earl Johnson	
Introduction by Earl Johnson	12–1
Towards Accessible Human–Computer Interaction	

Table of Contents (continued)

Phil Rosenzweig, Miriam Kadansky, and Steve Hanna Introduction by Phil Rosenzweig and Miriam Kadansky The Java™Reliable Multicast™Service: A Reliable Multicast Library (TR-98-68)	13-1
Guy L. Steele Jr. Introduction by Guy L. Steele Jr. Growing a Language	14-1
Derek White and Alex Garthwaite Introduction by Steve Heller The GC Interface in the EVM (TR-98-67)	15-1
Israel Cidon, Amit Gupta, Raphael Rom, and Christoph Schuba Introduction by Raphael Rom Hybrid TCP-UDP Transport for Web Traffic (TR-99-71)	16-1
Randall B. Smith, Michael J. Sipusic, and Robert L. Pannoni Introduction by Randall B. Smith Distributed Tutored Video Instruction: Experiments Comparing Face-to-Face with Virtual Collaborative Learning (TR-99-72)	17-1
Antero Taivalsaari, Bill Bush, and Doug Simon Introduction by Antero Taivalsaari and Bill Bush The Spotless System: Implementing a Java™ System for the Palm Connected Organizer (TR-99-73)	18-1
Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt Introduction by J. Duane Northcutt The interactive performance of SLIM: a stateless, thin-client architecture	19-1
William A. Woods, Lawrence A. Bookman, Ann Houston, Robert J. Kuhns, Paul Martin, and Stephen Green Introduction by William A. Woods Linguistic Knowledge Can Improve Information Retrieval (TR-99-83)	20-1

Table of Contents (continued)

Ivan E. Sutherland and Jon K. Lexau	
Introduction by Ivan E. Sutherland	21–1
Designing Fast Asynchronous Circuits	
 Whitfield Diffie and Susan Landau	
Introduction by Whitfield Diffie and Susan Landau	22–1
The Export of Cryptography in the 20 th Century and the 21 st	
 Laurent Daynès and Grzegorz Czajkowski	
Introduction by Laurent Daynès and Grzegorz Czajkowski	23–1
High–Performance, Space–efficient, Automated	
Object Locking	
 Vicky Reich and David S. H. Rosenthal	
Introduction by David S. H. Rosenthal	24–1
LOCKSS: A Permanent Web Publishing and Access System	
 Stephen A. Uhler	
Introduction by Stephen A. Uhler	25–1
The Road to Brazil – aka Design and Architecture of the	
Brazil Web Application Framework (TR–01–99)	

Notes from the CTO

*by Greg Papadopoulos, Chief Technology Officer,
Sun Microsystems, Inc.*

As chief technology officer for Sun Microsystems, I've found that a big part of the job is looking for technologies that have the power to change everything — disruptive technologies that aren't likely to show up on the company's well-planned product roadmap.

At first, the potential of these technologies may go unrecognized or even be denigrated. Do these sound familiar:

"Who would ever download music from the Web?"

"Gee, video on the Net looks lousy — it's not going to be anything."

History would suggest we keep an open mind. Companies with a vested interest in the status quo have often been slow to recognize the significance of new developments, from PCs (considered toys when first introduced) to the Internet.

At Sun, our Lab plays an invaluable role in discovering and developing things no one else saw coming — from platform-independent Java™ technologies to our ground-breaking SunRay™ and Sun™ Cluster products.

It's my pleasure to join Sun Labs in celebrating 10 years of innovation.

– Greg Papadopoulos

Foreword

by William R. "Bert" Sutherland, Director Emeritus, Sun Labs

I am honored by the request to provide comments for the Sun Labs 10 year anniversary report. My time as the Director of Sun Labs from 1992 to 1998 was the highlight of my research management career and certainly an enjoyable, exciting, and thrilling experience.

Sun Microsystems, Inc. has been a terrific supporter of its research laboratory, and I am proud of the manner in which the lab has benefited its patron company. When we started Sun Labs in 1990, Scott McNealy, our CEO, said ***"I know about research labs – they are black holes for money. Stay small at 100 staff until you prove worthwhile."*** Looking back over the past ten years, Sun's "black hole" shone brightly, producing and delivering much of value for the company. By the end of this first decade, Scott indicated his approval by allowing the lab to grow from its constrained beginnings.

The small sampling of papers and reports assembled here can celebrate only a portion of the lab's technical contributions to the company. For example, in some years with only 3–5% of Sun's technologists, Sun Labs produced 30% or more of Sun's new patents. I encourage readers to explore the Lab's technical output on the web at <http://research.sun.com/techrep/index.html> to get a more complete view of its technical accomplishments.

However, technology, while the nominal raison d'être for a technical research lab, is by no means its only contribution to a sponsoring company. Leaving technical retrospective to others, I would like to review some of the non-technical ways in which Sun Labs contributed during its first decade and provide some personal perspective on some subtle values contributed to Sun.

A lab's innovation and creativity derives from its people! Full stop, end of story! Well, not quite end of story. New ideas and technology created in the lab must be put into use by other employees throughout the company, which points out the role of Sun Labs as a "teaching institution" for its company — teaching whatever is new so that the new can become familiar, old, and used widely. Isn't it interesting that patent language refers to "teaching the invention" newly disclosed? Of course, a direct way to move the know-how is to reassign the head that contains it, and staff transfers from Sun Labs into the company have been a successful way of disseminating technology. Sun Labs has placed nearly as many former members as alumnae in the company as its current staff count. Regular lab Open House events have been well attended and permitted many Sun people to see, experience, and perhaps even appreciate firsthand what the lab has accomplished.

An important contribution to the dynamism of Sun Labs has been the summer interns and faculty visitors who have spent time here. We have even had a number of November to February "winter summer interns" from Australia who magically appeared for the southern hemisphere's summer vacation. International visiting professors from Spain, South Africa, Korea, Canada, and Britain, to mention only some, have added outside perspective and technical depth to Sun Labs' activities.

I am particularly proud of those Sun Labs' alumnae who were able as lab members, both to refine their technology abilities and to develop leadership and management skill. Many of these people have moved into the company in technical leadership positions, and they demonstrate the success of the lab as a development ground for technical managers.

"All work and no play makes Jack a dull boy." Two social and recreational aspects of this first decade are worth recounting. The ongoing Friday afternoon "Bash" in Mountain View has been an important social institution that brings together lab staff, other Sun employees, and visitors for discussion, fellowship, and food weekly. Jeanie Treichel and the support staff have for many years made the Bash a pleasant, stimulating, and well attended event. At an early Bash, the incomparable "Water Fight" was conceived and instigated as a SunLabs' challenge to the then SunSoft™ division. Several weeks of preparation helped cement the initial lab staff into a coherent team under "General Rosing", our then lab director — a team dedicated to soaking the opposing "General Zander" of SunSoft and dousing the head referee Scott McNealy in the lake between the opposing buildings. The video is still fun to watch. Until sheer numbers recently interfered, lab alumnae and their families were always invited to the winter parties and summer picnics as a way of keeping personal ties strong and avoiding lab isolation and insularity.

From its very beginning, Sun Labs has had an important outpost in Sun's Boston area engineering community. "Labs East" provides a presence in the Boston educational and technical communities, broadens the accessible pool of talent, and permits additional coupling to Sun's employees and business activities. The staff of Labs East has made many important technical contributions to Sun. As Sun expands into a global company, the recent opening of another Lab outpost in Grenoble, France, further broadens the capabilities, opportunities, and potential impact of Sun Labs.

The past decade also brought Sun Labs two unfortunate staff losses: Charles Molnar and Geoff Wyant passed away in this first decade. For me the loss of such splendid colleagues remains a continuing sadness of my time at Sun Labs.

Finally, let me close with a few personal hopes and expectations for Sun Labs' next decade. The staff is clearly talented and dedicated. Staying close to Sun's customers and employees can be a rich source of interesting problems begging for new technical solutions. New technology starts out as strange and challenging to the old order of activities and habits. Success comes when the new idea spreads from its original single source in one person's mind to become familiar and used by a large multitude of minds. The Java™ programming language started small in only a very few minds and has become an international success because it is understood and used by millions worldwide. All of Sun Labs' successes have seen new understanding develop in many people. The lab staff has a dual job: to create some interesting new technology along with the curriculum needed to see the newness understood and used by many.

Let me end with a quote from Warren Weaver, my high-school commencement speaker:
"The past is prologue!"

Building on its first ten years, I see Sun Labs moving forward into a bright, relevant, and productive future for Sun!

Introduction

by Jim Mitchell, Sun Fellow & VP, Sun Microsystems Laboratories

As this compendium of Sun Labs' first ten years is being assembled, I am the fortunate inheritor of the Labs created by my predecessors: Wayne Rosing, Bert Sutherland, and Greg Papadopoulos. Each one of them has shoes that are very hard to fill, and trying to be as good as all of them is an impossible task. Happily, I don't have to: In Sun Labs we have a collection of some of the most talented, accomplished research scientists and staff on the planet. They have created the ideas and technologies that have built our reputation and that are so well exemplified by this collection of papers.

This compendium, of course, is just a rather small sample of all the innovation that has occurred here in the Labs' first ten years. We hope the sample gives the flavor of Sun Labs, and if you like the sampler, visit our website,

<http://research.sun.com/techrep/>

for additional contributions from Labs researchers over our first ten years.

However, we don't just come up with great ideas. And we don't just build technologies and systems based on those ideas. We put great effort into turning research results into value for Sun. Indeed, this has been one of the hallmarks of Sun Microsystems Laboratories since our birth in 1991.

Many Labs researchers have moved with their creations to Sun product organizations to help turn the projects into products or tools for Sun to sell or use in our business. This culture of technology transfer has worked very well, and the Labs are very proud of our alumni who have gone on to important positions in other parts of Sun where they have become an integral part of Sun's ongoing businesses. In a very real sense, Sun Labs alumni have significantly impacted and influenced the company as much by their presence as by the development of great technologies like the Java™ programming language, the SPARC™ V9 architecture, etc.

This book is a monument to all the people of Sun Labs who have not only made it a vibrant place to conceive and incubate ideas, but who have been a great source of talent for our parent company. They have established the cultural foundation and the standards of quality that, I am sure, will make the next 10 years of accomplishments even better than the first.

As we were preparing this 10th Anniversary volume, the tragic events that led to the destruction of the World Trade Center in New York City claimed one of our own, Phil Rosenzweig. Phil was an enthusiastic, inspiring director in our Burlington, Massachusetts lab, and a warm and caring mentor to his group as well as a dedicated researcher. He will long be remembered by his colleagues and friends. One of his contributions, "The Java Reliable Multicast Service: A Reliable Multicast Library", graces these pages. To him and to all the victims of the terrible events of September 11, 2001, we dedicate this collection.

Hindsights on SPARC™ Version 9, The Transition from 32 to 64 bits.

Introduction by David R. Ditzel, Vice-Chairman and CTO, Transmeta Corporation

Sun Microsystems' future is directly tied to its internally developed SPARC™ microprocessors. First launched in 1987, SPARC was a success in helping Sun grow into the giant company it is today. The success of the SPARC processor family is in large part due to the significant number of software applications available for it. SPARC chips began as a 32-bit processor family, named simply SPARC Version 7 and Version 8, but by the late 1980s it was clear that Sun would eventually need a 64-bit microprocessor. The challenge was how to make this transition from 32 bits to 64 bits without losing customers. Digital Equipment Corporation, with its 64-bit Alpha processors, had chosen a completely new design from its 32-bit VAX processors that was not compatible with earlier VAX software. In hindsight, the lack of compatibility with its installed base of software was probably one of the downfalls of Digital Equipment Corporation and its Alpha processors.

The approach taken by the Version 9 extensions to SPARC was to add 64-bit addressing capability in a way that was completely upwards compatible with all earlier 32-bit SPARC software. That meant that the several thousand existing 32-bit SPARC applications could continue to run on newer 64-bit SPARC hardware. In hindsight, the achievement of compatibility with its installed base of software was probably one of the key technologies that contributed to Sun's success and growth.

Along with the changes for 64-bit addressing, the SPARC community used the Version 9 specification as an opportunity to make a number of other changes to the SPARC architecture. The paper included in this document gives a technical summary of some of the major changes to the SPARC architecture between Versions 8 and 9. This 1992 paper was bold in making claims for the next decade about what it would take to survive into the next century. Looking back from the year 2001 today, most of those claims have come true, and SPARC is one of the few remaining RISC architectures still strongly supported by its parent company.

First introduced in the UltraSPARC™-1 microprocessor, SPARC Version 9 was a major turning point for SPARC, and Version 9 chips now power all of Sun's SPARC computers. SPARC is one of the few enduring technologies that have touched nearly all of Sun's computing products. Looking forward, the changes introduced in SPARC Version 9 are still well positioned to carry SPARC forward for another decade.

SPARC™ Version 9: A Robust 64–Bit RISC

Dave Ditzel, Director of Advanced Systems, Sun Microsystems Laboratories, Inc.

The Version 9 SPARC™ Architecture is something you will hear a lot about, it has just been announced by SPARC International. Version 9 (V9) constitutes the first major architectural change made to SPARC since it was first announced five years ago. The following review should provide context to help you understand what V9 is all about.

First, V9 is an architectural specification from SPARC International that extends SPARC. Sometime the specification is confused with a new SPARC chip. V9 is an update to the V8 Architecture. V9 is not a SPARC chip, and it is not from Sun, it is an architectural specification.

When it is published, it will be the open specification of what defines a SPARC, in a book form like the V8 manual, and available in any bookstore worldwide. V9 is not in published form yet, but it is done, and the specification will not change at this point. One of the interesting things is that V9 is really the first major architectural change to SPARC since it was announced nearly five years ago. As a result, many different companies will begin implementing microprocessors and systems around this standard in the future.

Why SPARC™?

One of the reasons SPARC has been successful is that we've had real stability with the architecture. SPARC International member companies have implemented over twelve different SPARC compatible microprocessors since SPARC was first announced, and that's more than any other microprocessor family in history has done with this level of binary compatibility. As a result, SPARC has ended up with more than 5000 compatible applications, and is why SPARC has accumulated more volume. V9 maintains this same stability with upwards binary compatibility, a very important feature. However, we've learned a lot more about RISC in the past few years, and found there are some additional changes we would like, and believe that now is the time for the next

major change for SPARC.

Processor Needs for the 90s

In the early 1980s the idea of RISC began. Developers used what we had learned about the mix between compiler technology and hardware with 1980s level of knowledge, and provided a very minimal system interface for UNIX®. It was just enough to get UNIX® up and running.

The computer industry will need more in the 1990's. The industry has matured and people want growth with compatibility. There is a need we can clearly see in the future—over the next several years—for a larger address space. People want better interfaces to the operating system, as well as support for multiprocessors, lightweight threads, and object oriented programming. More reliability is needed for systems today, than in the past. We've learned more about computer design since the 1980s, and have been working hard to incorporate these new ideas in the design of V9. There are new pipelines that do more than one instruction per cycle and there needs to be support for that. And overall, users want their systems to have good performance growth.

Now if you look at what some of the competition has done—MIPS and Alpha for example, what they've really done is developed a 1980s style of RISC. They've left out a lot of things we think customers are going to need, and that is what SPARC V9 is addressing.

SPARC V9: Making RISC Robust for the Next Century

SPARC V9 is a robust RISC that will last into the next century. One of the main features of V9 is extending SPARC to 64-bit addressing. But V9 also does a lot of things which other architectures simply have not addressed, and are critical. The work on V9 includes performance enhancements, as well as developments to support advanced types of compiler optimization. As a result, we will be able to support today's compilers and the ones we see coming in the next several years.

Changes in V9 will support advanced superscalar pipeline implementations. We've completely redesigned the interface of the system architecture for the style of operating systems we expect in the mid to late 1990s. Fault tolerant features have been added to the basic system architecture, a rather unique feature not found in other microprocessor chips. Support has been added to make extremely fast traps and context switching in the architecture, because there is a belief that it will go with the style of programming we'll be using in the next several years.

Extending Addressing to 64 Bits

V9 supports 64-bit virtual addressing. We've used 64-bit data paths for a long time in SPARC implementations, now the architecture can scale up its addressing range as well. The benefit of this increased addressability is ensuring the long lifetime of the SPARC architecture. We don't want to end up with a 32-bit architecture that can't migrate forward, and that's a lot of what people want to know today. If SPARC can move forward with a bigger address space, then we know our software investment will last for many more years. If not, we might have to think about changing. The good news is that SPARC will indeed extend gracefully.

The way we implement extended addressing in SPARC is by extending all the integer registers to 64 bits. We've added a new condition-code register that is set by 64-bit computations. We've also added several new instructions that explicitly manipulate 64-bit values in the architecture, such as shifts, 64-bit load/store, things of this type.

Now we can still execute today's software programs on a new V9 microprocessor. The way this works is we continue to use the existing instructions which operate on registers. They might, for example, add register one to register two and put the results on register three. The basic instruction set doesn't really know whether the register is 32 or 64-bits long. So if you were to run one of today's programs on a V9 machine, it would get the same results as it does today. However, if you want to take advantage of the extended addressing and future capabilities of

SPARC V9, you can recompile the program and the compiler will know how to take advantage of these additional 64-bit features which are in the architecture and extend the address space and will give you all the additional functionality.

Performance Enhancements

One of the biggest concerns is performance, and we've changed some basic things in the hardware architecture of SPARC that will help performance. There are sixteen more double-precision floating point registers in SPARC, which brings us up to a total of 32. The benefit of registers is to reduce memory traffic; so you don't have to do as many loads and stores, and your program will run faster. There are also eight more quad-precision registers.

We supported a 128-bit floating point format which is also unique for microprocessors. We've added four floating-point condition registers; we've only had one in the past. This would then let you have more parallelism in the basic architecture for a superscalar machine, launching multiple instructions at a time. If you had only one condition code, you could have a serial dependence waiting for that one condition code to finish. This will allow us to launch multiple floating-point instructions simultaneously without slowing down the chip.

We've changed a number of other things in the instruction set area of SPARC purely for performance reasons. We've done things like add 64-bit integer multiply and divide instructions. We've added load and store quadword instructions to load and store 128 bits at a time. We've added branch prediction, because branches are something that are very difficult to implement in computers. By having branch prediction, we can eliminate a lot of the delays associated with branches. We have a new instruction called branch on register value. This reduces the total number of instructions you have to execute, thereby speeding up the program. This also gives the effect of having many integer condition codes, so we get the same advantages of parallelism on the integer side that we got from multiple condition codes on the floating point side.

Support for Advanced Compiled Optimizations

We have added support for advanced compiler optimizations. We are seeing a lot of interesting new optimizing compiler techniques that we'll be able to take advantage of in the 1990s to give us greater levels of performance. Just a couple of examples of things we have changed include instructions that do pre-fetching of data and instructions. The benefit here is to reduce the memory latency so the program isn't waiting for the memory to respond. If you get that response out long enough in advance then the answer is back from memory when you actually need to make use of it. There is support for misaligned data. One example of this benefit is in FORTRAN compilers. Because of the way the language is specified, in many cases compilers cannot analyze whether or not it is legal to use a double precision load, and many architectures do two load singles. Support for misaligned data will allow the compiler to always use the most optimal instruction. In those rare occasions where things aren't aligned properly, the underlying V9 architecture can fix things up. This will translate directly into increased performance. There is support for something called speculative loads which enables the compiler to do better code scheduling of instructions. This is something no other architecture has. We've added support for conditional move instructions. Conditional moves allow you to eliminate branch instructions, and any time you can make branch instructions go away or predict them better, the performance of the machine will go up. And we've added something I thought you might like to hear about called the TICK register. It's just a little timer that counts away once for every lock tick on the machine. It will let programmers make very accurate time and performance measurements.

Support for Advanced Superscalar Processors

V9 includes support for advanced superscalar processor designs. What we're finding is a trend where we are learning to execute more instructions per cycle every year with new pipelines. Two instructions at a time to three instructions at a time. We want to be able to get up to doing eight, nine, ten instructions at a time

with the SPARC architecture. In order to do this, we wanted to make some changes that would make superscalar execution work better. We already have superscalar SPARC chips with SuperSPARC and HyperSPARC. SPARC already has features which are good for superscalar by having simple to decode, fixed-length instructions, and having separate integer and floating point units. V9 supports superscalar processors in several ways. Having more floating point registers available allows computing more things in parallel. Support for speculative loads is very important. Having multiple condition codes allows more things to go on in parallel. Using branch prediction cuts down on branch penalties.

Support for 90's Operating Systems

The operating system interface in V9 has been completely redesigned. There are new privileged registers in the machine; there is a new structure to those registers which makes it much faster to get at important control bits in the machine. One of the interesting things to remember here is that the change in the operating system interface has no effect on user level applications. The user-level programs do not see these particular changes that have been made.

By making these changes we get support for the new microkernel style approach to operating system design, support for lightweight threads and are able to run them much more efficiently, much faster context switching by re-architecting this level, as well as support for object-oriented software. One of the things we have done is to take register windows and make them more flexible than they already were. This lets you do context switching, for example, between different processes and use the register windows as banks of registers to achieve no-overhead context switching.

We've added support for very large-scale multiprocessors. SPARC already supports multiprocessors, but for much larger systems in the future we found new ways to improve performance. One way is by relaxing the constraints on the memory system, using the new Relaxed Memory Ordering model. Individual processors can then synchronize efficiently using

a new Memory Barrier instruction.

Support for Reliability and Fault Tolerance

Something else that microprocessor design has pretty much ignored in the past is explicit support for reliability and fault-tolerance. Now you can build a reliable and fault-tolerant machine without some of the support, but it's a lot more work. So we've done a number of things.

First of all, we've added one very specific instruction called compare and swap. The benefit here is that this particular instruction has some very well-known fault-tolerant features and is also a very efficient way to do multiprocessor synchronization.

We've added multiple levels of stacked traps in V9. Stacked traps allow you to gracefully recover from various kinds of faults as well as allowing you to design your operating system to be much more efficient. If you look at SPARC V8 and other RISCs, they typically have only one trap level. This is okay, you can make things run, but in fact, to do a very nice fault-tolerant machine, having these multiple trap levels will be helpful.

Let me show you an example of what these levels do. There are five different levels. A program would normally execute at Level 0. You can trap into the operating system at level 1, and the operating system can go ahead and not worry too much about causing other traps to happen. If you take another kind of fault, you can trap again to level 2; for example, you can take yet another trap to get to the page fault handler. And, finally, we've added a special new mode in SPARC called RED mode, for Reset, Error and Debug mode. It fully defines the things you need to build a very fault-tolerant system.

Fast Traps and Context Switching

A last technical point is fast traps and context switching. Fast traps turn out to be something that system designers use a lot to make a number of things on the machine run faster. The way we've done this is to re-architect the trap entry code to get you into those instructions in the trap handler very quickly. We've added eight new

registers called "alternate globals" so that when you enter a trap routine, you have a fresh register set to work with, you don't have to worry about storing something away in order to do some computation. The benefit here is that it will give you very fast instruction emulation, and very quick system response time. When you do something like type a character on a keyboard or move the mouse around quite often, this will cause an interrupt in the machine. And so getting into the trap handler and getting back fast will directly effect the user's physical response to how his machine is operating.

We've added multiple levels of stacked traps. It's important that a trap routine itself will allow another trap to happen while you're in it. Otherwise, you'll have to do a tremendous number of checks to make sure that you don't get another problem. For example, in the register window trap handler, you currently need to check while you're in the trap handler that nothing else is going to go wrong, such as you don't take a page fault, or that something else doesn't happen. By adding these multiple levels of traps, we can reduce the number of instructions from one hundred down to about nineteen. That's a big increase in performance. The real benefit here is that it allows you to design your operating system with much better performance and a cleaner programming model.

We've also helped context switching. What we've done here is allow the machine to save or restore fewer registers during a context switch. The way we have done this is in both integer and floating point registers, we can tell more efficiently what registers are actually being used. The floating point registers now have a floating point enable bit that is split into two pieces with a "dirty bit" for the upper and lower part of the registers. So if you haven't written into the registers recently, when you do to do a context switch, there's no need to save them away. So the fastest way to save registers is to not save them at all.

We can also use integer registers in the same way by using register windows as banks of registers. So you could find that in a future

SPARC processor, you're doing a context switch without the need to save any registers away, either on the integer or floating point sides. That makes for very fast context switching.

Summary

So let me summarize some of the features and what we've done overall in V9. We have provided 64-bit addressing; support for fault tolerance; fast context switching; support for advanced compiler optimizations; very efficient design for superscalar processors; and a very clean structure for modern operating systems. And we've done it all with 100% upwards binary compatibility. That is a significant achievement. What I see in the future is good implementations of SPARC V9 chips that will give us superior performance, very robust systems, and very good cost efficiency. That is what our customers are asking for. SPARC has been the RISC leader for the last five years, and with these changes that we are making in SPARC V9, we expect SPARC to remain the RISC leader into the next century.

Why Do Users Like Video?

Studies of Multimedia–Supported Collaboration

John C. Tang and Ellen A. Isaacs

Introduction by John C. Tang

This technical report, written in 1992, describes our research group's experience with a desktop video conferencing prototype. The prototype enabled users to establish audio–video connections among SPARC™ workstations. It also included a whiteboard sharing capability that allowed users to draw and point together over a shared image on their computer screens. We studied how a group distributed between Sun's East and West Coast sites used the prototype and documented our findings on the value of video and how desktop conferencing fits into the larger spectrum of communication media.

This work was conducted when research groups were exploring new applications for video and audio, especially to support distributed groups. Researchers began examining the value of video, compared to just audio (especially given the added expense in technology and network bandwidth). Our project was part of a body of work in the research community that investigated the value of video and explored for what applications it was useful. Building on our initial project, we went on to develop other video–based prototypes (see Montage and Piazza references) and studied how people used these prototypes.

This research built on a hardware prototype board for a SPARC workstation for capturing and displaying video, prototype software for establishing desktop video conferences and shared whiteboard sessions among users, and a study of people actually using the system in their day–to–day activities. Thus, this research involved new hardware, new software, and a new way of studying how people use technology, all done within two years of the group's formation as one of the founding groups in Sun Labs.

The shared whiteboard prototype led directly to the Sun ShowMe™ Whiteboard product that was developed by SunSolutions™. The ShowMe product line eventually broadened to include video and audio, and was the precursor to the current line of SunForum™ desktop conferencing products. Our experience with the prototype video card helped shape the development of the SunVideo™ line of cards that integrated digital video into Sun workstations. While the widespread adoption of video and audio applications has been hampered by the lack of a standard multimedia platform for UNIX®, the recent advent of streaming video platforms provide the opportunity to fully realize and productize the concepts demonstrated in our prototypes.

Our current Network Communities research group in Sun Labs continues in this tradition of research on collaboration, exploring new applications for supporting distributed group work and studying how they are used.

REFERENCES:

Tang, John C. and Ellen A. Isaacs, "Why Do Users Like Video? Studies of Multimedia-Supported Collaboration", *Computer Supported Cooperative Work: An International Journal*, Vol. 1, Issue 3, 1993, pp. 163–196.

Isaacs, Ellen, A. and John C. Tang, "What video can and cannot do for collaboration: a case study", *Multimedia Systems*, Vol. 2, No. 2, August 1994, pp. 63–73.

Tang, John C. and Monica Rua, "Montage: Providing Teleproximity for Distributed Groups", *Proceedings of the Conference on Computer Human Interaction (CHI) '94*, Boston, MA, April 1994, pp. 37–43.

Tang, John C., and Monica Rua, "Montage: Multimedia Glances for Distributed Groups", *SIGGRAPH Video Review*, Issue 106, October 1994 (videotape).

Tang, John C., Ellen A. Isaacs, and Monica Rua, "Supporting Distributed Groups with a Montage of Lightweight Interactions", *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW) '94*, Chapel Hill, NC, October 1994, pp. 23–34.

Why Do Users Like Video?

Studies of Multimedia-Supported Collaboration

John C. Tang and Ellen Isaacs

1. The promise and perplexity of multimedia-supported collaboration

Multimedia technology promises to enable smooth and effective interactions among collaborators in different locations. The growing need to support technical and social activity that occurs across geographical distances has not been fully satisfied by the current technologies of phones, faxes, electronic mail, and video conference rooms. Visions of systems that allow people from around the world to see and hear each other have been promoted at least since AT&T unveiled the PicturePhone in the mid-1960's.

Recent technology and infrastructure developments are adding to the promise of multimedia support for remote collaboration [Gale 1992]. The emergence of digital audio and video technology allows voice and images to be computationally manipulated and transmitted over the existing computer networks. Improved compression algorithms running on faster hardware promise to provide acceptable audio-video quality at viable network bandwidth rates. The availability of affordable computer workstations, proliferation of digital networks, emergence of compression algorithm and network protocol standards, and marketing hype are converging to bring multimedia capability to personal desktops.

Research prototypes that provide what is often referred to as desktop conferencing (audio, video, and computational connections between computer desktops) have been demonstrated using analog [Root, 1988; Stults et al., 1989; Buxton & Moran, 1990] and digital [Watabe et al., 1990; Masaki et al., 1991] technology. Olson and Bly [1991] reported on the experiences of a distributed research group using a network of audio, video, and computer connections to explore ways of overcoming their separation in location and time. These prototypes have demonstrated the technical feasibility of desktop conferencing, experimented with some of its features, and provided a glimpse of how people will use it.

However, increased costs (e.g., upgrading networks, buying media-equipped workstations) and uncertainty over the benefits of collaborative multimedia have been significant barriers to its widespread adoption and use. While videophone products have recently reappeared in the marketplace, the lack of commercial success of PicturePhone since it was introduced almost 30 years ago indicates that there is much yet to be learned about the deployment and use of collaborative multimedia technologies [Francik et al., 1991].

Furthermore, research to date on the effects of various communication media on collaborative activity has not provided convincing evidence of the intuitively presumed value of video [Williams, 1977]. Ochsman and Chapanis [1974] examined problem-

solving tasks in various communication modes including typewriting, voice only, voice and video, and unrestricted (working side-by-side) communication. They concluded that relative to communication modes using an audio channel, "...there is no evidence in this study that the addition of a video channel has any significant effects on communication times or on communication behavior." [Ochsman & Chapanis, 1974, p. 618].

Gale [1990] compared computer-mediated collaboration on experimental tasks under three conditions: sharing data only (via a shared electronic whiteboard), sharing data and audio, and sharing data, audio, and video. He also concluded, "The results showed no significant differences in the quality of the output, or the time taken to complete the tasks, under three conditions: data sharing; data sharing plus audio; data sharing plus audio and video." [Gale, 1990, p. 175]. Gale did find that collaborators' perceptions of productivity increased as communication bandwidth increased and suggested that higher bandwidth media enabled the groups to perform more social activities.

Some research has begun to identify uses of video in support of remote collaboration. Smith et al. [1989] compared computer-mediated problem-solving activity in audio only, audio and video, and face-to-face settings. They found that the presence of the video channel encouraged more discussion about the *task* rather than the *mechanics of the computer tool* being used for the task. Fish et al. [1992] equipped mentor-student pairs of researchers with a desktop conferencing prototype and studied their informal communication over several weeks. They found that the prototype was used frequently, but the users thought of it more like a telephone or electronic mail rather than face-to-face communication.

Most of the studies to date have used artificial groups (subjects randomly assigned to work together) working on short, contrived tasks (problems unrelated to their actual work). We hypothesized that evidence for the value of video would be most visible in *actual* work activity of *real* working groups. We set out to study real examples of synchronous, distributed, small group collaboration in order to understand how multimedia technology (video in particular) could be designed to support that activity. The research pursued an iterative cycle of studying existing work activity, developing prototype systems to support that activity, and studying how people use those prototypes in their work [Tang, 1991b].

In this paper, we first describe two background studies that examine existing remote collaboration work practice—a survey of users' perceptions of an existing video conference room system and a study of a geographically divided work group in various collaboration settings. Then we describe the development of a prototype desktop conferencing system, which embodied some of the design implications identified by the background studies. We used that prototype to study a distributed team under three conditions: using their existing collaboration tools, adding the desktop conferencing prototype, and subtracting the video capability from the prototype. We conclude by discussing evidence from the three studies which help explain why users like video and other related issues.

2. Survey of video conference room users

The first background study surveyed users' perceptions about an existing video confer-

ence room system. The survey was conducted within Sun Microsystems, Inc. which at the time used commercially available video conference room systems (PictureTel Corporation model CT3100 operating at 112kb/s bandwidth). These systems connected conference rooms among sites in Mountain View, California; Billerica, Massachusetts; Colorado Springs, Colorado; and Research Triangle Park, North Carolina. In addition to the audio-video link, they could also send high quality video still images on a separate video display.

A survey was sent via electronic mail to users of the video conference room system. The survey asked for usage information and for the users' perceptions of the system. A total of 76 users responded to the survey, with representatives from all four sites and a variety of types of meetings (e.g., staff meetings, presentations, design meetings).

2.1 Survey Results

Users were asked to indicate the best aspects of video conferencing. Respondents could check more than one item from a list of choices as well as add their own items. Most respondents (89%) liked having regular visual contact with remote collaborators. Many also indicated that it saved travel (70%) and time (51%). This survey measured only the users' *perceptions* of saved time and travel, not whether those savings actually occurred.

Users were also asked to indicate the worst aspects of their video conferencing experience. The percentage of respondents who checked each aspect is shown in Figure 1. The most frequently indicated problem (72%) was difficulty in scheduling an available room. Poor audio quality (poor microphone pickup, moving the microphones into range of the speaker, echo, etc.) was indicated by 55% of the respondents, 53% mentioned not being able to see overheads and other materials used in presentations, and 52% complained about the time delay (latency) in transmitting audio and video through video conferences. Between Mountain View, California and Billerica, Massachusetts, the system exhibited about a 0.57 second delay between capturing voice and video on one end and producing them on the other end. Poor video quality was relatively less troubling; only 28% mentioned it as a problem.

Some sample comments from the survey that illustrate these observations:

... we need more video rooms! They are overbooked.

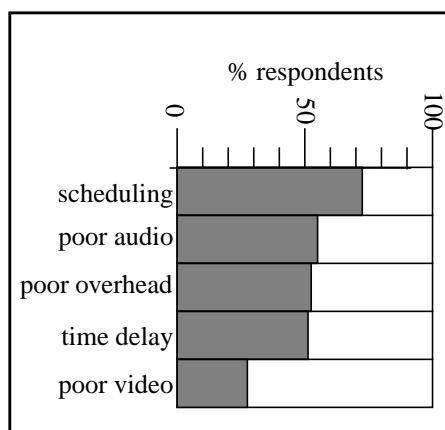


Figure 1. Worst aspects of video conferencing

Percentage of respondents who indicated the following as the worst aspects of their video conferencing experience: difficulty in scheduling the room, poor audio quality, poor ability to see and interact with overhead slides, time delay (latency) in audio-video transmission, and poor video quality.

... audio quality is the real problem. Both audio quality and delay.

Difficulty in being able to make a comment--can't see a verbal opening coming because of the delays and image and audio quality

Can't provide direct feedback on written material (e.g., by pointing and/or annotating slides and drawings presented remotely)

Respondents were also asked to rank order a list of additional capabilities (to which they could add their own) they would like to have in video conferencing. Figure 2 shows the five most frequently requested features with each item's average rank (rank 1 is most urgently desired) labelled on the bar chart. The need for a shared drawing surface stood out as the most commonly desired feature; 68% of the respondents mentioned it as a desired feature, and its average rank order was 1.76. Respondents also indicated that they wanted a larger video screen (34%) and the ability to connect multiple sites together at the same time (30%). Only 18% wanted to incorporate computer applications, but its low average rank (1.75) suggests that those who wanted it considered it a highly desirable feature. Users suggested incorporating software such as word processing, spreadsheet, and shared whiteboard applications. More comments from the survey:

A shared drawing surface could be really useful. It should be a single device, so that you draw on the device and see your marks and the other person's marks on that same device.

...networked [computer workstation] in each conference room would be nice, especially one that could project onscreen at the local site and at the remote sites...

[larger video screen] so I can really tell who's talking and get a fix on facial/body talk better...

A similar survey of users of video conferencing systems [Masaki et al., 1991] identified some of the same features as requirements for improving video conferencing. They found that users wanted a virtual common space (including a shared drawing space),

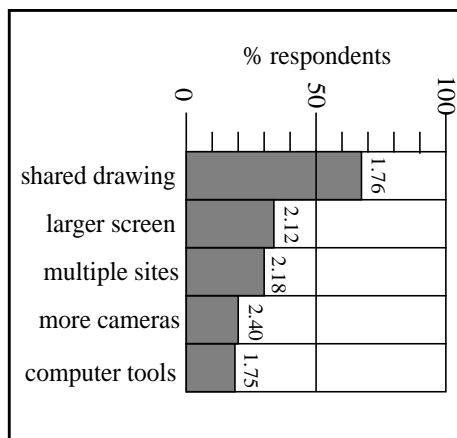


Figure 2. Desired features for video conferencing

Percentage of respondents who indicated the following as the features they would like to add to video conferencing: a shared drawing surface, larger video screen, connections among multiple sites at a time, more camera views, and access to computer tools while in a conference. The average rank of how urgently each feature was desired is also labelled (rank 1 being most urgently desired).

integration of teleconference and computational tools, and multiple site conferencing capability.

2.2 Design Implications From Surveying Video Conferencing Users

The survey did show that video conferencing users broadly appreciated the capabilities of video conferencing. Users' comments indicated that collaboration between remote sites would not be as effective or even possible without video conferencing. The survey responses indicated that multimedia tools to support collaboration should:

- be readily available for use,
- provide a shared workspace, and
- provide high quality, interactive audio among sites.

Note that the aspect of the system that users found most troublesome (scheduling difficulty) is a problem of *use*, not a *technical* problem per se. The technical benefits (and problems) of multimedia-supported collaboration tools will not be discovered if users cannot readily access them. Respondents' desire for a shared workspace reinforces research results identifying the crucial role of shared workspaces in remote collaboration [Olson & Bly, 1991; Tang, 1991a]. The responses also clearly indicated the need to improve the audio channel, both in sound quality and transmission delay. By contrast, users were not as disturbed about the image quality of the video channel. This pattern is consistent with results reported in the literature on the greater importance of audio relative to video in supporting remote collaboration [Gale, 1990; Ochsman & Chapanis, 1974].

3. Study of collaboration in various settings

After completing the video conferencing survey, we studied a work group composed of four members from two different sites: three in Billerica, near Boston, and one (the second author of this paper) in Mountain View, in the San Francisco Bay Area. Their discussions centered around graphical user interfaces for on-line help systems. The group conducted weekly video conference room meetings, supplemented by occasional phone conferences. At one point in the project, the Mountain View participant visited Billerica for a week of face-to-face meetings. Although some participants knew each other from previous work contacts, this was the first time they worked together extensively as a project team.

Over two months, meetings in each of the collaboration settings were videotaped and analyzed to identify characteristics of their collaboration that varied among the settings. The collected data comprised eight video conferences, five face-to-face meetings, and one phone conference, amounting to over 15 hours of data.

3.1 Findings From The Study Of Collaboration Settings

From reviewing the videotapes, we found that the team experienced certain problems while using the video conference rooms that did not arise in face-to-face meetings:

- problematic audio collisions,
- difficulty in directing the attention of remote participants, and
- diminished interaction.

During their video conferences, there were many instances of audio collisions when participants on both sides started talking simultaneously and then had difficulty negotiating who should take the next turn. Although such collisions naturally occur in face-to-face and phone conversations, they were more problematic in video conferencing. In face-to-face conversation, turn transitions are largely negotiated verbally (aided by gestures) through precise timing (sometimes involving overlapping talk) and systematic, implicit organization [Sacks et al., 1974]. The more than half-second delay in transmitting audio between video conference rooms disrupted these mechanisms for mediating turn-taking. As a result, participants sometimes relied on gestural cues (e.g., extending a hand toward the camera conveying “you go first”), which were usually successful if seen.

The participants also had occasional difficulty directing a remote collaborator’s attention to the video display so that these gestures would be seen. We observed several examples of “just missed” glances between remote collaborators when one participant would look up from her notes to glance at her remote collaborator, but returned looking down at her notes just before the remote collaborator looked up at his video display to glance at her. These missed glances could largely be explained by delayed reactions caused by the transmission delays. However, just missed glances have also been observed in audio-video links that do not have any perceivable delay [Smith et al., 1989; Heath & Luff, 1991]. Current research suggests that lack of peripheral vision, division of attention between video windows and the shared workspace, and other aspects of video links also play a role in disrupting the coordination of glancing at each other.

Difficulties in negotiating turn-taking and directing participants’ attention in video conferencing apparently combined to reduce the amount of interaction between the remote parties compared to face-to-face meetings. Video conferences tended to consist of a sequence of individual monologues rather than interactive conversations. We observed less frequent changes of speaker turns, longer turns, and less back-channelling in video conferencing than in face-to-face meetings. This reduced level of interaction appeared to affect the content of video conferences by suppressing complex, subtle, or difficult-to-manage interactions. Participants seemed inhibited from expressing their opinions and, in particular, avoided working through conflict and disagreement. Video conferences also exhibited a marked lack of humor (in part because humor relies on precise timing).

3.2 Design Implications From Studying Collaboration Settings

Comparing collaboration in video conferencing with face-to-face and phone conferencing settings underscored the need to provide responsive (minimally delayed) audio in technology to support interaction. The work group we studied was so frustrated by the audio delays in video conferencing that they turned off the audio provided by the video conferencing system and placed a phone call (using speakerphones) for their audio channel. Although this arrangement eliminated the audio delay, the audio now arrived *before*

the accompanying video (i.e., audio and video were no longer synchronized), the audio quality was poorer, and speakerphone audio was only half-duplex (only one party's sound was transmitted at a time). Nonetheless, the collaborators strongly preferred this arrangement to the frustrations they experienced with the delayed audio. Their meetings conducted under this arrangement appeared to exhibit more frequent changes in speaker turns, more back-channelling, and more humor than those using the normal video conference configuration. More research is needed to investigate these informal observations.

This experience indicates that users prefer audio with minimal delay even at the expense of disrupting synchrony with the video. This observation again confirms research findings of the greater importance of audio relative to video [Gale 1990; Ochsman & Chapanis, 1974], and is also consistent with users' perceptions from the survey that audio quality and responsiveness are more important than video quality. This finding suggests that, given the limited bandwidth and performance available for desktop conferencing, more attention should be devoted to providing responsive, interactive audio. More research on the trade-offs and limits of degrading other parameters of desktop conferencing (e.g., video quality, video refresh rate, audio quality, audio silence suppression) is needed.

While our studies confirm the greater importance of audio relative to video, they also provide evidence for the value of video in supporting remote collaboration. Through the video channel, gestures were used to demonstrate actions (e.g., enact how a user would interact with an interface) and the participants' attitudes. Especially under the delayed audio conditions of video conferencing, video was valuable in helping mediate interaction (e.g., using gestures to take a turn of talk) [Krauss et al., 1977].

4. Developing a desktop conferencing prototype

The observations gained from these two studies helped guide the design of our research prototypes for new multimedia technology to support collaboration. An initial phase of this research was to design and implement a prototype desktop conferencing system that provided real-time audio and video links and a shared drawing program among participants at up to three sites. The desktop conferencing prototype was built on a prototype hardware card that enabled real-time video capture, compression, and display on a workstation desktop. This prototype card, in conjunction with the workstation's built-in audio capability, enabled digital audio-video links among workstations on a computer network.

Figure 3 shows the user interface for establishing and managing desktop conferences. Initiating a conference was modeled after placing a telephone call. A user selected from a list of receivers to request a conference with them. An identical copy of the interface appeared on the receivers' screens, announced by three beeps. Each receiver could decide to join or decline the conference. A shared message area allowed users to send text messages among each other to negotiate joining or refusing a conference.

Once all receivers joined the conference, the collaborative tools that the group

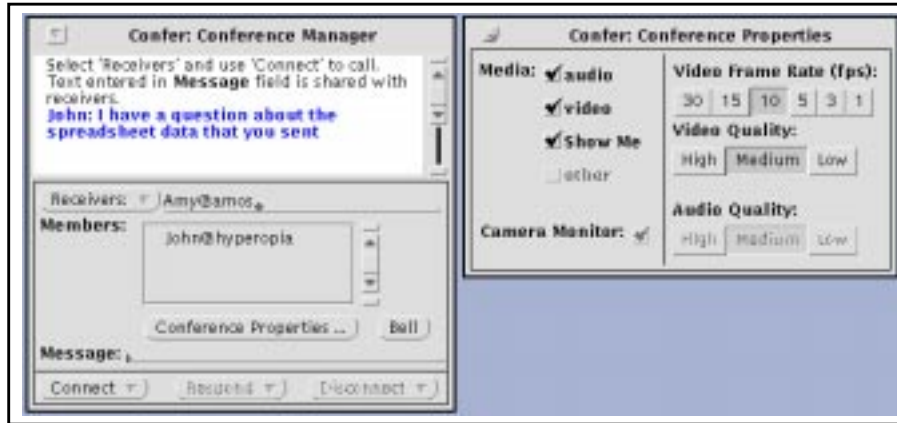


Figure 3. User interface for managing conference connections

John places a call to Amy to request a conference using the default conference properties of audio, 10 frames per second video, and the Show Me shared drawing tool.



Figure 4. Screenshot of a desktop conference

A typical 2-person desktop conference consists of the Show Me shared drawing tool (showing an image of a spreadsheet), a video window of the remote user (Amy), and preview video window of the outgoing video signal.

requested were invoked. Figure 4 shows a screen image of the tools that comprised the desktop conferencing prototype. For a two-way conference, each user's screen displayed:

- a video window of the remote collaborator,
- a preview window of the video signal being sent to the remote collaborator, and
- a shared markup and drawing program (called Show Me) for drawing, typing,

pointing, and erasing over shared bitmap images.

Show Me allowed users to create shared free-hand graphics and allowed any user to grab a bitmap image from their screen and share that image with the others.

The two studies of existing collaboration activity helped shape the design of the desktop conferencing prototype. The survey identified users' need for a shared drawing space, prompting a significant investment in the development of the Show Me shared drawing tool. The design of Show Me drew upon shared drawing research [Tang & Minneman, 1991; Minneman & Bly, 1991] to provide a drawing surface that remote collaborators could share in much the same way that face-to-face collaborators use a whiteboard.

The study of collaboration settings identified the problem of audio delay and underscored the importance of the audio channel in mediating interaction. The delay in the audio of the desktop conferencing prototype was minimized to be in the range of 0.22-0.44 seconds (depending on computational and networking constraints). Since the audio and video data streams were being sent through a computer network that was shared with other users, spurts of heavy network traffic affected the prototype's performance. When network loading prevented video frames from being delivered at the requested video rate, the video image would occasionally freeze until an updated frame was received. More severe loading caused cut outs in the audio signal. One characteristic of handling the audio and video data streams separately was that the timely delivery of video would degrade before the delivery of audio was disrupted. This behavior reflected the study's finding that immediate and responsive audio was more important than preserving audio-video synchrony.

5. A Study of the use of desktop conferencing

We used the desktop conferencing prototype (DCP) to study the collaborative work activity of a distributed team in three different conditions.

Pre-DCP: using conventional collaboration tools (phone, e-mail, video conference rooms, etc.) as they currently were doing.

Full-DCP: adding the desktop conferencing prototype (audio, video, Show Me).

DCP minus video: subtracting the video channel from the desktop conferencing prototype (audio and Show Me only).

By measuring the team's use of these communication media and analyzing actual work activity across the three conditions, we sought to learn how desktop conferencing, and the video channel in particular, would be used in remote collaboration.

5.1 Background

The team we studied initially consisted of four members distributed across three locations. One member was located in Billerica, Massachusetts, another worked in a building in Mountain View, California, and the remaining two members worked in offices

near each other in a different building in Mountain View (approximately 100 yards away from the other building). The team worked together developing automated software testing tools. They were previously all located together in neighboring cubicle offices at the Billerica site but, for reasons not related to their work on this project, were relocated to these distributed sites a few months before the beginning of the study. During the course of the study, a fifth team member was added at the Billerica site in a cubicle facing that of the other Billerica team member.

Although the team had no formal hierarchy, there were differences in their job responsibilities. The project leader (PL) was located alone in one Mountain View building. The two members located in the other Mountain View building were software developers (SD1 and SD2) who wrote most of the computer code. The customer representative (CR1) in Billerica communicated the customers' needs, requirements, and experiences to the rest of the team. The newly added member in Billerica (CR2) had a job similar to CR1.

Altogether, we studied the team's work activity for 14 weeks. After three weeks in the pre-DCP condition, the desktop conferencing prototype was installed into each team member's workstation. This installation involved inserting a hardware card into each existing workstation, adding a second display screen (except for CR1 who continued to use a single display), outfitting each office with a camera and speakers, and adding software to their system. Due to equipment limitations, we were unable to equip the added member of the team (CR2) with a prototype, but he often joined CR1 in desktop conferences or used CR1's workstation when he was not in the office. The team was studied in the full-DCP condition for seven weeks in an attempt to go beyond the initial novelty effect of introducing a technology to a more routine pattern of use. For the DCP minus video condition, the hardware card and second display screen were removed; the speakers and camera (camera's microphone was used for audio input) were left in their offices. The team was studied in the DCP minus video condition for four weeks.

When operating at 30 video frames per second (fps), a desktop conference (audio, video, Show Me) consumed approximately 1.6 Mbit/s of network bandwidth. The bandwidth demand came mostly from the video stream and could be reduced almost directly in proportion to the requested video frame rate. At 10 fps, desktop conferences could use the existing local area networks without overly disrupting other network traffic. However, dedicated network bandwidth was needed for robust connections between the Billerica and Mountain View sites. A 0.5 Mbit/s link was leased that provided enough bandwidth to support conferencing at 5 fps. Because of this limitation, the default video frame rate was set to 5 fps for all desktop conferences among the team, although any user could change this rate before starting a conference. This video frame rate was noticeably less lively than the 30 fps used in full-motion video, and we wanted to learn if that video rate was usable.

5.2 Observation Methodologies

A variety of observational methods were used to obtain information from several different perspectives for this study:

- Phone calls received from other team members were automatically logged (num-

ber of calls, average duration) by the corporate internal phone system.

- Electronic mail messages sent to the other team members and to the team's distribution list were collected.
- Desktop conferences made using the prototype were automatically logged (start & stop time, who was being conferenced, conference parameters, etc.) by software built into the prototype.
- Face-to-face meetings among team members at the Mountain View site were logged by the team members.

This data provided an opportunity to observe many differences in the use of these communication media across the three conditions.

In addition to the quantitative data collected, we videotaped selected samples of collaborative activity in each of the three conditions. After each videotape was made, the participants were always given the option of erasing the videotape if they were uncomfortable with having a record of that interaction. The videotape data captured 19 interactions including examples of: all team video conference room meeting, all team face-to-face meeting, two-person face-to-face meeting, three-way phone conference, two-way desktop conference, three-way desktop conference (involving all five team members), four-way desktop conference, and two-way Show Me conference (with phone audio). These tapes were analyzed by a multi-disciplinary group that included the designers of the prototype, a psychologist, and user interface designers. The group studied the tapes in the tradition of interaction analysis [Tatar, 1989] to understand how the team accomplished their collaborative work and compared similar types of activity across different instances collected on videotape.

Furthermore, we interviewed each team member individually to gather their perceptions about their work activities at various stages during the three conditions of the study:

- at the beginning of the study, to understand their existing work activity;
- before the installation of the desktop conferencing prototype, to test their expectations of how they would use the prototype;
- mid-way through the use of desktop conferencing prototype, to see how they were responding to it;
- just prior to removing the video capability, to test their expectation of how that would affect their use of the tools; and
- at the end of the study, to review their perceptions of the experience.

5.3 Limitations Of The Data

It is important to note the context and limitations of these data to appropriately understand and apply the results from this study. Since this team previously had been co-located, they were in some respects not representative of distributed groups in general. On the other hand, they also knew how they had interacted when co-located and could evaluate how well the prototype tools fulfilled those interactional needs. Since video is believed to be especially useful in supporting social activities [Gale, 1990], the team's existing social relationships made them a good candidate to demonstrate any benefit

from that capability.

Although we intended to collect data that would provide a clean comparison among the three conditions, several factors combined to complicate the data collection and the analyses that can be drawn from the data. In general, the quantitative data were relatively sparse and had large variances, making it less likely that we could demonstrate statistically significant differences. Several factors contributed to the variance in the data.

Company holidays shortened weeks 1 and 5 by one day. Training classes or travel caused one or more team members to be away from the office for an entire week during weeks 3, 4, 7, 8, and 11 of the study. These absences not only affected the data, but also caused some adjustments in the duration of the three conditions. A total of 15 other individual days of absence (e.g., illness, day off) occurred during the study. During the third week of the full-DCP condition (week 6), both CR1 and CR2 from Billerica traveled to Mountain View to meet with the team and others there. Besides affecting the data collected for that week, the visit had an effect on the progress and nature of the team's subsequent work.

Also, several uncertainties were discovered in the phone, e-mail, and face-to-face meeting logging. Problems with the automatic phone logging of the Billerica team members resulted in lost data. Consequently, our analyses are based only on the data of calls received by Mountain View team members. After the study started, PL realized that he was logging e-mail from only one of two sources that he sends mail from, resulting in some lost e-mail data from him. In addition, we allowed the participants to delete any e-mail messages that they did not want us to see before making their e-mail logs available to us. Although this added some uncertainty to the e-mail data collected, we felt that it was a worthwhile trade-off in order to accommodate their participation in the study. Since we did not provide the new team member (CR2) with a desktop conferencing prototype, we did not include any quantitative data collected on CR2's activity in the analyses.

Because we were relying on the team members to report their face-to-face meetings, some meetings were probably recorded inaccurately or not at all. These meeting logs also had some inherent uncertainty since individuals reported the same meeting differently (different start and stop times, different participants). We reminded them to log their meetings throughout the study to counteract any tendency to overlook their self-logging over time.

While all of these factors frustrated our attempt to get clean, quantitative data to compare among the three conditions, they were accommodated to preserve the team's actual working activity with minimal disruption from the study. The quantitative data were used to identify trends and raise issues that we could examine through the other qualitative data that we had collected. Even though these variations limit some of the claims we can make based on the quantitative data, we accept them as a characteristic of studying actual work activity, rather than studying behavior in an isolated, laboratory setting.

6. Analyzing the use of desktop conferencing

The quantitative and qualitative data were analyzed for any patterns or changes across the three conditions. We conducted statistical tests on the quantitative data to identify any significant differences across the conditions. We used the videotape and interview data to discover any changes across the conditions and to help explain patterns that were observed in the quantitative data. These analyses revealed that desktop conferencing:

- did not increase overall interactive communication usage,
- was used more heavily when video was available,
- substituted for e-mail messages,
- may have substituted for shorter face-to-face meetings,
- changed the usage pattern of phone calls,
- was a novel collaboration setting, and
- afforded being aware of where people were looking (gaze awareness).

6.1 No Increase In Overall Interactive Communication Usage

The data indicate that introducing desktop conferencing did not systematically change the total amount of interactive communication (face-to-face meetings, phone calls, desktop conferences) for the team. A measure of usage for each medium of interactive communication was calculated by multiplying the duration of each interaction by the number of people involved. Figure 5 graphs the combined measures of usage for the interactive communication media per week. The most visible feature of this graph is the spike in week 6. This was the week when the team members from Billerica traveled to Mountain View to meet face-to-face together with the team.

Besides the spike in week 6, there is no other visible pattern in the combined usage of interactive media throughout the three conditions. An analysis of variance showed no significant differences in the total measure of usage across the three conditions ($p < 0.17$, where less than 0.05 indicates significance). This lack of an effect is itself a finding, suggesting that the additional desktop conferencing capability did not cause the team to

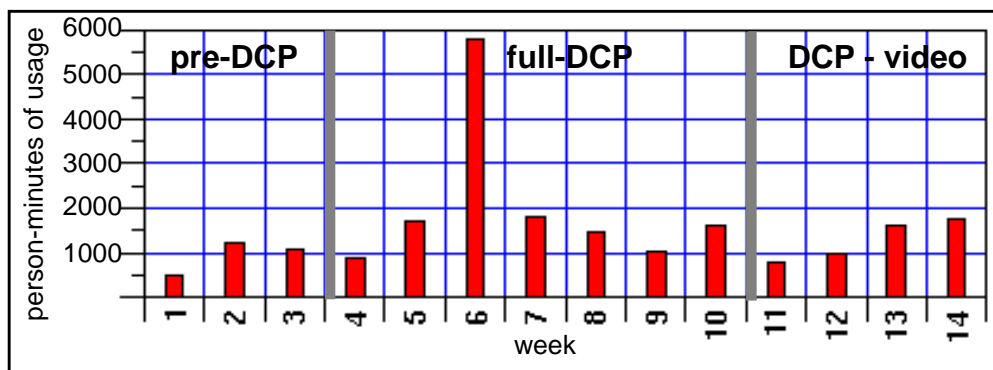


Figure 5. Total measure of usage for all forms of interactive communication

Total number of person-minutes of face-to-face meetings, phone calls, and desktop conferences combined per week. Note that weeks 1 and 5 only had four working days due to company holidays.

spend more time in interactive communication. Instead, desktop conferences apparently substituted for other forms of communication. A closer look at the data provides some insights into the usage relationships among the communication media.

6.2 Video Determined How Much Desktop Conferencing Was Used

The data clearly show that the presence of the video capability determined how much the desktop conferencing prototype was used. Figure 6 plots each of the communication media for the 14 weeks of the study (excluding the atypical week 6). For e-mail, the number of e-mail messages was counted as a measure of usage. The use of the desktop conferencing prototype significantly decreased during the DCP minus video condition when the video capability was taken away ($p < 0.02$). This result indicates that the video capability was the determining factor in whether the team used the desktop conferencing prototype. Why did the users like using video so much?

Interviews with the team indicated that they strongly liked the video because they could see each others' reactions, monitor if they were being understood, and engage in more social, personal contact through video. Besides using desktop conferencing for technical discussions, they also reported using it for informal chatting. Some team members expressed that having the video improved the communication among the team. Turning to the videotape data of their use of the desktop conferencing prototype, we could see specific evidence of their use of video that would contribute to these positive perceptions.

Video played a crucial role in facilitating their interaction. It clearly helped remote collaborators interpret long audio pauses. We observed many pauses in desktop conferences, lasting up to 15 seconds, but the participants did not mark them as problematic. The video channel provided visual cues that explained the purpose of the pause (e.g., reading e-mail, looking for some information in the office, looking up at the ceiling while thinking of what to say next, preparing an image to send in Show Me). Without the video channel, these pauses would have been mystifying, as evidenced in video records of phone calls where participants frequently asked for feedback (e.g., "Right?", "OK?").

Other gestures that facilitated their interaction included leaning into the camera when users could not hear what a remote collaborator said (usually prompting a repetition of the utterance) and hand gestures that indicated taking or yielding a turn of talk. Facial and body gestures often communicated whether a person was understanding what was being said, prompting the speaker to either continue explaining or move on to the next topic. The video channel conveyed many of the gestures people use to mediate their speech [Kendon, 1986]. Gestures were also used to express disagreement, as will be discussed in more detail with respect to eye gaze awareness.

We also observed several examples of turn completions in desktop conferencing when one person would complete a sentence or turn of talk for a remote collaborator. Completions are a demonstration of mutual understanding that require tight interaction and coordination among the participants [Wilkes-Gibbs, 1986]. The prototype demonstrated that it can support accomplishing turn completions between remote participants. Completions were notably absent when using the video conference rooms, largely due to

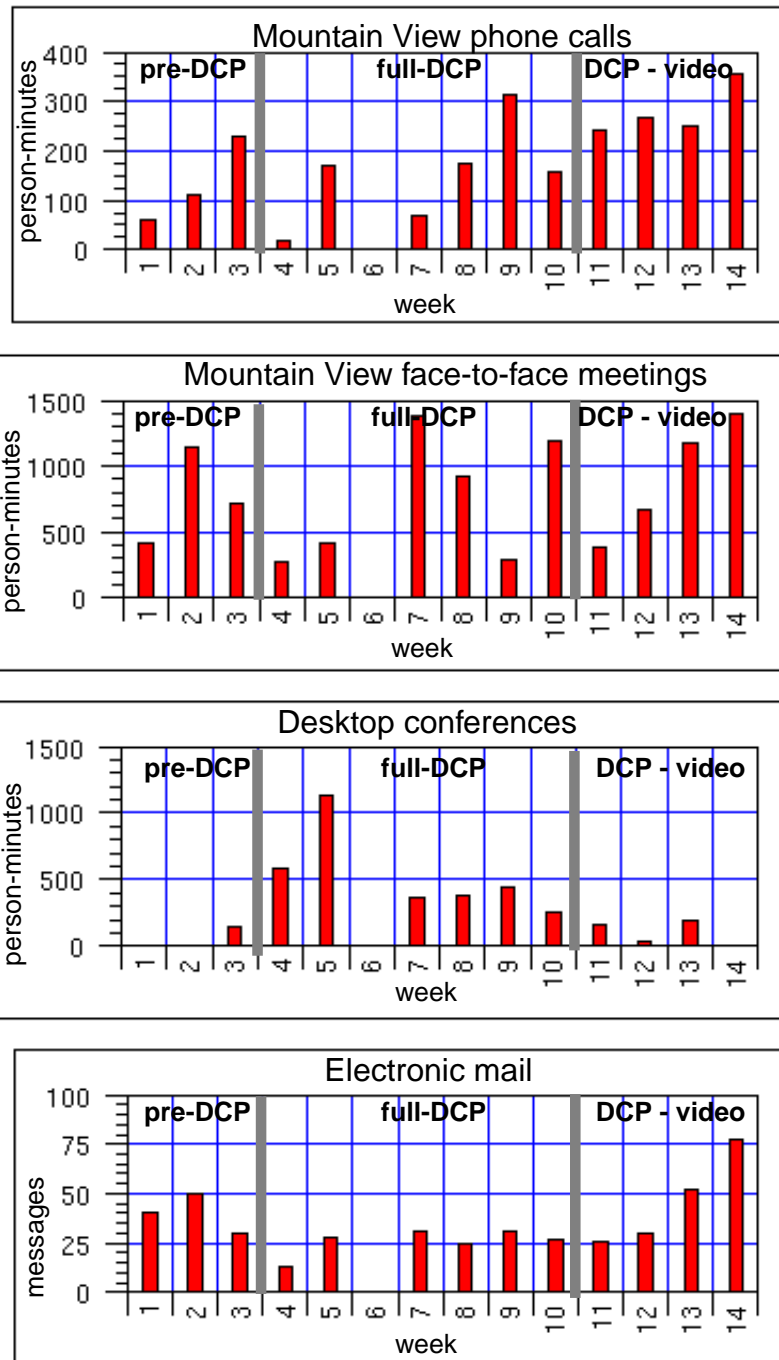


Figure 6. Usage of communication media

Weekly measures of usage of phone calls received by the Mountain View team members, face-to-face meetings held by the Mountain View team members, desktop conferences of the whole team, and electronic mail of the whole team across the three conditions. Phone calls, face-to-face meetings, and desktop conferences are measured in person-minutes. Electronic mail is measured in number of messages. Note that week 6 is eliminated since the team was all together at one site. Because it took a couple days to install the desktop conferencing prototype, some usage was recorded before the entire team was equipped for the full-DCP condition.

the more than half-second audio delay.

The video channel was also used to visually convey information. Shrugs were often demonstrated through the video channel without any accompanying talk indicating indifference or “I don’t know”. In a three-way conference involving all five team members, SD1 rhetorically asks “What does that benefit [this project]?” and emphatically answers the question by synchronizing a gesture indicating “zero” without saying anything else. Occasionally, objects were held in front of the camera to show them to the others. Team members sometimes noticed activities happening in the background through the video channel.

For example, they could often see people walking by at the Billerica site (which had open cubicle offices) and would wave and engage in conversations with them.

The team seemed to use gestures naturally in desktop conferences much as they would in face-to-face interaction. Some of the users' gestures were not transmitted through the video channel because they were not within the camera's field of view, indicating that in some ways the desktop conferencing prototype elicited an illusion of face-to-face interaction beyond what it could actually support. At other times, the team members were aware that they were deliberately using the video channel to convey gestures. In one example, PL noticed someone he knew in Billerica looking in on a desktop conference he was having with CR1. PL waved his hand, but it was not within the camera's field of view. He quickly repositioned his wave within camera view, which finally elicited a response. The data contain evidence that users' activity both built on the familiar face-to-face experience and also accommodated the capabilities of desktop conferencing.

As mentioned earlier, because of network bandwidth limitations between Billerica and Mountain View, the default video frame rate for desktop conferences was set to 5 fps. Although this rate is dramatically less than the 30 fps used in television video, the users found the lower frame rate to be usable for desktop conferencing purposes. There was only one instance (out of 72) where the users chose to increase the video frame rate (to 10 fps). When asked in the interviews about the slow frame rate, they commented that it did not bother them. They did comment on a related problem of having the video image occasionally freeze when the network traffic or computational load was heavy. Under severe loading conditions, images were frozen for several seconds before a new image was received. Users found this to be annoying, although it was sometimes amusing if the frozen image captured a humorous pose of one of the collaborators.

In the interviews just prior to removing the video, all team members anticipated they would hardly use the prototype once the video capability was removed. The prototype's audio quality was considerably worse than the telephone, due to the perceptible delay and echo. While the Show Me shared drawing tool might have motivated continued use of the prototype after removing the video, we did not observe heavy use of Show Me throughout the study. Although we could not collect statistics on the actual use of Show Me, the team apparently had only occasional need to use it. Comments from the interviews indicated that the team found Show Me very satisfying and helpful when they used it. However, the use of Show Me depends on whether the task at hand requires a shared drawing space.

6.3 Desktop Conferencing Substituted For E-mail Messages

We did not expect the availability of desktop conferencing, an interactive communication medium, to have any effect on the use of e-mail, which is asynchronous. However, the e-mail statistics in Figure 6 and Table I show that the average number of e-mail messages per day was significantly lower in the full-DCP condition compared to the pre-DCP or DCP minus video conditions ($p < 0.02$).

Why would the availability of desktop conferencing affect the use of e-mail? One explanation offered in the interviews is that they would sometimes choose to respond to

	total # msgs.	avg. msgs. / day	s.d.	avg. basic / day	avg. reply / day
pre-DCP	120	8.6	5.5	3.0	3.6
full-DCP	155	5.3	3.0	2.7	2.0
DCP - video	185	9.3	6.9	4.5	5.2

Table I. Overall e-mail statistics across conditions

Total number of e-mail messages, average number of messages per day, standard deviation of the average, and average basic and reply messages per day across the pre-DCP, full-DCP, and DCP minus video conditions.

an e-mail message by desktop conferencing instead of replying with e-mail. One member said that he sometimes started composing a reply e-mail message, but then decided to respond with a desktop conference instead and discarded the unfinished e-mail reply. Some team members also commented that they disliked using e-mail when handling certain topics because it generated many messages back and forth before resolving an issue. Issues that might require several cycles of e-mail messages could be easily and quickly resolved in an interactive group desktop conference. The availability of desktop conferencing might have obviated several cycles of e-mail traffic.

We tested these explanations by reviewing the e-mail data to count the number of “reply” e-mail messages compared to the number of “basic” messages (those not in reply to a previous message) across the three conditions, shown in Table I. The data show that the proportion of reply messages was lower in the full-DCP condition than the other two conditions, but this pattern was not statistically significant ($p < 0.27$).

Some team members also mentioned that the team rarely used e-mail among themselves when they were located together in Billerica (except when trying to avoid personal contact with someone). After moving to the three different sites, they began using e-mail heavily, especially since the three-hour time difference between Billerica and Mountain View made it difficult to catch remote team members by phone. Comments from the interviews indicate that they did not prefer using e-mail (except to send computer files), but resorted to using it because the other modes of communication were not effective, given the distribution of the team in time and space. The reduction in e-mail usage during the full-DCP condition could indicate that desktop conferencing restored some of the interactions that they had when located at the same site, thereby reducing their reliance on e-mail.

These explanations alone would not explain why using the phone did not offer the same benefits of reducing e-mail use as desktop conferencing. Phone calls, like desktop conferences, afford interactive rather than asynchronous communication, but they do not allow visual contact with the remote party. Perhaps the novelty effect of introducing a new technology (desktop conferencing) attracted the team to use it in ways that they did not use an existing technology (the telephone). However, the data show that the use of desktop conferencing did settle down after the first two weeks of the full-DCP condition, but the diminished use of e-mail stayed relatively constant throughout the full-DCP condition. There is no evidence in the data that the team, having learned the value of substituting interactive communication for e-mail, began using the phone to substitute for e-mail after the video capability was removed from the prototype. These observations

reinforce the role of video in determining the use of communication media.

6.4 Some Substitution For Short Face-to-face Meetings

Although Figure 6 does not exhibit a significant effect on the usage of face-to-face meetings among the Mountain View members, the data do suggest some meetings were being replaced by desktop conferences in the full-DCP condition. In the interviews, all of the Mountain View members perceived that they were having fewer face-to-face meetings during the full-DCP condition. One instance of a desktop conference that substituted for a face-to-face meeting was brought to our attention because the participants requested that we erase the videotape we had made of it. They did not want to have a record of the sensitive personnel issue that they discussed, which they normally would have handled face-to-face but used desktop conferencing because it was available.

Figure 7 shows a graphical comparison between the face-to-face meeting activity between week 2 in the pre-DCP condition with the meeting and desktop conferencing activity in week 5 in the full-DCP condition. For these representative weeks, Figure 7 shows that longer, 3-person face-to-face meetings persisted in the full-DCP condition, but many of the shorter, 2-person face-to-face meetings appeared to be replaced by desktop conferences. The data in Table II indicate that the average duration for face-to-face meetings was slightly longer in the full-DCP condition compared to the pre-DCP and DCP minus video conditions, although high variability in the data precluded statistical significance ($p < 0.60$). This pattern suggests that short face-to-face meetings might have been substituted by desktop conferences. Interviews with the team members confirmed that they felt that longer meetings with more than two people merited the effort to actually meet face-to-face rather than use desktop conferencing.

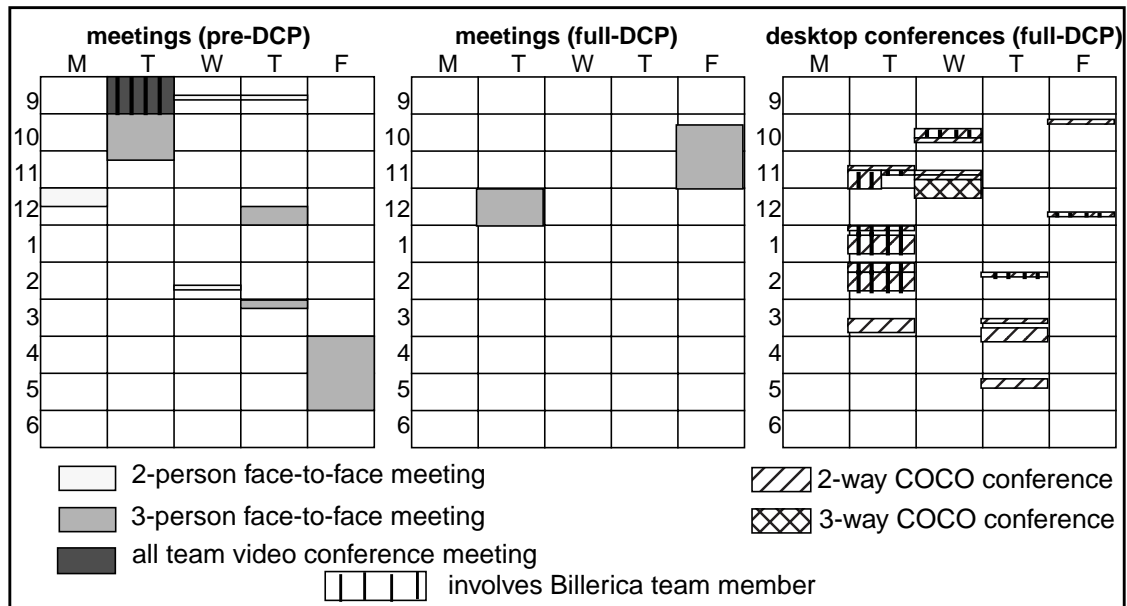


Figure 7. Comparison of face-to-face meetings between pre-DCP and full-DCP

Comparison of face-to-face meetings in pre-DCP condition week 2 (left) with the full-DCP condition week 6 face-to-face meetings (middle) and desktop conferences (right).

	total # mtgs	avg. length (mins.)	s.d.
pre-DCP	26	32.8	27.5
full-DCP	37	43.9	49.6
DCP - video	32	39.2	44.2

Table II. Face-to-face meeting statistics across conditions

Total number of face-to-face meetings, average duration of meetings, and standard deviation of the average duration (indicating variance) across the pre-DCP, full-DCP, and DCP minus video conditions.

Desktop conferencing was also used to increase visual contact between Billerica and Mountain View. Of the 72 desktop conferences logged, 28 (39%) involved team members from both sites. Thus in the six typical weeks in the full-DCP condition, there were 28 cross-site desktop conferences, while in the seven weeks of the pre-DCP and DCP minus video conditions combined there were only four video conference room meetings where collaborators at both sites could see each other. Although the team used desktop conferencing to gain visual access to remote team members, most of the desktop conferences (61%) were among the Mountain View team members. These team members could have walked to the other building to meet face-to-face, but they elected to use the desktop conferencing prototype instead. The data indicate a use of desktop conferencing among collaborators who are separated by a few hundred feet as well as thousands of miles.

In the pre-DCP condition, the team had just started using a weekly one-hour time slot in the video conference rooms to have all-team meetings. In the full-DCP condition, the team never used the slot, and resumed using it two weeks into the DCP minus video condition. The prototype was not designed to support a five-way connection for an all-team desktop conference. However, the team used three-way conferencing several times to have an all-team meeting by having pairs of people share a camera at two sites. Even though the team did not frequently meet all together via desktop conferencing, the sub-team desktop conferences apparently obviated the need for all-team video conference meetings.

The data from this five-person team indicate that the availability of full desktop conferencing eliminated their use of video conference rooms. However, video conferencing between meeting rooms is generally a different kind of collaborative activity than conferencing between personal desktops. Video conferencing rooms allow planned meetings between moderate-sized groups (perhaps ten or more on each side) in an environment that is relatively free from interruptions (e.g., telephone calls, e-mail arrival, impromptu visitors). Desktop conferencing on the other hand allows spontaneous interactions between individuals or small groups where each person has access to the resources of their own workstation and office. Although desktop conferencing was found to eliminate the use of video conference rooms for this small team of five people, we do not believe that desktop conferencing should be generally considered to replace video conference rooms.

6.5 Changes In The Use Of Phone Calls

In the interviews before installing the desktop conferencing prototype, some team members expected they would use a desktop conference for anything they currently did over the phone. In the interviews after they had used the prototype for a couple weeks, all team members reported less phone use when they had the prototype. In contrast to their perceived reduction in phone call use, the measures of usage in Figure 6 do not show any significant effect on the usage of phone calls across the three conditions, and the phone call statistics in Table III show a slight increase in the average number of phone calls per day over the three conditions. The average duration of phone calls was shorter in the full-DCP condition compared to the pre-DCP and DCP minus video conditions. This pattern suggests that longer phone calls may have been substituted by desktop conferences but shorter calls continued to be made by telephone. Wide variation in the relatively sparse data prevented statistical significance ($p < 0.17$).

	total # calls	avg. # / day	avg. length (secs.)
pre-DCP	26	1.9	461.5
full-DCP	78	2.7	348.2
DCP - video	58	2.9	453.3

Table III. Phone call statistics across conditions

Total number of calls, average number of calls per day, and average duration of calls across the pre-DCP, full-DCP, and DCP minus video conditions.

Interviews with the participants provided some reasons why participants continued to use the phone rather than desktop conferencing for short calls. The prototype was not optimized for quick performance and starting a desktop conference could take about a half-minute. For quick calls (e.g., “Ready to go to lunch?”, checking if someone is in the office before visiting or desktop conferencing), the users did not want to incur the overhead of starting a desktop conference since using the phone would be much quicker. Audio quality was also relatively poor compared to the phone due to the delay and echo. Desktop conferencing was also perceived to be inappropriate in some situations. SD1 commented that it seemed “decadent” to make a desktop conference to SD2 (who was located just a few offices away) instead of calling or just walking down the hall.

6.6 Desktop Conferencing Is A Novel Collaboration Setting

From the analysis of the videotapes, it is clear that desktop conferencing is a distinctly different collaboration setting than meeting face-to-face or talking on the phone. In desktop conferences, all members are located in their own offices where each person has access to his or her own resources and distractions (e.g., phone calls, e-mail arrivals, visitors). By contrast, face-to-face meetings are usually held in conference rooms, where everyone is isolated from their resources, or in one person’s office, where only that person can access her books, phone calls, etc. Consequently, in face-to-face meetings, it is generally considered poor etiquette to take long phone calls or spend much time reading e-mail while other people are waiting for attention. In the desktop conferences that we

analyzed, there were several examples of people reading e-mail and taking phone calls during a desktop conference. They seemed to treat desktop conferencing as a medium for focused interaction (like a phone call or meeting), but also one that tolerated significant amounts of attending to personal distractions. This kind of interaction is similar to the ebb and flow of group and individual activity that occurs when sharing an office or working in a computer-augmented meeting room [Stefik et al., 1987].

There are several reasons why desktop conferencing afforded this type of collaborative activity. Because all participants are located in their own offices, if one member attends to a personal distraction, every other member can easily attend to their own personal work while waiting for the conference to refocus. Also, desktop conferencing affords many cues (largely through audio and video) that enable a remote collaborator to make sense of what is happening when one person temporarily stops participating. By contrast, in a phone conversation it is often difficult to interpret long pauses. In addition, some users commented that since they did not have true eye contact with the remote collaborator, they felt that they were slightly detached from them, which allowed attending to personal work.

Although the users of the desktop conferencing prototype found themselves in a novel collaboration setting, they interacted in a very routine and seemingly familiar manner. They smoothly migrated from group interaction to individual work in a way that could not occur in any other medium, yet they did so in a familiar and natural way without marking the activity as novel. We believe that desktop conferencing, largely through the video channel, provided enough cues for participants to interpret the transitions between group interaction and individual work and accommodate a new style of interaction.

Although they were able to accommodate a new style of working in desktop conferences, interview comments indicate that they did not necessarily like it. Several team members found it annoying when someone stopped to take a long phone call or continued doing private work while desktop conferencing. Although the video channel helped them detect such distractions, it still required a delicate social negotiation to try to directly manage them. Just as participants in face-to-face conversation are often reluctant to direct their partner's action (e.g., "Excuse me, you need to wipe off some food smudged on your face"), so desktop conference participants did not feel free to tell their partners to stop doing other work or to reposition their head to be in camera view. It is notable that many of the rules of politeness that govern face-to-face interaction also appear to be in force in desktop conferencing.

The group interaction that occurred in desktop conferencing was notably more like face-to-face meetings than meeting in the commercial video conferencing rooms. Remote collaborators were able to interrupt each other, accomplish turn completions, and time jokes in their conversations. This improved interaction was enabled by reducing the audio delay in the prototype. During the study, the audio delay was measured to vary between 0.32-0.44 seconds (depending on processing and networking loads). This slight improvement over the 0.57 second delay in the video conference rooms was enough to noticeably affect the level of interaction that the collaborators could accomplish.

6.7 Gaze Awareness In Desktop Conferencing

To provide a sense of eye contact in desktop conferencing, the lens of the camera was positioned as close as possible to where the video window of the remote collaborator appeared on the screen. However, all of the team members remarked that they could not establish direct eye contact through the prototype. Rather than introducing half-silvered mirror devices that effectively provide eye contact [Buxton & Moran, 1990], we wanted to see if users could interact comfortably without true eye contact. Ishii and Kobayashi [1992] raised a distinction between eye contact (seeing eye-to-eye) and gaze awareness (being aware of where others are looking). While eye contact is the expected form of interaction from face-to-face meetings, providing each collaborator with a confident sense of gaze awareness may be sufficient to enable effective and comfortable interaction.

We found considerable evidence in the videotapes of desktop conferences that the collaborators had a strong sense of gaze awareness and were able to make use of that information. Figure 8 shows a sequence of video images that show one example of the use of gaze awareness. In a desktop conference between CR1 and PL, CR1 visually expresses continued disagreement with PL by avoiding “looking at” PL. PL continues to talk, and notices that CR1 is avoiding looking at him and gazes and speaks to CR1 in ways that invite CR1 to look up at him. After over 40 seconds of gaze avoidance, PL moves on to another topic, at which point CR1 immediately resumes looking up at him.

In the interviews, we asked whether the team members could tell when collaborators in a desktop conference were looking at them. After just two weeks of use, some members were occasionally uncertain, but by the end of the study everyone said that they could. We believe that if everyone’s equipment is configured to provide near eye contact, users can quickly gain a confident sense of gaze awareness and use that to convey cues in their interactions. Of course, establishing actual eye contact would be ideal in

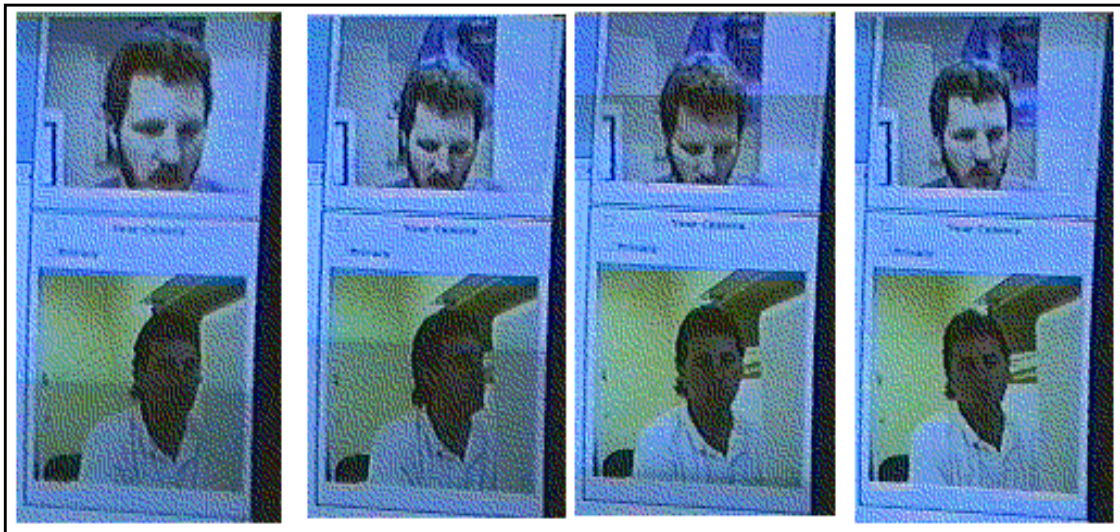


Figure 8. Demonstrating gaze awareness by avoiding “eye contact”

This sequence of images show CR1 (top) and PL (bottom) in a desktop conference. At left, CR1 and PL are “looking at” each other. In the middle two frames, CR1 visually expresses continued disagreement by avoiding “eye contact” with PL for over 40 seconds. After PL moves on to another topic, CR1 resumes “eye contact” with him.

desktop conferencing, but there may be situations where the trade-offs made to accomplish that (e.g., added footprint and volume occupied by half-silvered mirror devices) are not merited.

6.8 Design Implications From The Study Of Desktop Conferencing

This study indicates that, for a working team that is already familiar with each other, desktop conferencing is a useful medium for distributed collaboration. In contrast to studies that did not find a strong effect of a video channel, we found that video was the determining factor in how much desktop conferencing was used. The video provided visual and gestural cues that enabled them to interact smoothly. Gaze awareness among the collaborators in particular was used to convey cues in their interaction. When they used the shared drawing tool, they found it to be valuable in supporting distributed collaboration.

Users commented that the audio quality of the desktop conferencing prototype needed improvement. Because most team members used a speaker for audio output and an open microphone for audio input, the system exhibited a considerable amount of audio echo. Those speaking often heard a delayed echo of their speech as it traveled to others' speakers and back through their microphones. The audio quality was worse in 3-way conferencing, since mixing audio streams introduced even more echo and the increased network traffic caused deletions in the audio streams. Because of these problems, the team often resorted to using telephone audio in three-way conferences. Although we provided headsets that eliminated the audio echo problem, all but one user found them too bothersome to use.

Additionally, our experiences with the desktop conferencing prototype indicated that the phone call model for establishing and managing conferences was too limited. Users were sometimes reluctant to use desktop conferencing to contact others because they could not tell in advance whether a person was available or interruptible. The prototype also did not have the equivalent of a phone answering machine to handle conference requests when no one was there, making it frustrating to try to catch someone. This problem was evidenced in the many unsuccessful conference attempts found in the logs of prototype use. Besides the 72 desktop conferences recorded, 96 attempts to conference were unsuccessful (recipient not in office to receive conference request, recipient's workstation not operational, recipient declined to accept conference request). Mechanisms that integrate desktop conferencing with other forms of communication, such as automatically leaving e-mail or voice mail after an unsuccessful attempt to conference, would be helpful.

This study was also a methodological learning experience in trying to combine different observational perspectives to understand the team's work activity and reaction to the prototype. Although the quasi-experimental structure of the study (three conditions, quantitative measures) did not yield many statistically significant results, it was helpful in identifying patterns and trends that could be explored by analyzing video recorded examples of work activity or interviewing the users for their perceptions. User perceptions elicited by the interviews also helped guide us in selecting samples of videotaped activity on which to focus our analysis. The multiple perspectives also provided a broader understanding of the activity that could not be found in any single observational

method.

The multiple observational perspectives also presented some new problems. Collecting multiple types of data added complexity to the data collection process and resulted in a vast amount of data to sift through. Since our primary commitment was in collecting data on actual team work activity, we did not have the luxury of establishing control conditions and exercising other manipulations often used in laboratory experiments to produce clean quantitative data for statistical comparison.

7. What we learned about multimedia-supported collaboration

What can we learn from our studies about how to design multimedia technology to effectively support collaborative work? Two points clearly came out in each of the three studies presented: 1) users want video connections and 2) the quality of the audio connection is crucial. It is also important to distinguish desktop conferencing from other types of communication media (e.g., face-to-face meetings, video conference room meetings, phone calls) to understand how it and other new multimedia collaboration technologies will be incorporated into everyday use with existing communication technologies.

7.1 Users Want Video

Each of the three studies clearly indicated that the users wanted to have a video capability that allowed them to have visual contact during their interaction. Why do the users want this video capability? Although these studies do not claim to definitively answer this question, they do present a variety of evidence that helps explain why users like video.

The video channel is clearly a valuable resource in mediating interpersonal interaction. Not only does the visual channel provide cues that facilitate the mechanics of turn-taking, but it also naturally affords gestures and other visual information that convey how much is being understood, reasons for pauses in speech, participants' attitudes, and other modifiers (e.g., humor, sarcasm) on what is being said. This support for interactional mechanisms make video-mediated communications more efficient, effortless, and effective. A richer communication channel affords greater mutual understanding among the participants, and we would expect it to help improve the quality of their collaborative work in the long term. Isaacs and Tang [1993] describe more details comparing interactions through face-to-face, phone, and desktop conferences from our data.

Users' comments clearly show that they perceived added value from the video. Besides the benefits we identified in our analyses of video-mediated activity, users reported that the video capability made their interactions more satisfying. These user perceptions should play a major role in guiding the design of technology to support collaboration.

Why did our studies find such a strong effect of video whereas the studies cited earlier [Ochsman & Chapanis, 1974; Gale, 1990] found none? Firstly, the previous studies focused on effects that were associated with the resulting *product* (e.g., quality of the

result, time to complete the task). We found that the video channel had effects on the *process* of interaction (e.g., supporting turn-taking mechanisms, demonstrating understanding and attitudes). Although these effects on interpersonal communication have been hypothesized [Short et al., 1976] and recognized [Gale, 1990] in earlier studies, this paper presents specific evidence from real work activity of how video supports human interaction.

Secondly, the observational methods used in this research differed from those used in the previous studies. The previous studies measured completion times, graded the resulting artifacts, and ranked user assessments of their work. Our research used open-ended surveys and interviews, video-based analyses of work activity, and quantitative measures of actual usage. Perhaps more importantly, the previous studies analyzed the activity of artificial groups working on contrived tasks. The studies presented in this paper examined the activity of actual working groups engaged in their real work activity. Since audio and video tend to have the most effect on social, interpersonal communication, those effects would be most noticeable among a group in which social and personal relationships were well developed and exercised. Our ability to see how video supports social interaction was a direct result of studying actual working activity that had real social elements in it.

The value of video that we observed did not even include one of the inherent strengths of the video media. Video is good for showing and manipulating three-dimensional objects, such as a component to be manufactured or a volumetric shape to be designed. Since the groups studied in this research worked mainly with documents or computer software, they did not exercise this potential capability of video. We would expect that working teams in a domain that involved physical artifacts would find even greater value in video.

7.2 Audio Is Crucial

In each of our three studies, audio quality was an issue. Audio plays a fundamental role in supporting human interaction and users' expectations of audio are formed by their experiences in face-to-face and phone interactions. Technologies that degrade the audio channel (e.g., delays, echo, incomprehensible audio quality) will disrupt people's ability to smoothly interact with each other. Although the team using our desktop conferencing prototype was willing to endure the degraded audio to have the video capability, it was clearly the aspect they most wanted to see improved in the prototype.

Although the ideal is to strive for high fidelity audio and video, our experiences confirm that audio is relatively more important than video in supporting collaboration. Our desktop conferencing prototype made several trade-offs of degrading video performance in order to preserve audio quality. If high network traffic prevented transmitting all of the audio-video data between sites, the video data degraded first (image froze) to allow as much audio data to get through before cutting out. Audio was delivered with minimal delay, even though the audio arrived before the accompanying video image, violating audio-video synchrony. As long as network constraints require trade-offs to conserve bandwidth, our experiences indicate that degrading video quality before degrading audio quality provides a more usable experience.

7.3 Desktop Conferencing Is Not Face-to-face Meeting Is Not Video Conferencing Is Not...

The data from our study of desktop conferencing demonstrated that it substituted for certain amounts of other kinds of interaction (e.g., video conference room meetings, e-mail, some face-to-face meetings). Comments from the video conferencing room survey indicate that some users may like to think that video conference room meetings substitute for face-to-face meetings. However, these findings should not be taken to imply that desktop conferencing could completely replace face-to-face meetings, video conference room meetings, e-mail, or any other form of interaction. As discussed earlier, desktop conferencing is a distinct setting for collaboration and is unlikely to completely replace existing forms of interaction. The adoption of video conferencing rooms and other multimedia technology has suffered from marketing myths that promote them as replacements for face-to-face interaction [Egido, 1990].

Rather, we should strive to understand how new forms of interaction can be integrated with the existing ones into people's day-to-day work. By understanding how these new technologies augment, complement, and interact with people's existing work practice, we can design new technology that can be smoothly and naturally adopted. As we develop new technology for collaboration, more research is needed to understand existing collaborative practice as well as how users respond to the new technology in the context of their actual work. More research is needed into new issues that these technologies raise, such as the privacy concerns of having ubiquitously available audio and video and how to apply multimedia support to collaboration settings that are *non-cooperative*. By iteratively cycling between developing new technology and studying how people actually use that technology, we can both design better technology that is matched to users' needs and increase our understanding of human work activity.

Acknowledgements

We would like to acknowledge the other members of the Conferencing and Collaboration (COCO) group: David Gedye, Amy Pearl, Alan Ruberg, and Trevor Morris. They helped build the desktop conferencing prototype and provided many forms of support for this study. We thank the other participants in our regular video analysis sessions for the many insights and observations that they contributed: Monica Rua, Hagan Heller, Todd Macmillan, and Tom Jacobs. We thank Randy Smith for reviewing this paper. The Digital Integrated Media Environment (DIME) group in Sun Microsystems Laboratories, Inc. developed the SBus card that enabled the COCO conferencing prototype. This research was conducted at Sun Microsystems Laboratories, Inc. We especially thank our anonymous participants in the study for giving us generous access to their daily work activity.

References

- Buxton, Bill and Tom Moran, "EuroPARC's Integrated Interactive Intermedia Facility (IIIF): Early Experiences," *Multi-User Interfaces and Applications*, S. Gibbs and A. A. Verrijn-Stuart (Eds.), Amsterdam: Elsevier Science Publishers B.V., 1990, pp. 11-

- Egido, Carmen, "Teleconferencing as a Technology to Support Cooperative Work: Its Possibilities and Limitations," *Teamwork: Social and Technological Foundations of Cooperative Work*, Jolene Galegher, Robert E. Kraut, and Carmen Egido (Eds.), Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, 1990, pp. 351-371.
- Fish, Robert S., Robert E. Kraut, Robert W. Root, and Ronald E. Rice, "Evaluating Video as a Technology for Informal Communication," *Proceedings of the Conference on Computer Human Interaction (CHI) '92*, Monterey, CA, May 1992, pp. 37-48.
- Francik, Ellen. Susan Ehrlich Rudman, Donna Cooper, and Stephen Levine, "Putting Innovation to Work: Adoption Strategies for Multimedia Communication Systems," *Communications of the ACM*, Vol. 34, No. 12, December 1991, pp. 53-63.
- Gale, Stephen, "Human aspects of interactive multimedia communication," *Interacting with Computers*, Vol. 2, No. 2, 1990, pp. 175-189.
- Gale, Stephen, "Desktop video conferencing: Technical advances and evaluation issues," *Computer Communications*, Vol. 15, No. 2, October 1992, pp. 517-526.
- Heath, Christian and Paul Luff, "Disembodied Conduct: Communication Through Video in a Multi-media Office Environment," *Proceedings of the Conference on Computer Human Interaction (CHI) '91*, New Orleans, LA, April/May 1991, pp. 99-103.
- Isaacs, Ellen and John C. Tang, "What Video Can and Can't Do for Collaboration," *Conference on Computer-Human Interaction (INTERCHI '93)*, Amsterdam, Netherlands, April 1993, submitted.
- Ishii, Hiroshi and Minoru Kobayashi, "ClearBoard: A Seamless Medium for Shared Drawing and Conversation with Eye Contact," *Proceedings of the Conference on Computer Human Interaction (CHI) '92*, Monterey, CA, May 1992, in press.
- Kendon, Adam, "Current Issues in the Study of Gesture," in *The Biological Foundations of Gestures: Motor and Semiotic Aspects*, Jean-Luc Nespoulous, Paul Perron, and Andre Roch Lecours (Eds.), Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, 1986, pp. 23-47.
- Krauss, Robert M., Connie M. Garlock, Peter D. Bricker, and Lee E. McMahon, "The Role of Audible and Visible Back-Channel Responses in Interpersonal Communication," *Journal of Personality and Social Psychology*, Vol. 35, No. 7, 1977, pp. 523-529.
- Masaki, Shigeki, Naobumi Kanemaki, Hiroya Tanigawa, Hideya Ichihara, and Kazunori Shimamura, "Personal Multimedia-multipoint Teleconference System for Broadband ISDN," *High Speed Networking, III*, O. Spaniol and A. Danthine (Eds.), Amsterdam: Elsevier Science Publishers B.V. 1991, pp. 215-230.
- Minneman, Scott L. and Sara A. Bly, "Managing a trois: a study of a multi-user drawing tool in distributed design work," *Proceedings of the Conference on Computer Human Interaction (CHI) '91*, New Orleans, LA, April/May 1991, pp. 217-224.
- Ochsman, Robert B. and Alphonse Chapanis, "The Effects of 10 Communication Modes on the Behavior of Teams During Co-operative Problem-solving," *International Journal of Man-Machine Studies*, Vol. 6, 1974, pp. 579-619.
- Olson, Margrethe H. and Sara A. Bly, "The Portland Experience: A Report on a Distributed Research Group," *International Journal of Man-Machine Systems*, Vol. 34, No.

- 2, February 1991, pp. 211-228. Reprinted: *Computer-supported Cooperative Work and Groupware*, Saul Greenberg (Ed.), London: Academic Press, 1991, pp. 81-98.
- Root, Robert W., "Design of a Multi-Media Vehicle for Social Browsing," *Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR, September 1988, pp. 25-38.
- Sacks, H., E. Schegloff, and G. Jefferson, "A simplest systematics for the organization of turn-taking for conversation," *Language*, Vol. 50, 1974, pp. 696-735.
- Short, John, Ederyn Williams, and Bruce Christie, *The Social Psychology of Telecommunications*, London: John Wiley & Sons, 1976.
- Smith, Randall B., Tim O'Shea, Claire O'Malley, Eileen Scanlon, and Josie Taylor, "Preliminary experiments with a distributed, multi-media, problem solving environment," *Proceedings of the First European Conference on Computer Supported Cooperative Work: EC-CSCW '89*, London, UK, September 1989, pp. 19-34. Reprinted: *Studies in Computer Supported Cooperative Work: Theory Practice and Design*, J. Bowers and S. Benford (Eds.), Amsterdam: Elsevier Science Publishers B.V., 1991.
- Stefik, Mark, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman, "Beyond the chalkboard: Computer support for collaboration and problem solving in meetings," *Communications of the ACM*, Vol. 30, No. 1, January 1987, pp. 32-47. Reprinted: *Computer-Supported Cooperative Work: A Book of Readings*, Irene Greif (Ed.), San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1988, pp. 335-366.
- Stults, Robert, Steve Harrison, and Scott Minneman, "The Media Space - experience with video support of design activity," *Engineering Design and Manufacturing Management*, Andrew E. Samuel (Ed.), Amsterdam: Elsevier Science Publishers B.V., 1989, pp. 164-176.
- Tang, John C., "Findings from Observational Studies of Collaborative Work," *International Journal of Man-Machine Studies*, Vol. 34, No. 2, February 1991, pp. 143-160. Reprinted: *Computer-supported Cooperative Work and Groupware*, Saul Greenberg (Ed.), London: Academic Press, 1991, pp. 11-28.
- Tang, John, "Involving Social Scientists in the Design of New Technology," *Taking Software Design Seriously: Practical Techniques for Human-Computer Interaction Design*, John Karat (Ed.), Boston: Academic Press, 1991, pp. 115-126.
- Tang, John C. and Scott L. Minneman, "VideoDraw: A Video Interface for Collaborative Drawing," *ACM Transactions on Information Systems*, Vol. 9, No. 2, April 1991, pp. 170-184.
- Tatar, Deborah, "Using Video-Based Observation to Shape the Design of a New Technology," *SIGCHI Bulletin*, Vol. 21, No. 2, October 1989, pp. 108-111.
- Watabe, Kazuo, Shiro Sakata, Kazutoshi Maeno, Hideyuki Fukuoka, Toyoko Ohmori, "Distributed Multiparty Desktop Conferencing System: MERMAID," *Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA, October 1990, pp. 27-38.
- Wilkes-Gibbs, Deanna, *Collaborative Processes of Language Use in Conversation*, Ph.D. dissertation, Stanford University, 1986.
- Williams, Ederyn, "Experimental Comparisons of Face-to-Face and Mediated Communication: A Review," *Psychological Bulletin*, Vol. 84, No. 5, 1977, pp. 963-976.

An Overview of the Spring System

(In IEEE COMPCOM '94)

Introduction by Jim Mitchell

The Spring operating system project grew out of a desire to have a better implementation of UNIX®. In 1987, as part of an agreement between Sun Microsystems and AT&T to merge the System V and Berkeley (BSD) "strains" of UNIX®, Sun and AT&T agreed to collaborate on a project to "reimplement UNIX in an object-oriented fashion". That project consisted of only a few initial meetings and then was forgotten, but Sun had begun to build a team of OS experts, and we wanted to do something significant.

We decided to push the state of the art in operating systems. We would develop a highly modular, extensible, microkernel-based, distributed, object-oriented operating system. The main idea was that one should be able to build a distributed system (almost) as easily as a system on a single computer, and that most of the OS programming could be done without having to endure the constraints of kernel-level programming. Such a system could be extended by anyone, not just by kernel programmers, and not just by the company that produced the OS. Our intent was to get around the problems that most operating systems (still) have: they are hard to change; only the company that produces them can change them; and they become large and monolithic as more and more functionality is added.

By 1994, we had developed the Spring system and this paper was written to give an overview of the system. The system was architected and built by a small number of people: Michael Powell was the chief architect; Jim Mitchell managed the project; Graham Hamilton was the chief implementation architect and a main implementor; Yousef Khalidi, Mike Nelson, Peter Madany, and Pano Kougiouris designed many of the system's key parts such as virtual memory and file systems; and Sanjay Radia designed and built the federated naming system. Jonathan Gibbons designed and implemented IDL, which is described below and in the paper; and Peter Kessler built all the stub machinery that formed the connective tissue of Spring.

Spring was an open system that could be extended by adding new types of objects. Only the microkernel ran in kernel mode, so adding a new system component did not require being a kernel programmer. To make the notion of objects fit such an open, multilingual system, we (re)invented the idea of an Interface Definition Language (IDL) that was used to describe the parts of the system. One could compile an IDL interface to produce an object-oriented "stub" into which one's implementation could then be plugged. IDL interfaces can use (interface) inheritance as a way of specializing OS interfaces, a feature that later appeared in Java. When the Object Management Group (OMG) needed a way to talk about distributed objects for their CORBA architecture, they adopted IDL for that purpose.

Spring itself never became a product, but some of the technology developed as part of the project (such as IDL) was adopted in other areas. We invented a fast way to short circuit calls between objects when they are on the same machine rather than different machines connected by a network. This facility (Spring doors) has been incorporated in the Solaris™ Operating Environment for fast process-to-process communication. And, of course, IDL was used to allow CORBA objects to be written in Java and communicate with other CORBA objects.

With the enormous growth of the worldwide web and the growth of standards for it like HTTP, HTML, XML, etc., the importance of a new OS structure was considerably diminished. The result was that we stopped trying to make Spring a product and the team became part of the Java Software organization, and a number of the Spring ideas have been used to enhance the Java environment in areas like RPC (Java™ IDL), naming (JNDI) and others.

"An Overview of the Spring System." © 1994 Sun Microsystems, Inc. and IEEE. Reprinted, with permission, from Proceedings of Compcon, Spring 1994, February 1994.

PUBLICATIONS:

"An Overview of the Spring System" originally appeared in the Proceedings of Compcon Spring 1994. Copyright© February 1994.

"Subcontract: A Flexible Base for Distributed Programming" originally appeared in the Proceedings of the 14th Symposium on Operating Systems Principles, Asheville NC, December 1993. Copyright© 1993 ACM.

"A Client-Side Stub Interpreter" originally appeared in the Proceedings of ACM Workshop on Interface Definition Languages, January 1994. Copyright© 1994 ACM.

"Using Interface Inheritance to Address Problems in Software Evolution" originally appeared in the Proceedings of the ACM Workshop on Interface Definition Languages, January 1994. Copyright© 1994 ACM.

"The Spring Name Service" originally appeared as Sun Microsystems Laboratories Technical Report SMLI-93-16, October 1993. Copyright© 1993 Sun Microsystems Inc.

"Naming Policies in Spring" originally appeared in the Proceedings of the 1st International Workshop on Services in Distributed and Networked Environments. Copyright (C) 1994 IEEE.

"Persistence in the Spring System" originally appeared in the Proceedings of the 3rd Workshop on Object Orientation in Operating Systems, December 1993. Copyright© 1993 IEEE.

"The Spring Nucleus: A Microkernel for Objects" originally appeared in the Proceedings of the 1993 Summer Usenix Conference, June 1993. Copyright© 1993 USENIX.

"The Spring Virtual Memory System" originally appeared as the Sun Microsystems Laboratories Technical Report SMLI-93-9, March 1993. Copyright© 1993 Sun Microsystems Inc.

"A Flexible External Paging Interface" originally appeared in the Proceedings of the Usenix conference on Microkernels and Other Architectures, September 1993. Copyright© 1993 USENIX.

"The Spring File System" originally appeared as the Sun Microsystems Laboratories Technical Report SMLI-93-10, March 1993. Copyright© 1993 Sun Microsystems Inc.

"Extensible File Systems in Spring" originally appeared in the Proceedings of the 14th Symposium on Operating Systems Principles, Asheville NC, December 1993. Copyright© 1993 ACM.

"An Implementation of UNIX on an Object-oriented Operating System" originally appeared in the Proceedings of the 1993 Winter Usenix Conference, January 1993. Copyright© 1993 USENIX.

"High Performance Dynamic Linking Through Caching" originally appeared in the Proceedings of the 1993 Summer Usenix Conference, June 1993. Copyright© 1993 USENIX.

"A Framework for Caching in an Object Oriented System" originally appeared in the Proceedings of the 3rd Workshop on Object Orientation in Operating Systems, December 1993. Copyright© 1993 IEEE.

An Overview of the Spring System

James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler,
Yousef A. Khalidi, Panos Kougouris, Peter W. Madany, Michael N. Nelson,
Michael L. Powell, and Sanjay R. Radia

Sun Microsystems Inc.
2550 Garcia Avenue, Mountain View Ca 94303.

Abstract

Spring is a highly modular, distributed, object-oriented operating system. This paper describes the goals of the Spring system and provides overviews of the Spring object model, the security model, and the naming architecture. Implementation details of the Spring microkernel, virtual memory system, file system, and UNIX emulation are supplied.

1 Introduction

What would you do if you were given a clean sheet on which to design a new operating system? Would you make the new OS look the same as some existing system or different?

If you choose to make it look like UNIX, for example, then a better implementation had better be a primary goal. Changing the system as seen by application programs would, however, be a very bad thing to do, since you are supposedly making it look the same as UNIX in order to run existing software. In fact, if you take this route, you will be strongly pressured to make the new, improved system *binary compatible* with the existing one so that users can run all their existing software. Any new functionality that you would like to include would have to be done as a strict addition to the system's existing Application Programming Interfaces (APIs).

If you choose to make the new system different than any existing system, then you had better make it such an improvement over them that programmers will be willing to learn a new set of APIs to take advantage of its improved functionality. Indeed, you will have to convince other companies to adopt and support your new APIs so that there will be sufficient future sales of systems with the

new APIs to warrant software development investments by application developers.

Because the opportunity to begin afresh in OS design is increasingly rare, the Spring project has chosen to be different and to develop the best technology we could. However, we decided that we would innovate only where we could achieve large increases over existing systems and that we would try to keep as many as possible of their good features.

What are the biggest problems of existing systems? From Sun's point of view as a supplier of UNIX system technology in our Solaris products, the major issues are:

- the cost of maintaining, evolving, and delivering the system, including kernel and non-kernel code (e.g., window systems),
- a basis for security that is not particularly flexible, easy to use or strongly secure,
- the difficulty of building distributed, multi-threaded applications and services,
- the difficulty of supporting time-critical media (audio and video), especially in a networked environment,
- the lack of a unified way of locating things by name (e.g., lookup is done differently for files, devices, users, etc.).

However, we wanted to keep a number of features that have proven themselves in one or more systems; for example,

- good performance on a wide variety of machines, including multi-processor systems,
- memory protection, virtual memory, and mapped file systems,
- access to existing systems via application compatibility and network interoperability (e.g., standard protocols and services),

- window systems and graphical user interfaces.

Sun's belief in open systems means that we would like to include *extending the system* by more than one vendor as an important aspect of *evolving* it.

When we looked at these lists, we immediately decided that the Spring system should have a strong and explicit architecture: one that would pay attention to the interfaces between software components, which is really how a system's structure is expressed. Our architectural goal for Spring then became

- Spring's components should be defined by *strong interfaces* and it should be *open, flexible and extensible*

By a strong interface we mean one that specifies *what* some software component does while saying very little about *how* it is implemented

This way of stating our purpose led us to develop the idea of an Interface Definition Language (IDL) [15] so that we could define software interfaces without having to tie ourselves to a single programming language, which would have made the system less open. We also believed that the best way to get many of the system properties we wanted was to use an object-oriented approach.

The marriage of strong interfaces and object-orientation has been a natural and powerful one. It helps achieve our goals of openness, extensibility, easy distributed computing, and security. In particular, it has made the operation of invoking an operation on an object one that is *type safe*, *secure* (if desired), and *uniform* whether the object and its client are collocated in a single address space or machine or are remote from one another.

We have used a microkernel approach in concert with IDL interfaces. The Spring Nucleus (part of the microkernel) directly supports secure objects with high speed object invocation between address spaces (and by a system extension, between networked machines). Almost all of the system is implemented as a suite of *object managers* (e.g., the file system, which provides file objects) running in non-kernel mode, often in separate address spaces, to protect themselves from applications (and from one another). Consequently, it is as easy to add new system functionality as it is to write an application in Spring, and all such functionality is inherently part of a distributed system.

Object managers are themselves objects: for example, a file system is an object manager that supports an operation for opening files by name. The file objects that it returns from *open* operations are generally implemented as part of the same object manager because it is convenient and natural to do so. Because of the similarity of an object manager and the traditional notion of a *server* (e.g., a file

server), we use the two terms interchangeably in this paper.

The remainder of this paper will discuss

- IDL,
- the model and implementation of objects in Spring,
- the overall structure of the Spring system,
- the Spring Nucleus,
- the implementations of distributed object invocation, security, virtual memory, file systems, UNIX compatibility, and unified naming.

We will finish by drawing some conclusions from our experience designing and implementing the system.

2 Interface Definition Language (IDL)

The Interface Definition Language developed by the Spring project is substantially the same as the IDL that has been adopted by the Object Management Group as a standard for defining distributed, object-oriented software components. As such, IDL "compilers" are or have been implemented by a number of companies.

What does an IDL compiler do? After all, interfaces are not supposed to be implementations, so what is there to compile? Typically, an IDL compiler is used to produce three pieces of source code in some chosen target implementation language, e.g., C, C++, Smalltalk, etc.:

1. *A language specific form of the IDL interface:* For C and C++ this is a header file with C or C++ definitions for whatever methods, constants, types, etc. were defined in the IDL interface. We will give an example below.
2. *Client side stub code:* Code meant to be dynamically linked into a client's program to access an object that is implemented in another address space or on another machine.
3. *Server side stub code:* Code to be linked into an object manager to translate incoming remote object invocations into the run-time environment of the object's implementation.

These three outputs from an IDL compiler enable clients and implementations in a particular language, e.g., C++, to treat IDL-defined objects as if they were just objects in C++. Thus, a programmer writing in C++ would use an IDL-to-C++ compiler to get C++ header files and stub code to define objects as if they were implemented in C++. At the same time, the object's implementation might be written in C and would, therefore, have used an IDL-to-C compiler to generate the server side stub code to transform

incoming calls into corresponding C procedure invocations on the C “objects” corresponding to the IDL objects.

2.1 An example

To give the flavor of IDL, Figure 1 shows an example of IDL use to define the Spring IO interface. For the purposes of this overview, details have been elided, but the example is derived from an actual use of IDL.

```
interface io {
    raw_data read(in long size) raises (access_denied, alerted,
                                         failure, end_of_data)

    void write(in raw_data data) raises (access_denied, alerted,
                                         incomplete_write, failure, end_of_data)
};
```

FIGURE 1. IO Interface in IDL

The interface defines objects of type *IO*. In this example, any *IO* object has two operations defined on it, *read* and *write*. The *read* operation takes a parameter, *size*, of type *long*, and returns an object of type *raw_data*. The *write* method returns nothing (*void*) and takes a single argument, *data*, whose type is *raw_data*.

As noted above, instead of returning normally, a method may raise one of a number of defined exceptions.

A complete description of IDL is given in [15].

3 Objects in Spring

Although all Spring interfaces are defined in IDL, IDL says nothing about how operations on an object are implemented, or even how an operation request should be conveyed to an object.

The users of an object merely invoke operations defined in its interface. How and where the operation is actually performed is the responsibility of the object run-time and of the object implementation. Sometimes the operation will be performed in the same address space as the client, sometimes in another address space on the same machine, sometimes on another machine.

We will often use the phrase “invoke an object” as a short form for “invoke an operation on an object”.

3.1 Server-based objects

Many objects in Spring are implemented in servers that are in different address spaces from their clients. We pro-

vide special support for these kinds of objects by automatically generating *stubs* (see Section 2) which take the arguments for these calls and marshal them for transmission to the server and which unmarshal any results and return these to the client application. These stubs use our subcontract mechanism (see Section 3.3) to communicate with the remote server.

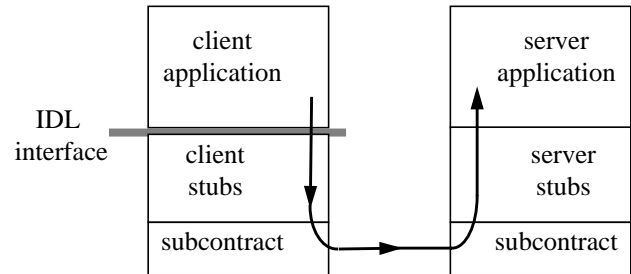


FIGURE 2. A call on a server-based object

Typically, server-based objects use the Spring *doors* communication mechanism (see section 5.1) to communicate between the client and the server. Most subcontracts optimize the case when the client and the server happen to be in the same address space by simply performing a local call, rather than calling through the kernel.

3.2 Serverless objects

Spring also supports serverless objects, where the entire state of the object is always in the client’s address space. This implementation mechanism is suitable for lightweight objects such as names or *raw_data*. When a serverless object is passed between address spaces, the object’s state is copied to the new address space. Thus passing a serverless object is more like passing a struct, while passing a server-based object is more like passing a pointer to its remote state.

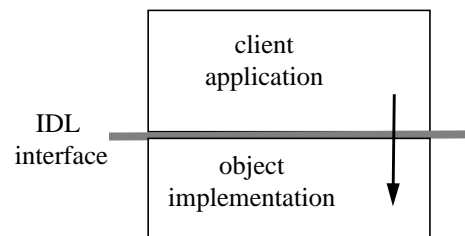


FIGURE 3. A call on a serverless object

3.3 Subcontract

Spring provides a flexible mechanism for plugging in different kinds of object runtime machinery. This mechanism, known as *subcontract*, allows control over how object invocation is implemented, over how object references are transmitted between address spaces, how object references are released, and similar object runtime operations [2].

For example, the widely used *singleton* subcontract provides simple access to objects in other address spaces. When a client invokes a singleton object, the subcontract implements the object invocation by transmitting the request to the address space where the object's implementation lives.

We have also implemented subcontracts that support replication. These subcontracts implement object invocation by transmitting a request to one or more of a set of servers that are conspiring to support a replicated object.

In addition we have used subcontract to implement a number of different object runtime mechanisms, including support for cheap objects, for caching, and for crash recovery.

4 Overall system structure

Spring is organized as a microkernel system. Running in kernel mode are the *nucleus*, which manages processes and inter-process communication, and the *virtual memory manager*, which controls the memory management hardware. The nucleus is entered by a trap mechanism. The virtual memory manager responds to page faults but also provides objects that interact with external pagers (see Section 8.2) and, in this guise, looks like any other object server.

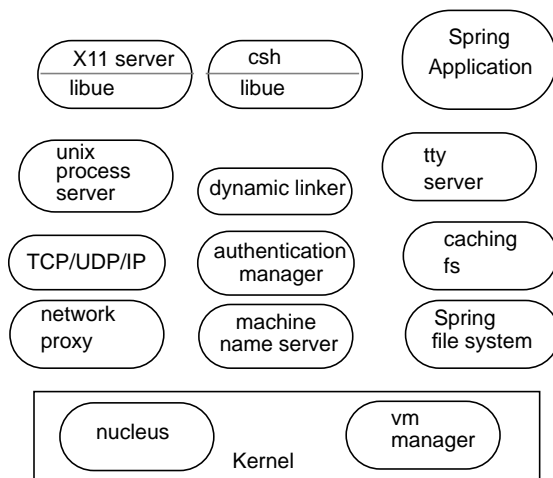


FIGURE 4. Major system components of a Spring node

All other system services, including naming, paging, network IO, filesystems, keyboard management, etc., are implemented as user-level servers. These servers provide object-oriented interfaces to the resources they manage and clients communicate with system servers by invoking these objects.

Spring is inherently distributed. All the services and objects available on one node are also available on other nodes in the same distributed system.

5 The nucleus

The nucleus is Spring's microkernel. It supports three basic abstractions: *domains*, *threads*, and *doors* [1].

Domains are analogous to processes in Unix or to tasks in Mach. They provide an address space for applications to run in and act as a container for various kinds of application resources such as threads and doors.

Threads execute within domains. Typically each Spring domain is multi-threaded, with separate threads performing different parts of an application.

Doors support object-oriented calls between domains. A door describes a particular entry point to a domain, represented by both a PC and a unique value nominated by the domain. This unique value is typically used by the object server to identify the state of the object; e.g., if the implementation is written in C++ it might be a pointer to a C++ object.

5.1 Doors

Doors are pieces of protected nucleus state. Each domain has a table of the doors to which it has access. A domain refers to doors using *door identifiers*, which are mapped through the domain's door table into actual doors. A given door may be referenced by several different door identifiers in several different domains.

Possession of a valid door gives the possessor the right to send an invocation request to the given door.

A valid door can only be obtained with the consent of the target domain or with the consent of someone who already has a door identifier for the same door.

As far as the target domain is concerned, all invocations on a given door are equivalent. It is only aware that the invoker has somehow acquired an appropriate door identifier. It does not know who the invoker is or which door identifier they have used.

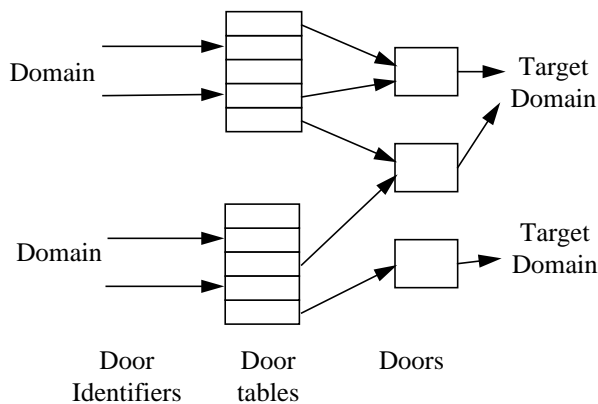


FIGURE 5. Domains, doors, and door tables

5.2 Object Invocation Via Doors

Using doors, Spring provides a highly efficient mechanism for cross-address-space object invocation. A thread in one address space can issue a door invocation for an object in another address space. The nucleus allocates a server thread in the target address space and quickly transfers control to that thread, passing it information associated with the door plus the argument data passed by the calling thread.

When the called thread wishes to return, the nucleus deactivates the calling thread and reactivates the caller thread, passing it any return data specified by the called thread.

For a call with minimal arguments, Spring can execute a low-level cross-address-space door call in 11 μ s on a SPARCstation 2, which is significantly faster than using more general purpose inter-process communication mechanisms [1].

Doors can be passed as arguments or results of calls. The nucleus will create appropriate door table entries for the given doors in the recipient's door table and give the recipient door identifiers for them.

6 Network Proxies

To provide object invocation across the network, the nucleus invocation mechanism is extended by *network proxies* that connect up the nuclei of different machines in a transparent way. These proxies are normal user-mode server domains and receive no special support from the nucleus. One Spring machine might include several proxy domains that speak different network protocols.

Proxies transparently forward door invocations between domains on different machines. In Figure 6, when a client on machine B invokes door Y, this door invocation goes to network proxy B; B forwards the call over the net to its buddy, proxy A; proxy A does a door invocation; and the door invocation then arrives in the server domain.

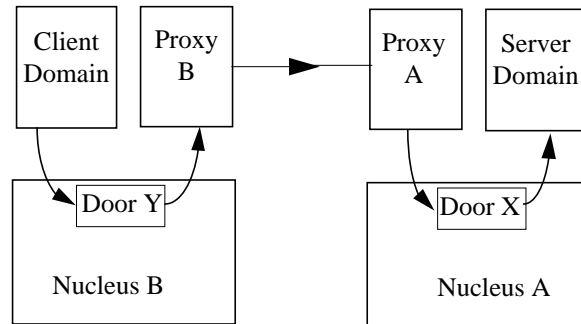


FIGURE 6. Using proxies to forward a call between machines

Notice that neither the client nor the server need be aware that the proxies exist. The client just performs a normal door invocation, the server just sees a normal incoming door invocation.

Door identifiers are mapped into *network handles* when they are transmitted over the network, and are mapped back into doors when they are received from the network.

A network handle contains a network address for the creating proxy, and a set of bits to identify a particular door that is exported by this proxy. In theory the set of bits is large enough to make it hard for a malicious user to guess the value of a network handle, thereby providing protection against users forging network handles.

7 Spring's security model

One of Spring's goals is to provide secure access to objects, so that object implementations can control access to particular data or services. To provide security we support two basic mechanisms, Access Control Lists and software capabilities.

Any object can support an Access Control List (ACL) that defines which users or groups of users are allowed access to that object. These Access Control Lists can be checked at runtime to determine whether a given client is really allowed to access a given object.

When a given client proves that it is allowed to access a given object, the object's server creates an *object reference* that acts as a software capability. This object reference uses a nucleus door as part of its representation so that it

cannot be forged by a malicious user. This door points to a *front object* inside the server. A front object is *not* a Spring object, but rather whatever the server's language of implementation defines an object to be.

A front object encapsulates information identifying the principal (e.g., a user) to which the software capability was issued and the access rights granted to that principal.

A given server may create many different front objects, encapsulating different access rights, all pointing to the same piece of underlying state. Later, when the client issues an object invocation on the object reference, the invocation request is transmitted securely through the nucleus door and delivered to the front object. The front object then checks that the request is permissible based on the encapsulated access rights, and if so, forwards the request into the server. For example, if the client issued an update request, the front object would check that the encapsulated access included write access.

When a client is given an object reference that is acting as a capability they can pass that object reference on to other clients. These other clients can then use the object reference freely and will receive all the access that was granted to the original client.

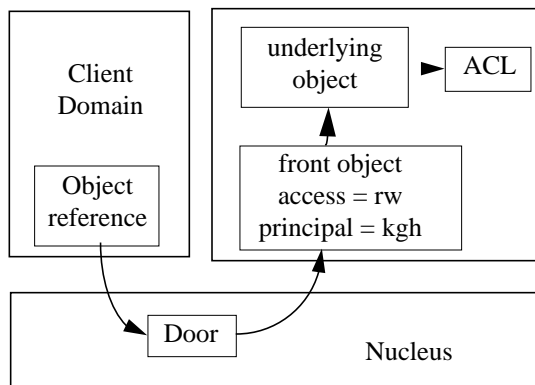


FIGURE 7. A client accessing a secure object

For example, say that user X has a file object foo, which has a restricted access control list specifying that only X is allowed to read the file. However X would like to print the file on a printserver P. P is not on the ACL for foo, so it would not normally have access to foo's data. However, X can obtain an object reference that will act as a software capability, encapsulating the read access that X is allowed to foo. X can then pass that object reference on to the printserver P and P will be able to read the file.

The use of software capabilities in Spring makes it easy for application programs to pass objects to servers in a way that allows the server to actually use the given object.

8 Virtual Memory

Spring implements an extensible, demand-paged virtual memory system that separates the functionality of caching pages from the tasks of storing and retrieving pages [7].

8.1 Overview

A per-machine virtual memory manager (VMM) handles mapping, sharing, protecting, transferring, and caching of local memory. The VMM depends on *external* *pag*ers for accessing backing store and maintaining inter-machine coherency.

Most clients of the virtual memory system only deal with *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain while a memory object is an abstraction of memory that can be mapped into address spaces. An example of a memory object is a file object (the file interface in Spring inherits from the memory object interface). Address space objects are implemented by the VMM.

A memory object has operations to set and query the length, and an operation to *bind* to the object (see Section 8.2). There are no page-in/out or read/write operations on memory objects. The Spring file interface provides file read/write operations (but not page-in/page-out operations). Separating the memory abstraction from the interface that provides the paging operations is a feature of the Spring virtual memory system that we found very useful in implementing our file system [13]. This separation enables the memory object server to be in a different

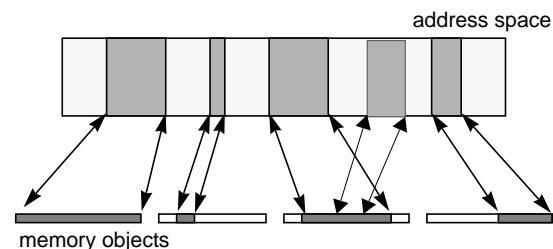


FIGURE 8. User's view of address spaces

An address space is a linear range of addresses with regions mapped to memory objects. Each region is mapped to a (part of) a memory object. Each page within a mapped region may be mapped with read/write/execute permissions and may be locked in memory.

machine than the *pager* object server which provides the contents of the memory object.

8.2 Cache and Pager Objects

In order to allow data to be coherently cached by more than one VMM, there needs to be a two-way connection between the VMM and an external pager (e.g., a file server). The VMM needs a connection to the external pager to allow the VMM to obtain and write out data, and the external pager needs a connection to the VMM to allow the provider to perform coherency actions (e.g., to invalidate data cached by the VMM). We represent this two-way connection as two objects.

The VMM obtains data by invoking a *pager* object implemented by an external pager, and an external pager performs coherency actions by invoking a *cache* object implemented by a VMM.

When a VMM is asked to map a memory object into an address space, the VMM must be able to obtain a pager object to allow it to manipulate the object's data. Associated with this pager object must be a cache object that the external pager can use for coherency.

A VMM wants to ensure that two equivalent memory objects (e.g., two memory objects that refer to the same file on disk), when mapped, will share the data cached by the VMM. To do this, the VMM invokes a *bind* operation on the memory object. The bind operation returns a *cache_rights* object, which is always implemented by the VMM itself. If two equivalent memory objects are mapped, then the same *cache_rights* object will be returned. The VMM uses the returned object to find a pager-cache object connection to use, and to find any pages cached for the memory object.

When a memory object receives a bind operation from a VMM, it must determine if there is already a pager-cache object connection for the memory object at the given VMM. If there is no connection, the external pager implementing the memory object contacts the VMM, and the VMM and the external pager exchange pager, cache, and *cache_rights* objects. Once the connection is set up, the memory object returns the appropriate *cache_rights* object to the VMM.

Typically, there are many pager-cache object channels between a given pager and a VMM (see Figure 9 for an example).

8.3 Maintaining Data Coherency

The task of maintaining data coherency between different VMMs that are caching a memory object is the responsi-

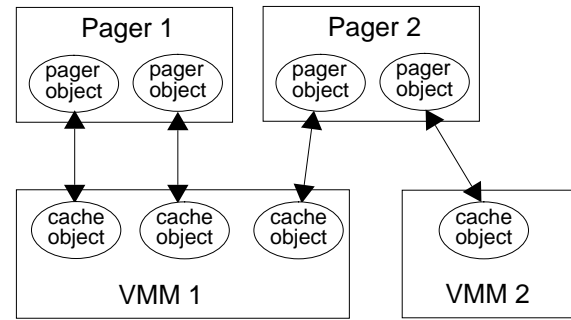


FIGURE 9. Pager-cache object example

A VMM and an external pager have a two-way pager-cache object connection. In this example, Pager 1 is the pager for two distinct memory objects cached by VMM 1, so there are two pager-cache object connections, one for each memory object. Pager 2 is the pager for a single memory object cached at both VMM 1 and VMM 2, so there is a pager-cache object connection between Pager 2 and each of the VMMs.

bility of the external pager implementing the memory object. The coherency protocol is not specified by the architecture—external pagers are free to implement whatever coherency protocol they wish. The cache and pager object interfaces provide basic building blocks for constructing the coherency protocol. Our current external pager implementations use a single-writer/multiple-reader per-block coherency protocol [12, 13].

9 File System

The file system architecture defines *file* objects that are implemented by file servers. The file object interface inherits from the *memory object* and *io* interfaces. Therefore, file objects may be memory mapped (because they are also memory objects), and they can also be accessed using the read/write operations of the *io* interface.

Spring includes file systems giving access to files on local disks as well as over the network. Each file system uses the Spring security and naming architectures to provide access control and directory services.

A Spring file system typically consists of several layered file servers [5]. The pager-cache object paradigm is used by file systems as a general layering mechanism between the different file servers and virtual memory managers. Among other things, this has enabled us to provide per-machine caching of data and attributes to decrease the number of network accesses for remote files.

9.1 File Server Implementations

The Spring Storage File System (SFS) is implemented using two layers as shown in Figure 10.

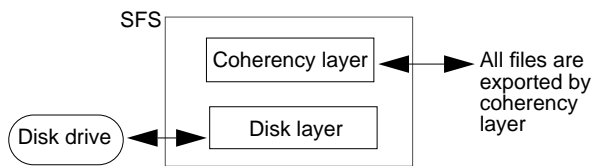


FIGURE 10. Spring SFS

The base disk layer implements an on-disk Unix compatible file system. It does not, however, implement a coherency algorithm. Instead, an instance of the coherency file server is stacked on the disk layer, and all files are exported to clients via the coherency layer.

The coherency layer implements a per-block multiple-reader/single-writer coherency protocol. Among other things, the implementation keeps track of the state of each file block (read-only vs. read-write) and of each cache object that holds the block at any point in time. Coherency actions are triggered depending on the state and the current request using a single-writer/multiple-reader per-block coherency algorithm. The coherency layer also caches file attributes.

The Caching File System (CFS) is an attribute-caching file system. Its main function is to interpose itself between remote files and local clients when they are passed to the local machine so as to increase the efficiency of many operations. Once interposed on, all calls to remote files end up being diverted to the local CFS.

An interesting aspect of CFS is the manner in which it *dynamically* interposes on individual remote DFS files. A *caching* subcontract is used to contact the local CFS in the process of unmarshalling file objects. When CFS is asked to interpose on a file, it becomes a cache manager for the remote file by invoking the bind operation on the file as described in Section 8.2.

10 Spring Naming

An operating system has various kinds of objects that need to be given names, such as users, files, printers, machines, services, etc. Most operating systems provide several name services, each tailored for a specific kind of object. Such *type specific* name services are usually built into the subsystem implementing those objects. For example, file systems typically implement their own naming service for naming files (directories).

In contrast, Spring provides a uniform name service [17]. In principle, any object can be bound to any name. This applies whether the object is local to a process, local to a machine, or resident elsewhere on the network, whether it is transient or persistent; whether it is a standard system object, a process environment object, or a user specific object. Name services and name spaces do not need to be segregated by object type. Different name spaces can be composed to create new name spaces.

By using a common name service, we avoid burdening clients with the requirement to use different names or different name services depending on what objects are being accessed. Similarly, we avoid burdening all object implementations with constructing name spaces—the name service provides critical support to integrate new kinds of objects and new implementations of existing objects into Spring. Object implementations maintain control over the representation and storage of their objects, who is allowed access to them, and other crucial details. Although Spring has a common name service and naming interface, the architecture allows different name servers with different implementation properties to be used as part of the name service.

The name service allows an object to be associated with a name in a *context*, an object that contains a set of name-to-object associations, or *name bindings*, and which is used by clients to perform all naming operations. An object may be bound to several different names in possibly several different contexts at the same time. Indeed, an object need not be bound to a name at all.

By binding contexts in other contexts we can create a *naming graph* (informally called a name space), a directed graph with nodes and labeled edges, where the nodes with outgoing edges are contexts.

Unlike naming in traditional systems, Spring contexts and name spaces are first class objects: they can be accessed and manipulated directly. For example, two applications can exchange and share a private name space. Traditionally, such applications would have had to build their own naming facility, or incorporate the private name space into a larger system-wide name space, and access it indirectly via the root or working context.

Since Spring objects are not persistent by default, naming is used to provide persistence [16]. It is expected that applications generally will (re)acquire objects from the name service. If the part of the name space in which the object is found is persistent, then the object will have been made persistent also.

A Spring name server managing a persistent part of a name space converts objects to and from their persistent

form (much like the UNIX file system, which converts open files to and from their persistent form). However, since naming is a generic service for an open-ended collection of object types, a context cannot be expected to know how to make each object type persistent. Spring object managers have ultimate control of the (hidden) states of their objects. Therefore we provide a general interface between object managers and the name service that allows persistence to be integrated into the name service while allowing the implementation to control how its (hidden) objects' states are mapped to and from a persistent representation.

Because the name service is the most common mechanism for acquiring objects, it is a natural place for access control and authentication. Since the name service must provide these functions to protect the name space, it is reasonable to use the same mechanism to protect named objects. The naming architecture allows object managers to determine how much to trust a particular name server, and an object manager is permitted to forego the convenience and implement its own access control and authentication if it wishes. Similarly, name servers can choose to trust or not to trust other name servers.

The Spring name service does not prescribe particular naming policies; different policies can be built on the top. Our current policy is to provide a combination of system-supplied shared name spaces, per-user name spaces, and per-domain name spaces that can be customized by attaching name spaces from different parts of the distributed environment.

By default, at start-up each domain is passed from its parent a private domain name space, which incorporates the user and system name spaces. A domain can acquire other name spaces and contexts if it desires.

11 UNIX Emulation

Spring can run Solaris binaries using the UNIX emulation subsystem [6]. It is implemented entirely by user-level code, employs no actual UNIX code, and provides binary compatibility for a large set of Solaris programs. The subsystem uses services already provided by the underlying Spring system and only implements UNIX-specific features that have no counterpart in Spring (e.g., signals). No modifications to the base Spring system were necessary to implement Solaris emulation.

The implementation consists of two components: a shared library (*libue.so*) that is dynamically linked with each Solaris binary, and a set of UNIX-specific services

exported via Spring objects implemented by a *UNIX process server* (in a separate domain). See Figure 4.

The UNIX process server implements functions that are not part of the Spring base system and which cannot reside in *libue.so* due to security reasons.

11.1 Libue

When a program is *execed*, *libue.so* is dynamically linked with the application image in place of *libc*, thus enabling the application to run unchanged.

The *libue.so* library encapsulates some of the functionality that normally resides in a monolithic UNIX kernel. In particular, it delivers signals forwarded by the UNIX process server, and keeps track of the association between UNIX file descriptor numbers (fd's) and Spring file objects.

For each UNIX system call, we implemented a library stub. In general, there are three kinds of calls:

1. Calls that simply take as an argument an fd, parse any passed flags, and invoke a Spring service (e.g., *read*, *write*, and *mmap*). Most file system and virtual memory operations fall in this category.
2. Calls that eventually call a UNIX-specific service in the UNIX process server. Examples include *pipe* and *kill*.
3. Calls that change the local state without calling any other domain. *Dup*, parts of *fcntl*, and many signal handling calls fall into this category.

11.2 UNIX Process Server

The UNIX process server maintains the parent-child relationship among processes, keeps track of process and group ids, provides sockets and pipes, and forwards signals.

The UNIX process server is involved in forking and executing of new processes. It is also involved in forwarding (but not delivering signals). Since it keeps track of process and group ids, it enforces UNIX security semantics when servicing requests from client processes.

12 Conclusions

The Spring project chose to build a different operating system, one based on the notions of strong interfaces, openness and extensibility and designed to be distributed and suited to multiprocessors. Using object-oriented ideas and strong interfaces has been a natural fit, with a number of benefits:

- A standardized basis for open, distributed object systems via the Interface Definition Language and a simple client model for objects
- Easy distributed services and applications
- Readily extensible system facilities, such as file systems and name services
- Unity of architecture together with a wide range of implementation opportunities as in virtual memory management, naming, subcontract, and serverless objects
- Highly efficient inter-address space object invocation in support of a microkernel-based architecture.

Finally, designing in security mechanisms from the start has provided a system that can support a wide range of secure mechanisms in a networked environment, from the most relaxed to the most secure.

13 References

- [1] Graham Hamilton and Panos Kougiouris, "The Spring Nucleus: A Microkernel for Objects," Proc. 1993 Summer USENIX Conference, pp. 147-160, June 1993.
- [2] Graham Hamilton, Michael L. Powell, and James G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming," Proc. 14th ACM Symposium on Operating Systems Principles, pp. 69-79, December 1993.
- [3] Graham Hamilton and Sanjay Radia, "Using Interface Inheritance to Address Problems in System Software Evolution," Proc. ACM Workshop on Interface Definition Languages, January 1994.
- [4] Peter B. Kessler, "A Client-Side Stub Interpreter," Proc. ACM Workshop on Interface Definition Languages, January 1994.
- [5] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," Proc. 14th ACM Symposium on Operating Systems Principles, pp. 1-14, December 1993.
- [6] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," Proc. Winter 1993 USENIX Conference, pp. 469-479, January 1993.
- [7] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-93-9, March 1993.
- [8] Yousef A. Khalidi and Michael N. Nelson, "A Flexible External Paging Interface," Proc. 2nd Workshop on Microkernels and Other Kernel Architectures, September 1993.
- [9] Michael N. Nelson and Graham Hamilton, "High Performance Dynamic Linking Through Caching," Proc. 1993 Summer USENIX Conference, pp. 253-266, June 1993.
- [10] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi, "Caching in an Object-Oriented System," Proc. 3rd International Workshop on Object Orientation in Operating Systems (I-WOOS III), pp. 95-106, December 1993.
- [11] Michael N. Nelson and Yousef A. Khalidi, "Generic Support for Caching and Disconnected Operation," Proc. 4th Workshop on Workstation Operating Systems (WWOS-IV), pp. 61-65, October 1993.
- [12] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," Proc. 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), September 1993.
- [13] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "The Spring File System," Sun Microsystems Laboratories Technical Report SMLI-93-10, March 1993.
- [14] Michael N. Nelson and Sanjay R. Radia, "A Uniform Name Service for Spring's Unix Environment," Proc. Winter 1994 USENIX Conference, Jan. 1994.
- [15] Object Management Group, "Common Object Request Broker Architecture and Specification," OMG Document 91.12.1, December 1991.
- [16] Sanjay Radia, Peter Madany, and Michael L. Powell, "Persistence in the Spring System," Proc. 3rd International Workshop on Object Orientation in Operating Systems (I-WOOS III), pp. 12-23, December 1993.
- [17] Sanjay R. Radia, Michael N. Nelson, and Michael L. Powell, "The Spring Name Service," Sun Microsystems Laboratories Technical Report SMLI-93-16, October 1993.

Specifying and Testing Software Components Using ADL

Sriram Sankar and Roger Hayes

Introduction by Alberto Savoia

Mission and business critical software must operate as specified, but ensuring that that implementation meets the specification has always been a challenge. There are two primary ways to determine if a program works as intended. The first one is to study the code and "prove" that the implementation does what the specification says. Since it takes even people with PhDs in math several days to generate a proof of correctness for even a trivial program, this approach is used very rarely. The second approach is to test the implementation of the software by creating and running a set of tests to exercise it. This approach is more practical, but most tests are developed manually and in a very ad-hoc manner, making it very difficult to determine if the set of tests is sufficiently thorough to provide the required level of confidence. The ADL project focused on bridging the huge gap between formal proofs of correctness and ad-hoc testing; its goal was to leverage the best of both worlds by borrowing formal specifications from the proof of correctness camp and executable tests from the traditional testing camp.

The key component, and one of the major innovations of the ADL project, was the ADL language (Assertion Definition Language). Unlike other formal specification languages, ADL was designed to be easily used and understood by average programmers (as opposed to mathematicians). ADL's syntax, for example, looks a lot like Java™ code, which minimizes the learning curve for most users. The other key component of the project was ADLT (ADL Translator). ADLT takes ADL specifications and generates self-checking tests that are used to verify the implementation.

ADL technology was developed concurrently with, and applied to, the Spring project. This was a particularly powerful combination, since a core component of Spring was the use of "contracts" to specify interface syntax. ADL was used to augment this syntax specification with a semantics section, which not only enabled automated test generation for Spring methods, but also increased the chances that the implementation would reflect the original intent of the methods.

When the Java™ programming language started to make waves, the ADL group thought that it would be a great help to Java's cause if early adopters had specialized testing tools for this new language. We also realized that a lot of the theory and technology behind ADL and ADLT could be applied to Java testing. Armed with some good arguments and lots of chutzpah, we enlisted the help of Bert Sutherland, Jim Mitchell, and Eric Schmidt, and got Scott McNealy's blessing for a new technology transfer model: starting a new business unit to develop and market technology developed by the lab. The new business unit, called SunTest™, grew to almost 50 people in over two years, and in addition to helping Java adoption, it generated almost six millions dollars in revenue in its first year of sales, after which it was absorbed into SunSoft™.

In addition to SunTest™, the results of the ADL projects have been used both at Sun and by standards organizations such as X/Open® (which was also involved in the project).

One of the most interesting aspects of the ADL project is that most of its funding (almost \$4M) came in the form of a research grant from Japan's MITI. Like many organizations that develop or utilize mission critical software, MITI was extremely concerned that the current state-of-the-art for software specification and testing was not adequate to provide the level of confidence that

they required. So, in order to advance the state-of-the-art, they contacted the top researchers in the field of software verification and testing, dangled a \$2M carrot in front of them, and asked them for proposals. Sun's proposal by the PrimaVera group beat all the others, including one from one of Sun's foes and competitors at the time, DEC. The first phase of the project was so successful that the project was extended and funded another \$2M.

Specifying and Testing Software Components using ADL

Sriram Sankar
Roger Hayes

SMLI TR-94-23

April 1994

Abstract:

This paper presents a novel approach to unit testing of software components. This approach uses the specification language ADL, that is particularly well-suited for testing, to formally document the intended behavior of software components. Another related language, TDD, is used to systematically describe the test-data on which the software components will be tested.

This paper gives a detailed overview of the ADL language, and a brief presentation of the TDD language. Some details of the actual test system are also presented, along with some significant results.



M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:

sriram.sankar@eng.sun.com
roger.hayes@eng.sun.com

Specifying and Testing Software Components using ADL

Sriram Sankar Roger Hayes

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, California 94043

1 Introduction

At the start of the programming task, the programmer is supplied with a *specification* of the problem. The specification can range from being quite detailed, such as a document that describes the intended behavior of the program in all possible circumstances, to being quite informal, such as a prose description of what the program is supposed to do in a few situations.

In practice, the programmer has several sources of information available that comprise the specification. These may include a formal specification document, a working prototype, instances of program behavior, and *a priori* knowledge about similar software.

Working from this specification, the programmer develops the program. The *validation* of the program lies in the comparison of the program with the specification of intended behavior.

To automate the validation process, it is essential to formalize the specification of the program. For this purpose (among others), various specification languages and specification techniques have been designed—an overview of which can be found in [10]. Some examples of specification languages are Anna [11], Z [15], Larch [7], and VDM [2].

Our approach to program validation is through *testing*. In this approach, the program is run with many different test inputs in a systematic manner. Correct behavior is determined by examining the results of the program or function in terms of the specification that describes its behavior. Correct execution of the program on these test inputs increases the level of confidence in the program.

There are two parts to software testing. First, there is the problem of *test-data selection*. Since any test-data will necessarily be a very small sample of all the possible inputs, test-data should be

selected in such a way that successful execution of a program on these test-data gives us a reasonable amount of confidence in the correctness of the program for all possible inputs. At the same time, the test-data should be such that redundancy in the testing process is minimized.

For this purpose, we have designed a high-level *test-data description* (TDD) language in which test-data for a program is described in a systematic manner.

Second, the program needs to be run on each set of test-data to determine if the program behaves as intended for this set of test-data. In most published work on testing, the details of this determination have not been dealt with. Rather, an *oracle* is assumed to exist. This oracle can judge for any specific set of test-data, whether or not the program behaves as intended. The idealization of the oracle has been essential for software testing. Lately, however, much work has been done to realize this oracle. In nearly all cases, this has been achieved by comparing the behavior of a program against its specifications that have been written in some formal specification language (e.g., see [5, 3, 14]).

We have designed a specification language called ADL (Assertion Definition Language). The design of ADL is well-suited for testing—generating a test oracle from an ADL specification is a straightforward process.

Pilot implementations for an ADL-based test generation system have been undertaken, and we have had significant results from these experiments. For example, we discovered an anomaly in an implementation of the `write` system call in the UNIX[®] operating system. The `write` function is specified to update the last modification time of a file. On a particular version of the UNIX operating system, performing a write with a 0 byte data value changed the modification time on local files, but did not in the case of remote files. Since a write of 0 bytes does not change the file, either behavior is defensible, but the English specification is not clear on the point. The anomaly had not been detected earlier, even though standard rigorous testing schemes had been applied on this system call. The reason we were able to detect this was due to the systematic nature of developing the specifications and test-data descriptions.

Overview of this paper

This paper presents an overview of our testing methodology. The ADL and TDD languages are described, although more attention is paid to the ADL language. Other aspects (other than ADL) are covered in brief in this paper, but will be covered in detail in future papers.

The remainder of the introduction lists the high-level features of ADL and TDD, and then provides some background information on test-data selection and on earlier work of the PrimaVera group at Sun Microsystems Laboratories, Inc.

Section 2 describes the ADL language. Section 3 describes how *assertion-checking functions* (our oracle) are generated from ADL specifications, and provides a description of the TDD language. Finally, Section 4 concludes the paper with a discussion of ongoing work.

More information on the ADL testing methodology and environment is available in [16] and [17].

1.1 ADL

ADL is a language designed for formal specification of software components. It is well-suited for the purpose of testing. ADL defines a set of general-purpose specification concepts applicable for the specification of software written in most programming languages. Some of the key features of ADL are listed below:

- ADL is a language framework that provides a set of high-level specification concepts. These concepts may be specialized for use with a programming language by rendering them into a syntax similar to that of the programming language.
- All ADL specifications are post-conditions on operations (or functions) of software components. Therefore, ADL specifications are constraints on the program state at the time of termination of operation evaluations.
- ADL specifications are written as separate units—i.e., they are not embedded in the program. The ADL specification writer defines a binding between the specifications and the functions in the program to provide the necessary association. This makes them suitable for specifying extant program libraries.
- ADL specifications may be partial. That is, the complete details of the function do not have to be written in ADL. Typically, ADL specifications are augmented with informal natural language documentation.
- ADL provides specialized constructs for the specification of errors. Most specifications written as natural language documents (such as UNIX man pages) describe error situations separately. ADL's error specification constructs allow a formal specification to be organized in a similar manner.
- ADL constructs are designed to allow translation of formal specifications into natural language documents. ADL's constructs are at a high-level of abstraction and permit a specification writer to write specifications very similar to the way they would do it in a natural language; hence the translation process is straightforward.
- ADL is well-suited for the purpose of testing software components. Difficult to evaluate constructs such as quantifiers have been excluded from the language for the time being.

1.2 TDD

The TDD language offers the test designer a formal and structured framework for describing test-data. TDD provides a structure for characterizing and documenting the data used in testing. Through the use of TDD's syntax, test-data becomes the subject of a design process. The important features of TDD are listed below:

- Data is characterized in an abstract and systematic way. By using a formal system for notating the description of test-data, we focus on the test designer's intention rather than on the details of generating a particular instance of test-data.

- Test-data is generated without prejudice. By isolating the description of test-data from its realization, we explore what might otherwise have been blind spots. TDD encodes the designer's insight into formal descriptions, which are decompositions of the properties of the data. These descriptions are recombined into test cases. This ensures that all combinations of properties are tested; without the decomposition/recomposition process, it is very easy to omit an important test because it does not occur in an imagined scenario of use.
- Input is characterized independently of any particular implementation. TDD describes the input data from the point of view of a user of the tested software component. A TDD description may be constructed with insight into implementations, but its correctness does not depend on a particular implementation. This means that a TDD test suite is portable across implementations.
- Iteration over the test cases is systematic and thorough. The regularity of the process allows for better statistics and also helps reduce the incidence of errors missed due to oversight.
- Data manufacture is isolated. The messy task of generating actual test values is encapsulated in well-defined functions. These functions, that translate symbolic descriptions into actual values, can be used in manually written tests as well as in ADL-generated tests.

1.3 Background on Test-Data Generation

A large number of tools have been designed for test-data generation since the early 1970's. The emphasis has been to generate test-data that exercises as much of the program code as is practically possible. Some approaches have been to generate test-data that force every program statement to be executed (statement coverage), while others force every edge in the program's flowchart to be traversed (path coverage). A useful technique for test-data generation is symbolic execution of the program [3, 9]. Symbolic execution can be performed in a forward traversal or a backward traversal of the program paths. During these traversals, various constraints are established which are then used to generate the test-data. This falls under the general category of *white-box testing*. In white-box testing, the structure of the program is examined and test-data are derived from the program's logic. The other category is *black-box testing*. This is also known as *functional testing*. In this case, the internal structure and behavior of the program is not considered. The objective is solely to find out when the input-output behavior of the program does not agree with its specification. In this approach, test-data are constructed from the specification [8, 12].

Weyuker and Ostrand [18], building on the work of Goodenough and Gerhart [6], attempt to define a theoretically sound and practical definition of what constitutes an adequate test. The idea is to divide the test-data into a finite number of equivalence classes where testing on a representative of an equivalence class will, by induction, test the entire class. The equivalence classes are derived from both the program specification (called a problem partition of the input in [18]) and an examination of the program structure (called a path domain partition). TDD is very nearly an implementation of the concept of revealing subdomains from this work.

1.4 The PrimaVera Project

The PrimaVera group at Sun Microsystems Laboratories, Inc. has been working on applying formal specification techniques to software testing for eight years. Our emphasis is on good engineering solutions, minimizing the cost of adoption and training. We strive for systems that combine the benefits of formal methods with a low entry cost, and systems that allow incremental adoption with an early payback.

After several years of internal development and deployment, it was decided to make our work externally available. The PrimaVera technology was submitted in response to a request for proposals for automated testing technology issued by X/Open Company Limited, and was selected for a joint research project sponsored by a research grant from Information-Technology Promotion Agency, Japan (IPA), a governmental organization under Ministry of International Trade and Industry (MITI).

2 The ADL Language

ADL is a language framework designed for the formal specification and testing of software components. ADL defines a set of general-purpose specification concepts applicable for the specification of software written in most programming languages. ADL excludes specification concepts that, although useful, are difficult to implement using state-of-the-art technology.

The concepts of the ADL language framework may be specialized for use with a programming language by rendering these concepts into a syntax similar to that of the particular programming language. This syntax may then be augmented with constructs from the programming language, such as its expression syntax. This approach of defining a language framework that may be specialized for use with any language has been used successfully in other projects, e.g., IDL [4], Rapide [1], and Larch [7].

The ADL language framework has been specialized for use with the C programming language, and with IDL—Object Management Group’s interface definition language for the CORBA architecture specification. We intend to specialize ADL for use with C++ and Ada[®] shortly. For the purposes of this paper, ADL will be described through its specialization for the C programming language.

Other important aspects of ADL are described below.

- *Post-condition specifications*
All ADL specifications are post-conditions on operations (or functions) of software components. Therefore, an ADL specification is a constraint on the program state at the time of termination of operation evaluation. This constraint may be contingent on pre-operation program state by use of the *call-state* operator.

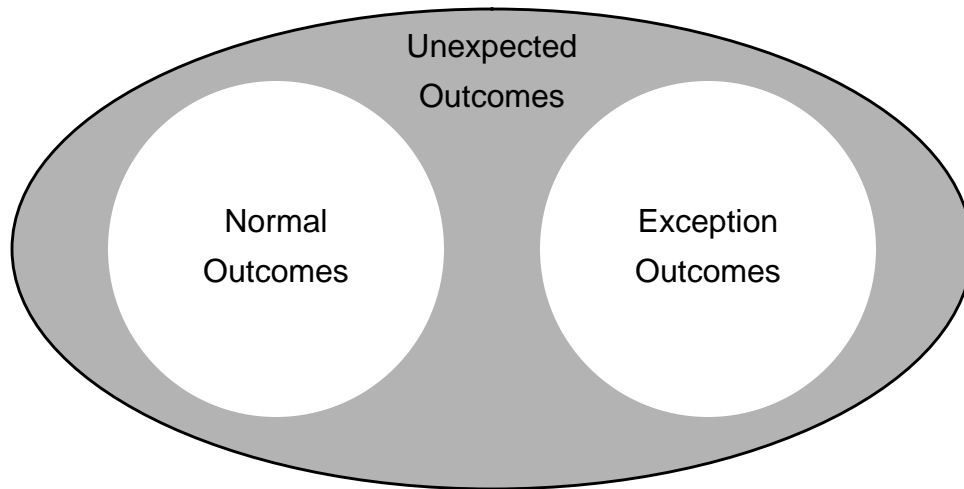


Figure 1. The Possible Outcomes of an Operation Evaluation

This is an example of ADL’s client orientation. An ADL specification does not give conditions under which the function must be called, but instead tells what will happen if it is called.

- *Non-intrusive*

ADL specifications are written as separate units—i.e., they are not embedded in the program (e.g., as in Anna). The ADL specification writer defines a binding between the specifications and the functions in the program to provide the necessary association. This binding provides sufficient information for the ADL testing tools to generate frameworks to test these functions.

This approach is non-intrusive to the extent that the functions being specified need not be recompiled for testing purposes. Hence, the ADL testing technology may be applied to precompiled code, including code such as operating systems, that by their very nature cannot be recompiled and reloaded in a straightforward manner.

- *Constructs for specification of errors*

Figure 1 illustrates the possible outcomes of an operation evaluation. The outcomes can be divided into two categories—*expected* and *unexpected*. Expected outcomes (the white portion of Figure 1) are those that are included in the documented behavior of the function, while unexpected outcomes (the grey portion of Figure 1) are outcomes that are not supposed to happen (e.g., $(2+2)$ evaluating to 5).

Our research has shown that it is convenient to divide the expected outcomes of an operation into *normal* and *exception* (or abnormal) outcomes. This division is usually subjective, but some general guidelines may be laid down. For example, the outcome of (2+2) evaluating to 4 is usually considered a normal outcome, while the outcome of (128+128) being an overflow is usually considered an exception outcome.

ADL provides constructs to separate the handling of normal and exception outcomes of an operation, and to specify against unexpected outcomes.

- *Simple natural language mapping*
One of the mandates of the ADL project has been to develop a capability to transform a specification written in ADL into an equivalent natural language representation. This problem is, in general, untractable. However, ADL's constructs are at a high-level of abstraction and permit a specification writer to write specifications very similar to the way they would in a natural language. The error specification mechanisms discussed earlier are an example of these high-level constructs. The task of translating ADL specifications into equivalent natural language documents (e.g., UNIX man pages) becomes quite simple if the specification writer adheres to these high-level constructs while writing ADL specifications.
- *Enables testing*
The fact that the ADL design emphasizes applicability to testing of software components has already been mentioned. We reiterate this in the context of the other features such as the specification being non-intrusive, and being from a client's point of view. Furthermore, specification constructs such as quantifiers and algebraic specifications have been omitted from the current version of ADL. We have plans to explore the introduction of these constructs in the future.

2.1 ADL Constructs

The constructs provided by the ADL framework are described in this section. Some of these constructs are illustrated through examples in Section 2.2.

An ADL specification is made up of a set of *modules*. Each module encapsulates a set of *constituents* that describe the entities in the C program that are being specified. Modules may also refer to each other's constituents by importing constituents from one module to another.

A constituent of a module may be one of the following:

- *Type constituent*
A type constituent defines a type and gives it a name. It's syntax is identical to the C `typedef` statement.

- *Object constituent*
An object constituent introduces an object¹ and associates it with a type. It's syntax is similar to that of a C object declaration. Objects introduced by object definitions are bound to C objects with the same type.
- *Function constituent*
A function constituent introduces a function and specifies its parameter and result types. It's syntax is similar to that of a C function declaration. Functions introduced by function definitions are bound to C functions that have the same parameter and result types.

Function constituents may contain *semantic descriptions*. A semantic description describes the behavior of the C function that is mapped to it's function constituent. The semantic description constrains the program state at the end of calls to this C function. A semantic description has two components:

- *Bindings*
Bindings are associations between expressions and names. These names may be used subsequently as a short form for their associated expressions.
- *Assertions*
An assertion is a Boolean expression that must be true whenever control returns from the function constrained by the semantic description.

It is often useful to make use of existing specification concepts while writing assertions. Sometimes these concepts may already exist as part of an module. However, there will be situations where these concepts are missing. In such situations, missing, but necessary, specification concepts can be declared as *auxiliary definitions*. Auxiliary definitions are simply ADL declarations that are visible only within bindings and assertions.

Predefined ADL operators and functions

Some of ADL's primitives for use in assertions are described below:

- *Call-state operator*
The call-state operator (“@”) takes one argument and evaluates it at the time of call to the function being specified.
- *normal and exception*
`normal` and `exception` are predefined names that may be bound to boolean expressions that characterize the normal and exception outcomes respectively (see Figure 1).

1. The word “object” is used here in the same sense as in C.

- *Implication operators*
ADL provides the standard logical implication and equivalence operators. It also provides an *exception operator* ($< : >$) that characterizes error situations by listing the conditions that cause the function to fail, and relates them to the error conditions that take place. This operator is used to characterize the exception outcomes of the function being specified.
- *normally*
This is a function that characterizes the normal outcomes of the function being specified. It takes a list of boolean parameters that must all be true on any normal outcome.

2.2 ADL Examples

This section provides two examples of an ADL specification of a bank module. The first example defines three operations, `balance`, `deposit`, and `withdraw`, within an module. These functions map to C functions with similar names and signatures. The specifications written in ADL describe the intended behavior of these C functions. The ADL specification is shown in Figure 2.

The bank specification of Figure 2 contains 7 constituents:

1. `errno`: This is an object constituent. It is defined to be of type `int`. This maps to the standard C global variable `errno`.
2. `NEG_AMT`: This is also an object constituent. It describes a particular value that `errno` can take. This maps to a C integer constant.
3. `INS_FUND`: Just as `NEG_AMT`, this describes another value that `errno` can take and maps to a C integer constant.
4. `acct_no`: This is a type constituent. This defines `acct_no` as another name for `int`.
5. `balance`: This is a function constituent. It describes a function that takes a parameter `acct` of type `acct_no` and returns a value of type `int`. The parameter is qualified to be of mode `in`, which indicates that only the input value of the parameter is relevant. `balance` maps to a C function with the same name, and same parameter and return types.
6. `deposit`: This is another function constituent. It takes two parameters and has a semantic description associated with it. Just as in the case of `balance`, this too maps to a C function with the same name, and same parameter and return types.
7. `withdraw`: This is another function constituent with a more detailed semantic description.

Semantic descriptions

The semantic description of `deposit` contains two assertions. The first assertion contains the call-state operator “@”. The call-state operator evaluates its argument (in this case `balance(acct)`) in the state at the time the function is called. This assertion states that the

```

module bank {

    int errno;
    int NEG_AMT, INS_FUND;

    typedef int acct_no;

    int balance(in acct_no acct);

    int deposit(in acct_no acct, in int amt)
        semantics {
            balance(acct) == @balance(acct) + amt,
            return == balance(acct)
        };

    int withdraw(in acct_no acct, in int amt)
        semantics {
            exception := (return == -1),
            normal := !exception,
            negative_amount := (errno == NEG_AMT),
            insufficient_funds := (errno == INS_FUND),
            @(amt < 0) <:> negative_amount,
            @(amt > balance(acct)) <:> insufficient_funds,
            exception --> unchanged(balance(acct)),
            normally (
                balance(acct) == @balance(acct) - amt,
                return == balance(acct)
            )
        };
};

```

Figure 2. The Bank Module in ADL

balance of account `acct` (i.e., `balance(acct)`) after the call to `deposit` is equal to the sum of the balance before the call and the amount deposited (i.e., `amt`).

The second assertion about `deposit` contains the reserved word `return`. This is used to refer to the value returned by the function (in this case `deposit`). This assertion states that the value returned by `deposit` is the new balance of account `acct`.

The semantic description of `withdraw` contains four bindings (the first four lines) and then a list of assertions. The bindings bind expressions to names. Use of these names in subsequent expressions then refer to the bound expressions.

The first two bindings bind expressions to the special names `exception` and `normal`. The bindings to these names, together with their use in specifications, characterize the normal and exception outcomes of `withdraw`. `exception` is bound to the expression `(return == -1)`, while `normal` is bound to the expression `!exception`, (i.e., `!(return == -1)`). In addition to providing a binding for `exception` and `normal`, these bindings also define the meanings of the exception operator `<:>` and the function `normally`. These are described in the following paragraphs.

The next two bindings provide short forms for the expressions `(errno == NEG_AMT)` and `(errno == INS_FUND)`.

The first assertion about `withdraw` contains the exception operator `<:>`. The exception operator characterizes error situations by listing the conditions that cause the function to fail and relating them to the error conditions that result. More specifically, the exception operator states that if its left operand is true, then the function will fail (i.e., `exception` will be true), and if the function fails and the right operand is true, then the left operand must be true. The intent is that the left operand defines the only program state that can cause the particular exception defined by the right operand, without prohibiting another independent exception.

This particular assertion states that if `amt` is less than 0 when `withdraw` is called, the function will fail (the function will return the value -1). Also, if the function fails and `negative_amount` is true (`errno == NEG_AMT`), then `amt` had to be less than 0 when `withdraw` was called.

Similarly, the second assertion about `withdraw` states that if `amt` is greater than the balance of account `acct` when `withdraw` is called, then the function will fail. Also, if the function fails and `insufficient_funds` is true, then `amt` has to be greater than the balance of account `acct` when `withdraw` was called.

The third assertion about `withdraw` contains a predefined function called `unchanged`. This function returns true if its argument has the same value after the call as before the call. This assertion therefore states that if `withdraw` fails, the balance of account `acct` will not change.

The fourth assertion about `withdraw` uses the predefined function `normally`. `normally` takes an arbitrary number of boolean parameters and returns true if all its parameters are true whenever `normal` is true. Therefore, on a normal return from the function being specified, all the parameters of `normally` must be true.

This particular assertion states that on a normal return from `withdraw` (i.e., the function does not return -1), the balance in account `acct` is decremented by `amt`, and the function returns the new account balance.

The second example illustrates the use of auxiliary definitions. Suppose the bank module did not define the function `balance`. Then it would not be possible to write assertions for `deposit` and `withdraw` in the style of the previous example, since these assertions make use of the notion of `balance`. In this case, `balance` may be introduced as an auxiliary definition. Figure 3 shows the bank module with the function `balance` introduced as an auxiliary definition. The auxiliary definition must be bound to a *C test function* with the same parameter and result types as `balance` for testing to be possible.

```

module bank {

    . . .

    auxiliary {
        int balance(in acct_no acct);
    }

    int deposit(in acct_no acct, in int amt)
        . . .;

    int withdraw(in acct_no acct, in int amt)
        . . .;
};

```

Figure 3. Auxiliary Definitions in ADL

3 Software Testing using ADL

ADL may be used in the testing of software in a variety of ways. In this section, we describe how ADL is used for the unit testing of C functions. A brief summary of other ways of testing being considered is presented in Section 4.

The following components are required for unit testing:

1. A function to be tested.
2. Test-data on which to execute the function. In this case, test-data is specified through the TDD file. This is described in Section 3.2.
3. A means of determining whether or not the function executed correctly. In this case, *assertion-checking functions* generated from the ADL specifications handle this task. This is described in Section 3.1.

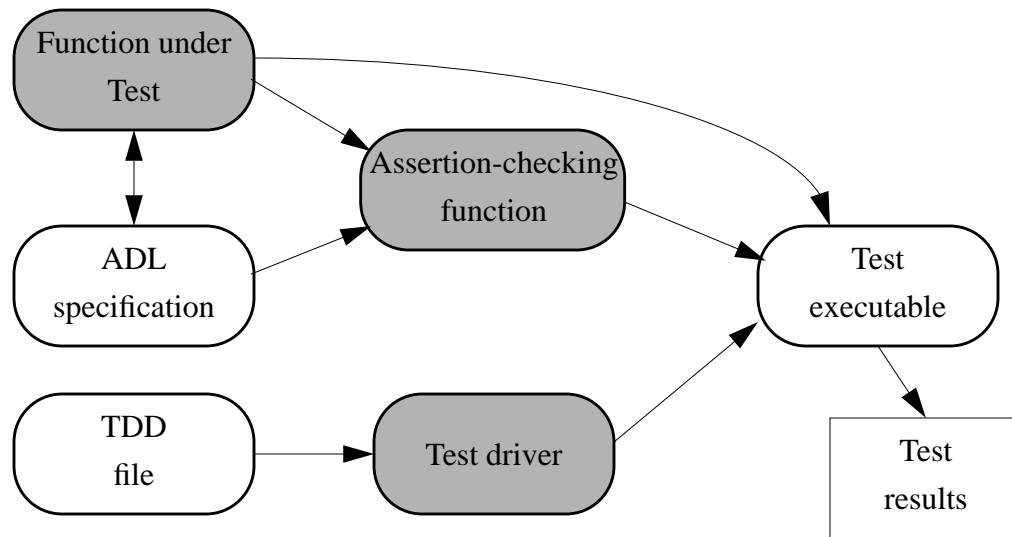


Figure 4. The Steps Involved in Unit Testing

Figure 4 illustrates the steps required to perform unit testing. The three components required for unit testing are shown on the left column. The double-edged arrow between the function under test and the ADL specification illustrates the binding between these two entities. From these two entities is constructed the assertion-checking function. Also, from the third component, the TDD file, is constructed the test driver that calls the function under test with different sets of test-data. The three shaded boxes are then linked and loaded together to obtain the test executable. When the text executable is run, it produces the test results that describe how the function performed on the given test-data.

The next two parts of this section present an overview of assertion-checking functions and TDD files.

3.1 Assertion-Checking Functions

An assertion-checking function is a wrapper around the function under test which, in addition to calling the function under test, also evaluates its ADL assertions to determine whether or not the function is behaving correctly. The assertion-checking function works roughly as follows:

1. Evaluate all sub-expressions qualified by the call-state operator, and save these results.
2. Call the function under test with the same parameters that were passed to it.
3. Evaluate all assertions after replacing the sub-expressions qualified by the call-state operator with their corresponding values as evaluated in the first step.
4. If any of the assertions evaluate to false, then report it as an error.

Advantages

The assertion-checking function isolates into a single function the task of checking that the function meets its specifications. Being a wrapper, assertion-checking functions can be used with any input. It could be used with exhaustive input which would not be feasible in any system that required human attention to each test case.

The assertion-checking function also isolates the task of checking from the preparation of test input and from function implementation.

Auxiliary definitions

If assertions refer to auxiliary code (such as functions), the assertion-checking functions will evaluate this auxiliary code. Hence, the implementations of auxiliary definitions have to be trusted. Care must be taken in choosing implementations of auxiliary definitions, for if they fail, they can jeopardize the testing process too.

3.2 TDD Files

The TDD file offers the test designer a formal and structured framework for describing test-data. The TDD file provides a structure through which the type of data used to test the function is characterized and documented. Through the use of the TDD syntax, test-data becomes the subject of a design process.

The testing process has both mechanical and creative aspects. Writing iterative loops to supply test-data to a function is rather mechanical work. However, deciding which test-data to supply requires some ingenuity. The possible universe of test-data for any one function parameter may be very large. The problem is breaking down that possibly very large universe into a finite set of test cases.

There are many analysis tools available for examining test results and relating them to the code under test, performing coverage analysis and error analysis of various sorts. TDD is not a tool in that category; instead, it is a design tool for test-data. It gives test designers a language for recording their insights in a systematic and easily communicated way. The design can then be used to help generate test programs.

The mathematical foundation for the language is the idea of dividing the possible test-data into equivalence classes and then selecting a representative for each class.

Equivalence and representatives

The basis for selecting a subset of all possible inputs is the notion of equivalence. Two test inputs are equivalent, with respect to a particular correctness check for a particular function under test, if they produce the same test result. The two inputs are equivalent if it is not possible to tell them apart by running the test.

Note that this definition of equivalence depends on the measure of correctness as well as on the function under test.

If a group of test inputs is equivalent, it is not necessary to run the test on more than one of the inputs; any one can be chosen. No additional information can be gathered by running the test on another input in the same equivalence class.

The art of the test designer is to define an equivalence partition; it requires insight into the implementation and into possible errors, as well as into the typical uses of the function under test and the structure of the data. There are tools to assist in the process of analysis. The TDD language is a tool to help record the results of the analysis, not to perform the analysis. In the terminology of [18], TDD is a tool to specify the problem partition.

The result of the analysis is a set of test cases that provide the desired coverage of the function under test. Mathematically, these are a set of representatives of the equivalence classes in the test input space. Operationally, they are the values on which to test the function.

Abstraction of test input

The TDD file describes the test-data symbolically. It is not simply a list of test inputs, but a set of characteristics of the test inputs. These characteristics are called *properties*. These symbolic properties serve as documentation of the test-data; they describe the intent of the data. In addition, by separating the characterization of the data from the actual data, the test specification can survive changes in the system environment.

Formally, a property is applied to a data type and notates a partition of that data type. Each datum in the data type is described by exactly one of the choices for the property. A data type is typically described by the cross-product of several properties. Each element in the data type is described by a tuple of choices from the cross-product of properties. The data type is hence partitioned into classes, each of which is an intersection of the classes determined by the individual properties of the data type.

Example

Consider testing a function that appends to the end of a file a transaction with a particular size and kind. In the TDD file, one would not list a set of numbers to use as the number of bytes, but instead describe the numbers symbolically:

```
prop trans_size =  
    {negative, zero, small, large, huge}
```

Thus, the size of a transaction is characterized by one of the properties: negative, zero, small, large, or huge. Similarly, the kind of transaction is described symbolically:

```
prop trans_kind =  
    {read, write, create, delete}
```

Anyone reading the TDD file has, at a glance, an idea of the tests that will be run on the function.

As a consequence, as the implementation is modified over time, the design of the test-data can survive. In testing a file system, the specific values for the size of a transaction will typically

depend on the block size of the file system implementation. By using an abstract definition, the same test design can be used for any block-structured file system.

This symbolic description of the input, as well as being independent of the details of the data structures described, is also independent of the particular implementation of the function under test. While the TDD file should be written with insight into the possibilities of an implementation, it specifies a black box test. There is no direct input in terms of code coverage, for example, although feedback from such measurements can certainly be used to improve the quality of the input descriptions when developing and running tests.

Provide functions

The TDD file gives a symbolic description of the desired test input. That symbolic description is converted into actual data values during the execution of a test, by *provide functions*. These are functions written by the test engineer and linked with the generated test program; they encapsulate the task of generating test values. All too often, the complexity of the mechanism that produces test-data hides the intent of the data. Provide functions, in contrast, have a well-defined interface and perform a clearly defined task: to generate one data value with a specific set of characteristics described by a set of properties.

Isolating the data-generation code into the provide functions is an important benefit. The often-messy details of generating data values are encapsulated in the provide function; the property arguments to the provide function express the intent of the test variable. Provide functions are modular, reusable, and have a clearly defined task; the technique would be useful even without automatic test generation.

Relation to ADL and the test program

The purpose of the TDD language is to describe data values that will be used in tests. Those values are data in the host programming language. Values of test variables are described symbolically; the test designer writes a provide function to translate those symbolic descriptions to programming language data during the execution of the test. The ADL type of the test variable determines the signature of the provide function.

A test directive in the TDD file is translated into a test driver program; that test driver makes calls to the provide functions and to the assertion-checking functions.

A TDD file is written with respect to a fixed ADL file; the type names and function names that can be used in the TDD file are those declared in the ADL file. There can be more than one TDD file for a particular ADL file.

4 Conclusions and Future Work

In this paper, we presented an overview of a capability to test software components in a systematic and non-intrusive manner. We have not yet published any details of our experience in implementing and using a test generation system implementing these capabilities. We are currently

constructing an implementation targeted for an ANSI C environment, which will be made widely available.

There is also scope to extend our capability in many ways. Keeping the ADL specifications and assertion-checking functions, we can replace the TDD file with many other test-data generating schemes. A straightforward approach is the generation of random data; another approach is the generation of large volumes of (possibly similar) data for stress-testing.

We are also considering more involved extensions of our testing capability. One extension is to test the behavior of combinations of operations (as opposed to unit testing). As a motivating example, there is very little that unit testing can do for a stack in this case; the push and pop operations have to be run in various combinations for comprehensive testing. Another extension is to test the behavior of an operation when multiple processes attempt to access it simultaneously. This happens frequently with system calls.

Research is currently underway to develop methods for translating ADL specifications into natural language documents. This will be the subject of a future paper.

Finally, we intend to apply the ADL specification methods to other languages such as C++ and Ada.

5 Acknowledgements

The original work on ADL was done by Sun Microsystems Laboratories, Inc. This has now been extended in collaboration with X/Open Company Limited with funding from the Information-Technology Promotion Agency, Japan. All results are being made publicly available and open industry review is invited.

6 References

- [1] Belz, Frank and Luckham, David C. “A New Approach to Prototyping Ada-based Hardware/Software Systems.” *Proceedings of the ACM Tri-Ada Conference* (December 1990).
- [2] Bjorner, D. “VDM '87—A Formal Method at Work.” Vol. 252 of *Lecture Notes in Computer Science*. New York: Springer-Verlag, 1987.
- [3] Boyer, R. S., B. Elspas, and K. N. Levitt. “SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution.” *Proceedings of the International Conference on Reliable Software* (April 1975): 234–245.
- [4] Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification*. revision 1.1. OMG document number 91.12.1 edition. December 1991.

- [5] Gannon, J., P. McMullin, and R. Hamlet. "Data-abstraction Implementation, Specification, and Testing." *ACM Transactions on Programming Languages and Systems* 3, no. 3 (July 1981): 211–223.
- [6] Goodenough, J. B. and S. L. Gerhart. "Towards a Theory of Test-data Selection." *Proceedings of the International Conference on Reliable Software* (April 1975): 493–510.
- [7] Guttag, J. V., J. J. Horning, and J. M. Wing. "The Larch family of Specification Languages." *IEEE Software* 2, no. 5 (September 1985): 24–36.
- [8] Infotech International. *Infotech State of the Art Report. Software Testing Volume 1: Analysis and Bibliography*. 1979.
- [9] King, J. C. "A New Approach to Program Testing." *Proceedings of the International Conference on Reliable Software* (April 1975): 228–233.
- [10] Liskov, B. and S. Zilles. "Specification Techniques for Data Abstraction." *IEEE Transactions on Software Engineering* SE-1, no. 1 (March 1975): 7–19.
- [11] Luckham, David C., Friedrich W. von Henke, Bernd Krieg-Bruckner, and Olaf Owe. "ANNA, A Language for Annotating Ada Programs." Vol. 260 of *Lecture Notes in Computer Science*. New York: Springer-Verlag, 1987.
- [12] Meyers, G. J. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.
- [13] Richardson, D. J., S. L. Aha, and T. O. O'Malley. "Specification-based Test Oracles for Reactive Systems." *Proceedings of the Fourteenth International Conference on Software Engineering* (May 1992).
- [14] Sankar, S. "Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs." Ph.D. thesis, Stanford University, 1989. Also Stanford University Department of Computer Science Technical Report No. STAN-CS-89-1282 and Computer Systems Laboratory Technical Report No. CSL-TR-89-391.
- [15] Spivey, J. M. "Understanding Z, A Specification Language and its Formal Semantics." *Tracts in Theoretical Computer Science*, vol. 3. Cambridge University Press, 1988.
- [16] Sun Microsystems, Inc., U. S. A. and Information-Technology Promotion Agency, Japan. *ADL Language Reference Manual*. document no. MITI/0002/D/0.1 edition. August 1993.
- [17] Sun Microsystems, Inc., U. S. A. and Information-Technology Promotion Agency, Japan. *ADL Translator Design Specification*. document no. MITI/0001/D/0.1 edition. August 1993.
- [18] Weyuker, Elaine J. and Thomas J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE Transactions on Software Engineering* 6, no. 3 (May 1980): 236–246.

A Note on Distributed Computing

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall

Introduction by Jim Waldo

Unlike the majority of the technical reports produced by Sun Microsystems Laboratories, "A Note on Distributed Computing" does not report on the results of a project. Instead, this technical report sets the context for the problem being addressed by the Large-Scale Distributed Systems group, made up of the authors. This group had the goal of investigating how to create applications that could scale up to millions of machines on a network — a network size that was unheard of at the time.

When the report was written, the goal of most distributed computing infrastructure projects, such as the Object Management Groups (OMG), Common Object Request Broker Architecture (CORBA), was to simplify the production of distributed systems by making the programming of such systems look like the programming of non-distributed systems. The infrastructure, it was thought, could be built to remove all of the differences between references to a remote service and references to a local service. This paper argues that all such attempts are doomed to failure and can, at best, enable the development of systems that are fragile, prone to unavoidable errors, and restricted in scale. In particular, we argued that distributed infrastructures must present a model of partial failure to the programmer, since only at the application level can such failure be dealt with; must deal with concurrency issues, rather than leaving them to the infrastructure; and must at the application level realize what parts of the program are local and what parts are at least potentially remote.

Within the Large-Scale Distributed Systems group, the paper became the foundation for a program of research that attempted to find simple mechanisms for the building of distributed systems without attempting to mask the distinctions at the programming model level between local and remote interactions. Part of the model that was adopted was to build systems that were simplified by being centered around a single language. A direct result of this work was the invention (by Ann Wollrath) of the Remote Method Invocation system (RMI) that has become a standard part of the Java™ platform. The paper also formed the basis of a design philosophy that led to the Jini™ Networking technology, in which many of the techniques and programming models first explored in the Large-Scale Distributed Systems project have found commercial use.

While originally written to set the context for a particular research group, the paper has had considerably broader impact than originally imagined. It has been widely cited, and has been reprinted in Vitek and Tschudin (eds), *Mobile Object Systems* (Springer, 1996) and Waldo, et. al., *The Jini Specification* (Addison Wesley, 2001). Indeed, the conclusions of the paper have become part of the accepted wisdom in the distributed computing community.

A Note on Distributed Computing

Jim Waldo
Geoff Wyant
Ann Wollrath
Sam Kendall

SMLI TR-94-29

November 1994

Abstract:

We argue that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space. These differences are required because distributed systems require that the programmer be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure.

We look at a number of distributed systems that have attempted to paper over the distinction between local and remote objects, and show that such systems fail to support basic requirements of robustness and reliability. These failures have been masked in the past by the small size of the distributed systems that have been built. In the enterprise-wide distributed systems foreseen in the near future, however, such a masking will be impossible.

We conclude by discussing what is required of both systems-level and application-level programmers and designers if one is to take distribution seriously.

 **Sun Microsystems**
Laboratories, Inc.

A Sun Microsystems, Inc. Business

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:

jim.waldo@east.sun.com
geoff.wyant@east.sun.com
ann.wollrath@east.sun.com
sam.kendall@east.sun.com

A Note on Distributed Computing

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Much of the current work in distributed, object-oriented systems is based on the assumption that objects form a single ontological class. This class consists of all entities that can be fully described by the specification of the set of interfaces supported by the object and the semantics of the operations in those interfaces. The class includes objects that share a single address space, objects that are in separate address spaces on the same machine, and objects that are in separate address spaces on different machines (with, perhaps, different architectures). On the view that all objects are essentially the same kind of entity, these differences in relative location are merely an aspect of the implementation of the object. Indeed, the location of an object may change over time, as an object migrates from one machine to another or the implementation of the object changes.

It is the thesis of this note that this unified view of objects is mistaken. There are fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects. Further, work in distributed object-oriented systems that is based on a model that ignores or denies these differences is doomed to failure, and could easily lead to an industry-wide rejection of the notion of distributed object-based systems.

1.1 Terminology

In what follows, we will talk about local and distributed computing. By *local computing* (local object invocation, etc.), we mean programs that are confined to a single address space. In contrast, we will use the term *distributed computing* (remote object invocation, etc.) to refer to programs that make calls to other address spaces, possibly on another machine. In the case of distributed computing, nothing is known about the recipient of the call (other than that it supports a particular interface). For example, the client of such a distributed object does not know the hardware architecture on which the recipient of the call is running, or the language in which the recipient was implemented.

Given the above characterizations of “local” and “distributed” computing, the categories are not exhaustive. There is a middle ground, in which calls are made from one address space to another but in which some characteristics of the called object are known. An important class of this sort consists of calls from one address space to another on the same machine; we will discuss these later in the paper.

2 The Vision of Unified Objects

There is an overall vision of distributed object-oriented computing in which, from the programmer's point of view, there is no essential distinction between objects that share an address space and objects that are on two machines with different architectures located on different continents. While this view can most recently be seen in such works as the Object Management Group's Common Object Request Broker Architecture (CORBA) [1], it has a history that includes such research systems as Arjuna [2], Emerald [3], and Clouds [4].

In such systems, an object, whether local or remote, is defined in terms of a set of interfaces declared in an interface definition language. The implementation of the object is independent of the interface and hidden from other objects. While the underlying mechanisms used to make a method call may differ depending on the location of the object, those mechanisms are hidden from the programmer who writes exactly the same code for either type of call, and the system takes care of delivery.

This vision can be seen as an extension of the goal of remote procedure call (RPC) systems to the object-oriented paradigm. RPC systems attempt to make cross-address space function calls look (to the client programmer) like local function calls. Extending this to the object-oriented programming paradigm allows papering over not just the marshalling of parameters and the unmarshalling of results (as is done in RPC systems) but also the locating and connecting to the target objects. Given the isolation of an object's implementation from clients of the object, the use of objects for distributed computing seems natural. Whether a given object invocation is local or remote is a function of the implementation of the objects being used, and could possibly change from one method invocation to another on any given object.

Implicit in this vision is that the system will be "objects all the way down"; that is, that all current invocations or calls for system services will be eventually converted into calls that might be to an object residing on some other machine. There is a single paradigm of object use and communication used no matter what the location of the object might be.

In actual practice, of course, a local member function call and a cross-continent object invocation are not the same

thing. The vision is that developers write their applications so that the objects within the application are joined using the same programmatic glue as objects between applications, but it does not require that the two kinds of glue be implemented the same way. What is needed is a variety of implementation techniques, ranging from same-address-space implementations like Microsoft's Object Linking and Embedding [5] to typical network RPC; different needs for speed, security, reliability, and object co-location can be met by using the right "glue" implementation.

Writing a distributed application in this model proceeds in three phases. The first phase is to write the application without worrying about where objects are located and how their communication is implemented. The developer will simply strive for the natural and correct interface between objects. The system will choose reasonable defaults for object location, and depending on how performance-critical the application is, it may be possible to alpha test it with no further work. Such an approach will enforce a desirable separation between the abstract architecture of the application and any needed performance tuning.

The second phase is to tune performance by "concretizing" object locations and communication methods. At this stage, it may be necessary to use as yet unavailable tools to allow analysis of the communication patterns between objects, but it is certainly conceivable that such tools could be produced. Also during the second phase, the right set of interfaces to export to various clients—such as other applications—can be chosen. There is obviously tremendous flexibility here for the application developer. This seems to be the sort of development scenario that is being advocated in systems like Fresco [6], which claim that the decision to make an object local or remote can be put off until after initial system implementation.

The final phase is to test with "real bullets" (e.g., networks being partitioned, machines going down). Interfaces between carefully selected objects can be beefed up as necessary to deal with these sorts of partial failures introduced by distribution by adding replication, transactions, or whatever else is needed. The exact set of these services can be determined only by experience that will be gained during the development of the system and the first applications that will work on the system.

A central part of the vision is that if an application is built using objects all the way down, in a proper object-oriented

fashion, the right “fault points” at which to insert process or machine boundaries will emerge naturally. But if you initially make the wrong choices, they are very easy to change.

One conceptual justification for this vision is that whether a call is local or remote has no impact on the correctness of a program. If an object supports a particular interface, and the support of that interface is semantically correct, it makes no difference to the correctness of the program whether the operation is carried out within the same address space, on some other machine, or off-line by some other piece of equipment. Indeed, seeing location as a part of the implementation of an object and therefore as part of the state that an object hides from the outside world appears to be a natural extension of the object-oriented paradigm.

Such a system would enjoy many advantages. It would allow the task of software maintenance to be changed in a fundamental way. The granularity of change, and therefore of upgrade, could be changed from the level of the entire system (the current model) to the level of the individual object. As long as the interfaces between objects remain constant, the implementations of those objects can be altered at will. Remote services can be moved into an address space, and objects that share an address space can be split and moved to different machines, as local requirements and needs dictate. An object can be repaired and the repair installed without worry that the change will impact the other objects that make up the system. Indeed, this model appears to be the best way to get away from the “Big Wad of Software” model that currently is causing so much trouble.

This vision is centered around the following principles that may, at first, appear plausible:

- there is a single natural object-oriented design for a given application, regardless of the context in which that application will be deployed;
- failure and performance issues are tied to the implementation of the components of an application, and consideration of these issues should be left out of an initial design; and
- the interface of an object is independent of the context in which that object is used.

Unfortunately, all of these principles are false. In what follows, we will show why these principles are mistaken, and why it is important to recognize the fundamental differences between distributed computing and local computing.

3 Déjà Vu All Over Again

For those of us either old enough to have experienced it or interested enough in the history of computing to have learned about it, the vision of unified objects is quite familiar. The desire to merge the programming and computational models of local and remote computing is not new.

Communications protocol development has tended to follow two paths. One path has emphasized integration with the current language model. The other path has emphasized solving the problems inherent in distributed computing. Both are necessary, and successful advances in distributed computing synthesize elements from both camps.

Historically, the language approach has been the less influential of the two camps. Every ten years (approximately), members of the language camp notice that the number of distributed applications is relatively small. They look at the programming interfaces and decide that the problem is that the programming model is not close enough to whatever programming model is currently in vogue (messages in the 1970s [7], [8], procedure calls in the 1980s [9], [10], [11], and objects in the 1990s [1], [2]). A furious bout of language and protocol design takes place and a new distributed computing paradigm is announced that is compliant with the latest programming model. After several years, the percentage of distributed applications is discovered not to have increased significantly, and the cycle begins anew.

A possible explanation for this cycle is that each round is an evolutionary stage for both the local and the distributed programming paradigm. The repetition of the pattern is a result of neither model being sufficient to encompass both activities at any previous stage. However, (this explanation continues) each iteration has brought us closer to a unification of the local and distributed computing models. The current iteration, based on the object-oriented approach to both local and distributed programming, will

be the one that produces a single computational model that will suffice for both.

A less optimistic explanation of the failure of each attempt at unification holds that any such attempt will fail for the simple reason that programming distributed applications is not the same as programming non-distributed applications. Just making the communications paradigm the same as the language paradigm is insufficient to make programming distributed programs easier, because communicating between the parts of a distributed application is not the difficult part of that application.

The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. The hard problems in distributed computing concern insuring adequate performance and dealing with problems of concurrency. The hard problems have to do with differences in memory access paradigms between local and distributed entities. People attempting to write distributed applications quickly discover that they are spending all of their efforts in these areas and not on the communications protocol programming interface.

This is not to argue against pleasant programming interfaces. However, the law of diminishing returns comes into play rather quickly. Even with a perfect programming model of complete transparency between “fine-grained” language-level objects and “larger-grained” distributed objects, the number of distributed applications would not be noticeably larger if these other problems have not been addressed.

All of this suggests that there is interesting and profitable work to be done in distributed computing, but it needs to be done at a much higher-level than that of “fine-grained” object integration. Providing developers with tools that help manage the complexity of handling the problems of distributed application development as opposed to the generic application development is an area that has been poorly addressed.

4 Local and Distributed Computing

The major differences between local and distributed computing concern the areas of latency, memory access, partial

failure, and concurrency.¹ The difference in latency is the most obvious, but in many ways is the least fundamental. The often overlooked differences concerning memory access, partial failure, and concurrency are far more difficult to explain away, and the differences concerning partial failure and concurrency make unifying the local and remote computing models impossible without making unacceptable compromises.

4.1 Latency

The most obvious difference between a local object invocation and the invocation of an operation on a remote (or possibly remote) object has to do with the latency of the two calls. The difference between the two is currently between four and five orders of magnitude, and given the relative rates at which processor speed and network latency speeds are changing, the difference in the future promises to be at best no better, and will likely be worse. It is this disparity in efficiency that is often seen as the essential difference between local and distributed computing.

Ignoring the difference between the performance of local and remote invocations can lead to designs whose implementations are virtually assured of having performance problems because the design requires a large amount of communication between components that are in different address spaces and on different machines. Ignoring the difference in the time it takes to make a remote object invocation and the time it takes to make a local object invocation is to ignore one of the major design areas of an application. A properly designed application will require determining, by understanding the application being designed, what objects can be made remote and what objects must be clustered together.

The vision outlined earlier, however, has an answer to this objection. The answer is two-pronged. The first prong is to rely on the steadily increasing speed of the underlying hardware to make the difference in latency irrelevant. This, it is often argued, is what has happened to efficiency concerns having to do with everything from high level languages to virtual memory. Designing at the cutting edge has always required that the hardware catch up before the design is efficient enough for the real world. Arguments from efficiency seem to have gone out of style in software

1. We are not the first to notice these differences; indeed, they are clearly stated in [12].

engineering, since in the past such concerns have always been answered by speed increases in the underlying hardware.

The second prong of the reply is to admit to the need for tools that will allow one to see what the pattern of communication is between the objects that make up an application. Once such tools are available, it will be a matter of tuning to bring objects that are in constant contact to the same address space, while moving those that are in relatively infrequent contact to wherever is most convenient. Since the vision allows all objects to communicate using the same underlying mechanism, such tuning will be possible by simply altering the implementation details (such as object location) of the relevant objects. However, it is important to get the application correct first, and after that one can worry about efficiency.

Whether or not it will ever become possible to mask the efficiency difference between a local object invocation and a distributed object invocation is not answerable *a priori*. Fully masking the distinction would require not only advances in the technology underlying remote object invocation, but would also require changes to the general programming model used by developers.

If the only difference between local and distributed object invocations was the difference in the amount of time it took to make the call, one could strive for a future in which the two kinds of calls would be conceptually indistinguishable. Whether the technology of distributed computing has moved far enough along to allow one to plan products based on such technology would be a matter of judgement, and rational people could disagree as to the wisdom of such an approach.

However, the difference in latency between the two kinds of calls is only the most obvious difference. Indeed, this difference is not really the fundamental difference between the two kinds of calls, and that even if it were possible to develop the technology of distributed calls to an extent that the difference in latency between the two sorts of calls was minimal, it would be unwise to construct a programming paradigm that treated the two calls as essentially similar. In fact, the difference in latency between local and remote calls, because it is so obvious, has been the only difference most see between the two, and has tended to mask the more irreconcilable differences.

4.2 Memory access

A more fundamental (but still obvious) difference between local and remote computing concerns the access to memory in the two cases—specifically in the use of pointers. Simply put, pointers in a local address space are not valid in another (remote) address space. The system can paper over this difference, but for such an approach to be successful, the transparency must be complete. Two choices exist: either all memory access must be controlled by the underlying system, or the programmer must be aware of the different types of access—local and remote. There is no inbetween.

If the desire is to completely unify the programming model—to make remote accesses behave as if they were in fact local—the underlying mechanism must totally control all memory access. Providing distributed shared memory is one way of completely relieving the programmer from worrying about remote memory access (or the difference between local and remote). Using the object-oriented paradigm to the fullest, and requiring the programmer to build an application with “objects all the way down,” (that is, only object references or values are passed as method arguments) is another way to eliminate the boundary between local and remote computing. The layer underneath can exploit this approach by marshalling and unmarshalling method arguments and return values for intra-address space transmission.

But adding a layer that allows the replacement of all pointers to objects with object references only *permits* the developer to adopt a unified model of object interaction. Such a unified model cannot be *enforced* unless one also removes the ability to get address-space-relative pointers from the language used by the developer. Such an approach erects a barrier to programmers who want to start writing distributed applications, in that it requires that those programmers learn a new style of programming which does not use address-space-relative pointers. In requiring that programmers learn such a language, moreover, one gives up the complete transparency between local and distributed computing.

Even if one were to provide a language that did not allow obtaining address-space-relative pointers to objects (or returned an object reference whenever such a pointer was requested), one would need to provide an equivalent way of making cross-address space reference to entities other

than objects. Most programmers use pointers as references for many different kinds of entities. These pointers must either be replaced with something that can be used in cross-address space calls or the programmer will need to be aware of the difference between such calls (which will either not allow pointers to such entities, or do something special with those pointers) and local calls. Again, while this could be done, it does violate the doctrine of complete unity between local and remote calls. Because of memory access constraints, the two *have* to differ.

The danger lies in promoting the myth that “remote access and local access are exactly the same” and not enforcing the myth. An underlying mechanism that does not unify all memory accesses while still promoting this myth is both misleading and prone to error. Programmers buying into the myth may believe that they do not have to change the way they think about programming. The programmer is therefore quite likely to make the mistake of using a pointer in the wrong context, producing incorrect results. “Remote is just like local,” such programmers think, “so we have just one unified programming model.” Seemingly, programmers need not change their style of programming. In an incomplete implementation of the underlying mechanism, or one that allows an implementation language that in turn allows direct access to local memory, the system does not take care of all memory accesses, and errors are bound to occur. These errors occur because the programmer is not aware of the difference between local and remote access and what is actually happening “under the covers.”

The alternative is to explain the difference between local and remote access, making the programmer aware that remote address space access is very different from local access. Even if some of the pain is taken away by using an interface definition language like that specified in [1] and having it generate an intelligent language mapping for operation invocation on distributed objects, the programmer aware of the difference will not make the mistake of using pointers for cross-address space access. The programmer will know it is incorrect. By not masking the difference, the programmer is able to learn when to use one method of access and when to use the other.

Just as with latency, it is logically possible that the difference between local and remote memory access could be completely papered over and a single model of both presented to the programmer. When we turn to the problems

introduced to distributed computing by partial failure and concurrency, however, it is not clear that such a unification is even conceptually possible.

4.3 Partial failure and concurrency

While unlikely, it is at least logically possible that the differences in latency and memory access between local computing and distributed computing could be masked. It is not clear that such a masking could be done in such a way that the local computing paradigm could be used to produce distributed applications, but it might still be possible to allow some new programming technique to be used for both activities. Such a masking does not even seem to be logically possible, however, in the case of partial failure and concurrency. These aspects appear to be different in kind in the case of distributed and local computing.²

Partial failure is a central reality of distributed computing. Both the local and the distributed world contain components that are subject to periodic failure. In the case of local computing, such failures are either total, affecting all of the entities that are working together in an application, or detectable by some central resource allocator (such as the operating system on the local machine).

This is not the case in distributed computing, where one component (machine, network link) can fail while the others continue. Not only is the failure of the distributed components independent, but there is no common agent that is able to determine what component has failed and inform the other components of that failure, no global state that can be examined that allows determination of exactly what error has occurred. In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link.

These sorts of failures are not the same as mere exception raising or the inability to complete a task, which can occur in the case of local computing. This type of failure is caused when a machine crashes during the execution of an object invocation or a network link goes down, occurrences that cause the target object to simply disappear rather than return control to the caller. A central problem in distributed computing is insuring that the state of the

2. In fact, authors such as Schroeder [12] and Hadzilacos and Toueg [13] take partial failure and concurrency to be the defining problems of distributed computing.

whole system is consistent after such a failure; this is a problem that simply does not occur in local computing.

The reality of partial failure has a profound effect on how one designs interfaces and on the semantics of the operations in an interface. Partial failure requires that programs deal with indeterminacy. When a local component fails, it is possible to know the state of the system that caused the failure and the state of the system after the failure. No such determination can be made in the case of a distributed system. Instead, the interfaces that are used for the communication must be designed in such a way that it is possible for the objects to react in a consistent way to possible partial failures.

Being robust in the face of partial failure requires some expression at the interface level. Merely improving the implementation of one component is not sufficient. The interfaces that connect the components must be able to state whenever possible the cause of failure, and there must be interfaces that allow reconstruction of a reasonable state when failure occurs and the cause cannot be determined.

If an object is coresident in an address space with its caller, partial failure is not possible. A function may not complete normally, but it always completes. There is no indeterminism about how much of the computation completed. Partial completion can occur only as a result of circumstances that will cause the other components to fail.

The addition of partial failure as a possibility in the case of distributed computing does not mean that a single object model cannot be used for both distributed computing and local computing. The question is not “can you make remote method invocation look like local method invocation?” but rather “what is the price of making remote method invocation identical to local method invocation?” One of two paths must be chosen if one is going to have a unified model.

The first path is to treat all objects as if they were local and design all interfaces as if the objects calling them, and being called by them, were local. The result of choosing this path is that the resulting model, when used to produce distributed systems, is essentially indeterministic in the face of partial failure and consequently fragile and non-robust. This path essentially requires ignoring the extra failure modes of distributed computing. Since one can’t

get rid of those failures, the price of adopting the model is to require that such failures are unhandled and catastrophic.

The other path is to design all interfaces as if they were remote. That is, the semantics and operations are all designed to be deterministic in the face of failure, both total and partial. However, this introduces unnecessary guarantees and semantics for objects that are never intended to be used remotely. Like the approach to memory access that attempts to require that all access is through system-defined references instead of pointers, this approach must also either rely on the discipline of the programmers using the system or change the implementation language so that all of the forms of distributed indeterminacy are forced to be dealt with on all object invocations.

This approach would also defeat the overall purpose of unifying the object models. The real reason for attempting such a unification is to make distributed computing more like local computing and thus make distributed computing easier. This second approach to unifying the models makes local computing as complex as distributed computing. Rather than encouraging the production of distributed applications, such a model will discourage its own adoption by making all object-based computing more difficult.

Similar arguments hold for concurrency. Distributed objects by their nature must handle concurrent method invocations. The same dichotomy applies if one insists on a unified programming model. Either all objects must bear the weight of concurrency semantics, or all objects must ignore the problem and hope for the best when distributed. Again, this is an interface issue and not solely an implementation issue, since dealing with concurrency can take place only by passing information from one object to another through the agency of the interface. So either the overall programming model must ignore significant modes of failure, resulting in a fragile system; or the overall programming model must assume a worst-case complexity model for all objects within a program, making the production of any program, distributed or not, more difficult.

One might argue that a multi-threaded application needs to deal with these same issues. However, there is a subtle difference. In a multi-threaded application, there is no real source of indeterminacy of invocations of operations. The application programmer has complete control over invocation order when desired. A distributed system by its nature

introduces truly asynchronous operation invocations. Further, a non-distributed system, even when multi-threaded, is layered on top of a single operating system that can aid the communication between objects and can be used to determine and aid in synchronization and in the recovery of failure. A distributed system, on the other hand, has no single point of resource allocation, synchronization, or failure recovery, and thus is conceptually very different.

5 The Myth of “Quality of Service”

One could take the position that the way an object deals with latency, memory access, partial failure, and concurrency control is really an aspect of the implementation of that object, and is best described as part of the “quality of service” provided by that implementation. Different implementations of an interface may provide different levels of reliability, scalability, or performance. If one wants to build a more reliable system, one merely needs to choose more reliable implementations of the interfaces making up the system.

On the surface, this seems quite reasonable. If I want a more robust system, I go to my catalog of component vendors. I quiz them about their test methods. I see if they have ISO9000 certification, and I buy my components from the one I trust the most. The components all comply with the defined interfaces, so I can plug them right in; my system is robust and reliable, and I’m happy.

Let us imagine that I build an application that uses the (mythical) queue interface to enqueue work for some component. My application dutifully enqueues records that represent work to be done. Another application dutifully dequeues them and performs the work. After a while, I notice that my application crashes due to time-outs. I find this extremely annoying, but realize that it’s my fault. My application just isn’t robust enough. It gives up too easily on a time-out. So I change my application to retry the operation until it succeeds. Now I’m happy. I almost never see a time-out. Unfortunately, I now have another problem. Some of the requests seem to get processed two, three, four, or more times. How can this be? The component I bought which implements the queue has allegedly been rigorously tested. It shouldn’t be doing this. I’m angry. I call the vendor and yell at him. After much finger-pointing and research, the culprit is found. The problem turns out to be the way I’m using the queue. Because of

my handling of partial failures (which in my naiveté, I had thought to be total), I have been enqueueing work requests multiple times.

Well, I yell at the vendor that it is still their fault. Their queue should be detecting the duplicate entry and removing it. I’m not going to continue using this software unless this is fixed. But, since the entities being enqueued are just values, there is no way to do duplicate elimination. The only way to fix this is to change the protocol to add request IDs. But since this is a standardized interface, there is no way to do this.

The moral of this tale is that robustness is not simply a function of the implementations of the interfaces that make up the system. While robustness of the individual components has some effect on the robustness of the overall systems, it is not the sole factor determining system robustness. Many aspects of robustness can be reflected only at the protocol/interface level.

Similar situations can be found throughout the standard set of interfaces. Suppose I want to reliably remove a name from a context. I would be tempted to write code that looks like:

```
while (true) {
    try {
        context->remove(name);
        break;
    }
    catch (NotFoundInContext) {
        break;
    }
    catch (NetworkServerFailure) {
        continue;
    }
}
```

That is, I keep trying the operation until it succeeds (or until I crash). The problem is that my connection to the name server may have gone down, but another client’s may have stayed up. I may have, in fact, successfully removed the name but not discovered it because of a network disconnection. The other client then adds the same name, which I then remove. Unless the naming interface includes an operation to lock a naming context, there is no way that I can make this operation completely robust. Again, we see that robustness/reliability needs to be expressed at the interface level. In the design of any opera-

tion, the question has to be asked: what happens if the client chooses to repeat this operation with the exact same parameters as previously? What mechanisms are needed to ensure that they get the desired semantics? These are things that can be expressed only at the interface level. These are issues that can't be answered by supplying a "more robust implementation" because the lack of robustness is inherent in the interface and not something that can be changed by altering the implementation.

Similar arguments can be made about performance. Suppose an interface describes an object which maintains sets of other objects. A defining property of sets is that there are no duplicates. Thus, the implementation of this object needs to do duplicate elimination. If the interfaces in the system do not provide a way of testing equality of reference, the objects in the set must be queried to determine equality. Thus, duplicate elimination can be done only by interacting with the objects in the set. It doesn't matter how fast the objects in the set implement the equality operation. The overall performance of eliminating duplicates is going to be governed by the latency in communicating over the slowest communications link involved. There is no change in the set implementations that can overcome this. An interface design issue has put an upper bound on the performance of this operation.

6 Lessons from NFS

We do not need to look far to see the consequences of ignoring the distinction between local and distributed computing at the interface level. NFS[®], Sun's distributed computing file system [14], [15] is an example of a non-distributed application programmer interface (API) (open, read, write, close, etc.) re-implemented in a distributed way.

Before NFS and other network file systems, an error status returned from one of these calls indicated something rare: a full disk, or a catastrophe such as a disk crash. Most failures simply crashed the application along with the file system. Further, these errors generally reflected a situation that was either catastrophic for the program receiving the error or one that the user running the program could do something about.

NFS opened the door to partial failure within a file system. It has essentially two modes for dealing with an inaccessi-

ble file server: soft mounting and hard mounting. But since the designers of NFS were unwilling (for easily understandable reasons) to change the interface to the file system to reflect the new, distributed nature of file access, neither option is particularly robust.

Soft mounts expose network or server failure to the client program. Read and write operations return a failure status much more often than in the single-system case, and programs written with no allowance for these failures can easily corrupt the files used by the program. In the early days of NFS, system administrators tried to tune various parameters (time-out length, number of retries) to avoid these problems. These efforts failed. Today, soft mounts are seldom used, and when they are used, their use is generally restricted to read-only file systems or special applications.

Hard mounts mean that the application hangs until the server comes back up. This generally prevents a client program from seeing partial failure, but it leads to a malady familiar to users of workstation networks: one server crashes, and many workstations—even those apparently having nothing to do with that server—freeze. Figuring out the chain of causality is very difficult, and even when the cause of the failure can be determined, the individual user can rarely do anything about it but wait. This kind of brittleness can be reduced only with strong policies and network administration aimed at reducing interdependencies. Nonetheless, hard mounts are now almost universal.

Note that because the NFS protocol is stateless, it assumes clients contain no state of interest with respect to the protocol; in other words, the server doesn't care what happens to the client. NFS is also a "pure" client-server protocol, which means that failure can be limited to three parties: the client, the server, or the network. This combination of features means that failure modes are simpler than in the more general case of peer-to-peer distributed object-oriented applications where no such limitation on shared state can be made and where servers are themselves clients of other servers. Such peer-to-peer distributed applications can and will fail in far more intricate ways than are currently possible with NFS.

The limitations on the reliability and robustness of NFS have nothing to do with the implementation of the parts of that system. There is no "quality of service" that can be improved to eliminate the need for hard mounting NFS volumes. The problem can be traced to the interface upon

which NFS is built, an interface that was designed for non-distributed computing where partial failure was not possible. The reliability of NFS cannot be changed without a change to that interface, a change that will reflect the distributed nature of the application.

This is not to say that NFS has not been successful. In fact, NFS is arguably the most successful distributed application that has been produced. But the limitations on the robustness have set a limitation on the scalability of NFS. Because of the intrinsic unreliability of the NFS protocol, use of NFS is limited to fairly small numbers of machines, geographically co-located and centrally administered. The way NFS has dealt with partial failure has been to informally require a centralized resource manager (a system administrator) who can detect system failure, initiate resource reclamation and insure system consistency. But by introducing this central resource manager, one could argue that NFS is no longer a genuinely distributed application.

7 Taking the Difference Seriously

Differences in latency, memory access, partial failure, and concurrency make merging of the computational models of local and distributed computing both unwise to attempt and unable to succeed. Merging the models by making local computing follow the model of distributed computing would require major changes in implementation languages (or in how those languages are used) and make local computing far more complex than is otherwise necessary. Merging the models by attempting to make distributed computing follow the model of local computing requires ignoring the different failure modes and basic indeterminacy inherent in distributed computing, leading to systems that are unreliable and incapable of scaling beyond small groups of machines that are geographically co-located and centrally administered.

A better approach is to accept that there are irreconcilable differences between local and distributed computing, and to be conscious of those differences at all stages of the design and implementation of distributed applications. Rather than trying to merge local and remote objects, engineers need to be constantly reminded of the differences between the two, and know when it is appropriate to use each kind of object.

Accepting the fundamental difference between local and remote objects does not mean that either sort of object will require its interface to be defined differently. An interface definition language such as IDL can still be used to specify the set of interfaces that define objects. However, an additional part of the definition of a class of objects will be the specification of whether those objects are meant to be used locally or remotely. This decision will need to consider what the anticipated message frequency is for the object, and whether clients of the object can accept the indeterminacy implied by remote access. The decision will be reflected in the interface to the object indirectly, in that the interface for objects that are meant to be accessed remotely will contain operations that allow reliability in the face of partial failure.

It is entirely possible that a given object will often need to be accessed by some objects in ways that cannot allow indeterminacy, and by other objects relatively rarely and in a way that does allow indeterminacy. Such cases should be split into two objects (which might share an implementation) with one having an interface that is best for local access and the other having an interface that is best for remote access.

A compiler for the interface definition language used to specify classes of objects will need to alter its output based on whether the class definition being compiled is for a class to be used locally or a class being used remotely. For interfaces meant for distributed objects, the code produced might be very much like that generated by RPC stub compilers today. Code for a local interface, however, could be much simpler, probably requiring little more than a class definition in the target language.

While writing code, engineers will have to know whether they are sending messages to local or remote objects, and access those objects differently. While this might seem to add to the programming difficulty, it will in fact aid the programmer by providing a framework under which he or she can learn what to expect from the different kinds of calls. To program completely in the local environment, according to this model, will not require any changes from the programmer's point of view. The discipline of defining classes of objects using an interface definition language will insure the desired separation of interface from implementation, but the actual process of implementing an interface will be no different than what is done today in an object-oriented language.

Programming a distributed application will require the use of different techniques than those used for non-distributed applications. Programming a distributed application will require thinking about the problem in a different way than before it was thought about when the solution was a non-distributed application. But that is only to be expected. Distributed objects are different from local objects, and keeping that difference visible will keep the programmer from forgetting the difference and making mistakes. Knowing that an object is outside of the local address space, and perhaps on a different machine, will remind the programmer that he or she needs to program in a way that reflects the kinds of failures, indeterminacy, and concurrency constraints inherent in the use of such objects. Making the difference visible will aid in making the difference part of the design of the system.

Accepting that local and distributed computing are different in an irreconcilable way will also allow an organization to allocate its research and engineering resources more wisely. Rather than using those resources in attempts to paper over the differences between the two kinds of computing, resources can be directed at improving the performance and reliability of each.

One consequence of the view espoused here is that it is a mistake to attempt to construct a system that is “objects all the way down” if one understands the goal as a distributed system constructed of the *same kind* of objects all the way down. There will be a line where the object model changes; on one side of the line will be distributed objects, and on the other side of the line there will (perhaps) be local objects. On either side of the line, entities on the other side of the line will be opaque; thus one distributed object will not know (or care) if the implementation of another distributed object with which it communicates is made up of objects or is implemented in some other way. Objects on different sides of the line will differ in kind and not just in degree; in particular, the objects will differ in the kinds of failure modes with which they must deal.

8 A Middle Ground

As noted in Section 2, the distinction between local and distributed objects as we are using the terms is not exhaustive. In particular, there is a third category of objects made up of those that are in different address spaces but are guaranteed to be on the same machine. These are the sorts

of objects, for example, that appear to be the basis of systems such as Spring [16] or Clouds [4]. These objects have some of the characteristics of distributed objects, such as increased latency in comparison to local objects and the need for a different model of memory access. However, these objects also share characteristics of local objects, including sharing underlying resource management and failure modes that are more nearly deterministic.

It is possible to make the programming model for such “local-remote” objects more similar to the programming model for local objects than can be done for the general case of distributed objects. Even though the objects are in different address spaces, they are managed by a single resource manager. Because of this, partial failure and the indeterminacy that it brings can be avoided. The programming model for such objects will still differ from that used for objects in the same address space with respect to latency, but the added latency can be reduced to generally acceptable levels. The programming models will still necessarily differ on methods of memory access and concurrency, but these do not have as great an effect on the construction of interfaces as additional failure modes.

The other reason for treating this class of objects separately from either local objects or generally distributed objects is that a compiler for an interface definition language can be significantly optimized for such cases. Parameter and result passing can be done via shared memory if it is known that the objects communicating are on the same machine. At the very least, marshalling of parameters and the unmarshalling of results can be avoided.

The class of locally distributed objects also forms a group that can lead to significant gains in software modularity. Applications made up of collections of such objects would have the advantage of forced and guaranteed separation between the interface to an object and the implementation of that object, and would allow the replacement of one implementation with another without affecting other parts of the system. Because of this, it might be advantageous to investigate the uses of such a system. However, this activity should not be confused with the unification of local objects with the kinds of distributed objects we have been discussing.

9 References

- [1] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number 91.12.1* (1991).
- [2] [Parrington, Graham D. "Reliable Distributed Programming in C++: The Arjuna Approach." *USENIX 1990 C++ Conference Proceedings* (1991).
- [3] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald." *IEEE Transactions on Software Engineering* SE-13, no. 1, (January 1987).
- [4] Dasgupta, P., R. J. Leblanc, and E. Spafford. "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System." *Georgia Institute of Technology Technical Report GIT-ICS-85/29*. (1985).
- [5] Microsoft Corporation. *Object Linking and Embedding Programmers Reference*. version 1. Microsoft Press, 1992.
- [6] Linton, Mark. "A Taste of Fresco." Tutorial given at the *8th Annual X Technical Conference* (January 1994).
- [7] Jaayeri, M., C. Ghezzi, D. Hoffman, D. Middleton, and M. Smotherman. "CSP/80: A Language for Communicating Sequential Processes." *Proceedings: Distributed Computing CompCon* (Fall 1980).
- [8] Cook, Robert. "MOD- A Language for Distributed Processing." *Proceedings of the 1st International Conference on Distributed Computing Systems* (October 1979).
- [9] Birrell, A. D. and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* 2 (1978).
- [10] Hutchinson, N. C., L. L. Peterson, M. B. Abott, and S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques." *Proceedings of the Twelfth Symposium on Operating Systems Principles* 23, no. 5 (1989).
- [11] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. *Network Computing Architecture*. Prentice Hall, 1990.
- [12] Schroeder, Michael D. "A State-of-the-Art Distributed System: Computing with BOB." In *Distributed Systems*, 2nd ed., S. Mullender, ed., ACM Press, 1993.
- [13] Hadzilacos, Vassos and Sam Toueg. "Fault-Tolerant Broadcasts and Related Problems." In *Distributed Systems*, 2nd ed., S. Mullender, ed., ACM Press, 1993.
- [14] Walsh, D., B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. "Overview of the SUN Network File System." *Proceedings of the Winter Usenix Conference* (1985).
- [15] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the SUN Network File System." *Proceedings of the Summer Usenix Conference* (1985).
- [16] Khalidi, Yousef A. and Michael N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the Winter USENIX Conference* (1993). Also *Sun Microsystems Laboratories, Inc. Technical Report SMLI TR-92-3* (December 1992).

Programming as an Experience: The Inspiration for Self

Randall B. Smith and David Ungar

Introduction by David Ungar

The Self project started in 1986 as a language design exercise at Xerox PARC. Randy Smith and I (with a germ of inspiration from Peter Deutsch) tried to come up with a language design that, without implementation compromises, would give the programmer even more of the benefits of object-oriented programming than Smalltalk (our then-favorite) did. To that end, we sought simplicity by unifying disparate concepts and were pleased and surprised to discover that the simplification enhanced rather than diminished the language's expressive power. This experience was so significant to us that we chose to call the first Self paper: "Self: The Power of Simplicity."

Randy went off to England, and I went off in search of a research area for my graduate students and myself at Stanford. Taking a leap of faith, I decided to see if it were actually possible to turn this paper design into a full-fledged programming environment, with a speedy implementation and rich graphical user interface. Craig Chambers, Elgin Lee, Bay-Wei Chang, Urs Hoelzle, and Ole Agesen joined me at Stanford to pursue this vision. Emil Sarpa, Wayne Rosing, and Bill Joy at Sun also became interested in our project and, along with other corporations and government grants, helped provide the early funding.

We were able to achieve some interesting results in the areas of object-oriented high-performance implementation techniques and the application of cartoon animation to enhance the legibility of graphical user interfaces. In 1991, the project left Stanford to join the fledgling Sun Microsystems Laboratories. I was delighted when Randy Smith decided to leave PARC and join us as co-leader of the project here at Sun. And Jim Mitchell, now head of Sun Labs, lent us his management expertise, helping us settle on a specific roadmap and schedule for creating the full Self environment. Over the next two years, three more talented individuals, Lars Bak, John Maloney, and Mario Wolczko joined us at Sun.

Almost three years later, we had reached our goal of developing a complete system with many novel characteristics, only to be dealt a difficult blow. Feeling that we would not be able to attract a significant user community, Sun Microsystems Laboratories decided to cancel the project. Since then, the members of the Self group have gone on to contribute to object-oriented systems both inside and outside of Sun:

- Craig Chambers and Urs Hoelzle, two Ph.D. students who were sponsored by Sun, have had successful academic careers and made their own contributions to the field of object-oriented languages and implementation techniques.
- Bay-Wei Chang went to Xerox PARC to work on graphical interfaces.
- John Maloney left Sun to join Alan Kay at Apple, where he was instrumental in implementing Squeak, a publicly-available Smalltalk system. Later John reimplemented the Morphic GUI framework that he and Lars had built under Randy's direction for Self.
- Lars Bak and Urs Hoelzle joined a startup, which was later acquired by Sun and built the Java HotSpot™ Java™ Virtual Machine, the core of Sun's desktop and server Java platform.

- Ole Agesen went on to start and drive the ExactVM project, Sun's first JVM™ with accurate garbage collection. (See paper #15 – The GC Interface in the EVM.)
- Mario Wolczko became West Coast tech lead for the ExactVM project for the first six months of the project, manager of the Spotless (Java™ Virtual Machine for small devices) project during its transition to product land, and has recently led the design of the hardware performance instrumentation architecture in a design for a future SPARC™ processor.
- Randy Smith exploited the Self system as a basis for several other projects in the Labs, including the Distributed Tutored Video Instruction work, (See paper #17 – Experiments Comparing Face-to-Face with Virtual Collaborative Learning). The work done in Self also influenced the "Nebraska" project and the JAMM (Java Applets Made Multi-User) Project.
- The K Virtual Machine (KVM) Java implementation, ubiquitous on cell phones today, is based on the Sun Labs Spotless project, undertaken by Antero Taivalsaari and Bill Bush. Antero and Bill were initially hired to participate in a successor to the Self project.
- Finally, David Ungar was loaned out to JavaSoft™ from Sun Labs. There he built the portability framework for Java HotSpot™ and built the first version of HotSpot to run on SPARC™ instead of Intel processors. (David just ported the interpreter—others built the compilers for SPARC.)

The various facets of Self have had varying amounts of impact. NewtonScript and other, lesser-known languages were heavily influenced by the language. The Morphic GUI framework has been used by many in the Squeak community. But the largest impact has been in the area of implementation techniques.

Perhaps because we were forced to think ambitiously in order to make Self practical, we found techniques, such as transparent, adaptive optimization, that have been harnessed for Java and many other subsequent projects. Descendants of Self's implementation techniques have aided Java's acceptance by slashing the performance penalty faced by Java's adopters.

All of Sun's Java Virtual Machines since the original have benefited from key participation by Self alumni or extended alumni (such as Bill Bush, Antero Taivalsaari, and Phillip Yelland).

REFERENCES:

Retrospective:

Programming as an Experience: The Inspiration for Self (1995) Randall B. Smith and David Ungar. Invited paper for ECOOP'95. Reprinted in "Prototype-Based Programming: Concepts, Languages and Applications," James Noble, I. Moore, A. Taivalsaari, eds. Springer-Verlag.

Language:

Self: The Power of Simplicity (1987) David Ungar and Randall B. Smith. OOPSLA'87. Revised version published in Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991. TR-94-30

Parents are Shared Parts: Inheritance and Encapsulation in Self (1991) Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hoelzle. Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

Organizing Programs Without Classes (1991) David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hoelzle. Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

Implementation:

Object Storage and Inheritance for Self (1988) Elgin Lee. Stanford University Engineer's thesis.

Customization: Optimizing Compiler Technology for Self, a Dynamically-Typed Object-Oriented Programming Language (1989) Craig Chambers and David Ungar. PLDI'89.

An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes (1989) Craig Chambers, David Ungar, and Elgin Lee. OOPSLA'89. Also in the Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs (1990) Craig Chambers and David Ungar. PLDI'90. Also in the Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

Making Pure Object-Oriented Languages Practical (1991) Craig Chambers and David Ungar. OOPSLA'91.

Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches (1991) Urs Hoelzle, Craig Chambers, and David Ungar. ECOOP'91.

The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages (1992) Craig Chambers. Stanford Doctoral Dissertation.

Debugging Optimized Code with Dynamic Deoptimization (1992) Urs Hoelzle, Craig Chambers, and David Ungar. OOPSLA'92.

Object, Message, and Performance: How They Coexist in Self (1992) David Ungar, Randall B. Smith, Craig Chambers, and Urs Hoelzle. I.E.E.E. Computer Magazine, October, 1992. Reprinted in Readings in Object-Oriented Systems and Applications, David Rine, ed, IEEE Computer Society Press.

Fast Write Barrier for Generational Garbage Collection," U. Hoelzle, OOPSLA'93.

Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback (1994) Urs Hoelzle and David Ungar. PLDI'94.

Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming (1994) Urs Hoelzle. Stanford doctoral dissertation.

A Third-Generation Self Implementation, Urs Hoelzle and David Ungar. OOPSLA '94.

Do object-oriented languages need special hardware support? Urs Hoelzle and David Ungar. ECOOP '95.

Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming, Urs Hoelzle, 1994 Stanford doctoral dissertation. TR-95-35

The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages, Craig Chambers. 1992 Stanford doctoral dissertation.

User Interface:

Experiencing Self Objects: An Object-Based Artificial Reality (1990) Bay-Wei Chang and David Ungar.

The Use-Mention Perspective on Programming for the Interface (1992) Randall B. Smith, David Ungar, and Bay-Wei Chang. In Computer Languages for Programming User Interface Software, a workshop at the ACM SIGCHI'91 conference, Brad A. Myers, organizer and editor. Reprinted in Languages for Developing User Interfaces, Brad Myers, ed., Jones and Bartlett, London, 1992.

Animation: From Cartoons to the User Interface (1993) Bay-Wei Chang and David Ungar. UIST'93. TR 94-33

Getting Close to Objects: Object-Focused Programming Environments (1995) Bay-Wei Chang, David Ungar, and Randall B. Smith. In Visual Object Oriented Programming, Margaret Burnett, Adele Goldberg, & Ted Lewis, eds., Prentice-Hall, 1995.

The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility (1995) Randall B. Smith, John Maloney, and David Ungar. OOPSLA'95.

Concreteness and Cartoon Animation in Seity, A user Interface for Object-Oriented Programming, Bay-Wei Chang. 1996 Stanford Doctoral Dissertation.

Type Inference and Application Extraction:

Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance (ECOOP'93) Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach

Constraint-Based Type Inference and Parametric Polymorphism (SAS'94) Ole Agesen

Sifting Out the Gold: Delivering Compact Applications From an Exploratory Object-Oriented Programming Environment (OOPSLA'94) Ole Agesen and David Ungar.

The Cartesian Product Algorithm (ECOOP'95) Ole Agesen.

Type Feedback vs. Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages (OOPSLA'95) Ole Agesen and Urs Hoelzle.

The Design and Application of Type Inference for Object Oriented Programming Environments, Ole Agesen. 1995 Stanford doctoral dissertation. TR 96-52

Other:

Integrating Independently-Developed Components in Object-Oriented Languages (1993) Urs Hoelzle

Mango – A Parser Generator for Self (1994) Ole Agesen TR 94-27

Programming as an Experience: The Inspiration for Self

Randall B. Smith and David Ungar

Sun Microsystems Laboratories
2550 Casey, Ave. MS MTV29-116
Mountain View, CA 94043, USA

randall.smith@sun.com david.ungar@sun.com

Abstract. The Self system attempts to integrate intellectual and non-intellectual aspects of programming to create an overall experience. The language semantics, user interface, and implementation each help create this integrated experience. The language semantics embed the programmer in a uniform world of simple objects that can be modified without appealing to definitions of abstractions. In a similar way, the graphical interface puts the user into a uniform world of tangible objects that can be directly manipulated and changed without switching modes. The implementation strives to support the world-of-objects illusion by minimizing perceptible pauses and by providing true source-level semantics without sacrificing performance. As a side benefit, it encourages factoring. Although we see areas that fall short of the vision, on the whole, the language, interface, and implementation conspire so that the Self programmer lives and acts in a consistent and malleable world of objects.

1 Introduction

During the last decade, over a dozen papers published about Self have described the semantics, the implementation, and the user interface. But they have not completely articulated an important part of the work: our shared vision of what programming should be. This vision focuses on the overall experience of working with a programming system, and is perhaps as much a feeling thing as it is an intellectual thing. This paper gives us a chance to talk about this underlying inspiration, to review the project, and to make a few self-reflective comments about what we might have done differently. Although the authors have the luxury of commenting from an overview perspective, the reader should keep in mind that the this work is the result of efforts by the many individuals who have worked in the Self project over the years.

1.1 Motivation

Programmers are human beings, embedded in a world of sensory experience, acting and responding to more than just rational thought. Of course to be effective, programmers need logical language semantics, but they also need things like confidence, comfort, and satisfaction — aspects of experience which are beyond the domain of pure logic. These concerns have traditionally been addressed separately by putting the logic in the language and providing for the rest of experience with the programming environment. The Self system attempts to integrate the intellectual and experiential sides of programming.

In our vision, the Self programmer lives and acts in a consistent and malleable world, from the concrete motor-sensory, to the abstract, intellectual levels. At the lowest, motor-sensory level of experience, objects provide the foundation for natural interaction. Consequently, every visual element in Self, from the largest window to the smallest triangle is a directly manipulable object. At the higher, semantic levels of the language, there are many possible computational models: in order to harmonize with the sensory level, Self models computation exclusively in terms of objects. Thus, every piece of Self data, from the largest file to the smallest number is a directly manipulable object. And, in order to ensure that these objects could be directly experi-

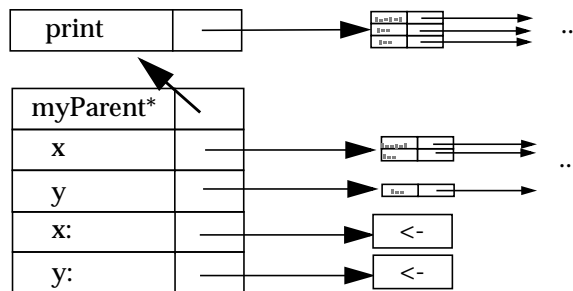


Figure 1. A Self point has **x** and **y** slots, with **x:** and **y:** slots containing the assignment primitive for changing **x** and **y**. The slot **myParent*** carries a “parent” denotation (shown as an asterisk). Parent slots are an inheritance link, indicating how message lookup continues beyond the object’s slots. For example, this point object will respond to a **print** message, because it inherits a **print** slot from the parent.

enced and manipulated, we devised a model based on “prototypes.” Just as a button can be added to any graphical object, so can a method be added to any individual object in the language, without needing to refer to a class. This prototype model and the use of objects for everything requires a radically new kind of implementation. In Self, implementation, interface, and language were designed to work together to create a unified programming experience.

In the following sections we in turn review the language semantics, the user interface, and the implementation. In each section we will try to point out where we think we succeeded and where we think we failed in being true to the vision.

2 Language Semantics

Self was initially designed by the authors at Xerox PARC [US87]. We employed a minimalist strategy, striving to distill an essence of object and message. Self has evolved over the years in design and implementation at Stanford University and most recently at Sun Microsystems Laboratories. A user interface and programming environment built in Self are part of the system: Self today is a fairly large system, and includes a complete programming environment and user interface framework.

A computation in Self consists solely of objects which in turn consist of slots. A slot has a name and a value. Slot names are always strings, but slot values can be any Self object. A slot can be marked with an asterisk to show that it designates a *parent*. Figure 1 illustrates a Self object representing a two-dimensional point with **x** and **y** slots, a parent slot called **myParent***, and two special assignment slots, **x:** and **y:**, that are used to assign to the **x** and **y** slots. The object’s parent has a single slot called **print** (containing a method object).

When sending a message, if no slot name matches within the receiving object, its parent’s slots are searched, and then slots in the parent’s parent, and so on. Thus our point object can respond to the messages **x**, **y**, **x:**, **y:**, and **myParent**, plus the message **print**, because it *inherits* the **print** slot from its parent. In Self, any object can potentially be a parent for any number of children, or it can be a child of any object. This uniform ability of any object to participate in any role of inheritance contributes to the consistency and malleability of Self and, we hope, contributes to the programmer’s comfort, confidence, and satisfaction.

In addition to slots, a Self object can include code. Such objects are called *methods*; since they do what methods in other languages do. For example, the object in the **print** slot above includes code and thus serves as a method. However in Self, any object can be regarded as a method; a “data” object contains code that merely returns itself. This viewpoint serves to unify computation with data access: when an object is found in a slot as a result of a message send it is *run*; data returns itself, while a method invokes its code. Thus when the **print** message is sent to our point object, the code in the print slot’s method will run immediately. This unification reinforces the interpretation that the experience of the client matters, not the inner details of the object used. For example, some soda machines dispense pre-mixed soda, while others dynamically mix syrup and carbonated water on demand. The user does not care to become involved in the distinction, what matters is only the end product, be it soda or the result of a computation.

Of course Self is not the only language that unifies access and computation. For example Beta [MMN93] does too. From a traditional computer science viewpoint, this unification serves to provide data abstraction; it is impossible to tell, even from within an abstraction, whether some value is stored or computed. However when designing Self we also sought to unify assignment and computation; this unification is slightly more unusual. Assignment in Self is performed with assignment slots, such as **x:** and **y:**, which contain a special method (symbolized by the arrow), that takes an argument (in addition to the receiver) and stuffs it in either the **x** or **y** slot. This desire for access/assignment symmetry can be interpreted as arising from the sensory-motor level of experience. From the time we are children, experience and manipulation are inextricably intertwined; we best experience an object when we can touch it, pick it up, turn it over, push its buttons, or even taste it. We believe that the notion of a container is a fundamental intuition that humans share and that by unifying assignment and computation in the same way as access and computation, Self allows abstraction over containerhood; since all containers are inspected or filled by sending messages; any object may pretend to be a container while employing a different implementation.

A contrast between Self and Beta may illustrate the role that our concern for a particular kind of programming experience played in the language design. In Beta, data takes two forms: values and references. Beta unifies accessing values with computation and also unifies assignment of values with computation, but uses a different syntax for accessing or changing references. For example, a data attribute containing a reference to a point would be accessed by saying `pt[]->` but a method in the receiver returning a reference to a point would be run by `pt->`. Giving the programmer two forms of data can be seen as giving the programmer more tools with which to work. More tools can of course be a good thing, but since in our opinion, the value/reference distinction does not really exist at the sensory-motor level of experience, we chose a slightly less elaborate scheme with a single kind of data structuring mechanism. We also believe there are advantages to uniformity in and of itself.

The design of slots in Self has proven to be challenging because of the unified access to state and behavior. In Self the same message can be sent to either read a data slot or run a method in the slot. Where should the distinction be maintained? We have taken the position that method objects are fundamentally different, in that they run non-trivial code whereas data objects just return themselves. Still, this behavior of method objects means that they cannot be directly manipulated because they would run. So we have had to invent primitive objects called *mirrors* as a way of indirectly mentioning methods. Mirrors can be better justified as encapsulators of reflective operations and as relieving each object of the burden of inheriting such behavior. But mirrors can hamper uniformity; it is sometimes unclear whether a method should take a mirror as argument or the object itself. Another approach to unifying invocation and access would be to add a bit to a slot that would record whether or not the slot was a method or data slot. This approach has its own problems: what would it mean to put 17 into a method slot? In our opinion, this issue is not fully resolved.

The treatment of assignment slots in Self is also a bit troublesome. A Self assignment slot is a slot containing a special primitive (the same one for all assignment slots), which uses the *name of the slot* (very odd) to find a corresponding data slot *in the same object* (also odd). This treatment leads to all sorts of special rules, for

instance it is illegal to have an object contain an assignment slot without a corresponding data slot, so the code that removes slots is riddled with extra checks. Also, this treatment fails to capture the intuitive notion of a container. Other prototype-based languages address this issue by having a slot-pair be an entity in the language and casting an assignable slot as such a slot pair. Another alternative might be to make the assignment object have a slot identifying its target, so that in principle any slot could assign to any other.

Messages can have extra arguments in addition to the receiver. For example, **3 + 4** sends the message **+** to the object **3**, with **4** as argument. In contrast to many other object-oriented languages numbers are objects in Self, just as in Smalltalk. Because languages like Smalltalk or Self do arithmetic by sending messages, programmers are free to add new numeric data types to the language, and the new types can inherit and reuse all the existing numeric code. Adding complex numbers or matrices to the system is straightforward: after defining a **+** slot for matrices, the user could have the matrix freely inherit from some slot with code that sends **+**. Code that sends **+** would then work for matrices as well as for integers. Self's juxtaposition of a simple and uniform language (objects for numbers and messages for arithmetic in this case) with a sophisticated implementation permits the programmer to inhabit a more consistent and malleable computational universe.

There is more to be said about the language: new Self objects are made simply by copying—there are no special class objects for instantiation (see the later section entitled Prototypes and Classes). The current implementation allows multiple inheritance, which requires a strategy for dealing with multiple parents. It has block closure objects, and threads. (A few constructs that would be somewhat obscure in a language like Self, such as methods contained in the local slots of methods, are not yet supported by the implementation.)

2.1 Discussion

Our desire to provide a certain kind of programming experience has colored Self's stance on some traditional issues:

Type Declarations. In order to understand the design of the Self language it helps to examine the assumptions that underlie language design. In the beginning, there were FORTRAN, ALGOL and Lisp. In all three of these languages the programmer only has to say what is necessary to execute programs. Since Lisp was interpreted, no type information was supplied at all. Since ALGOL and FORTRAN were compiled, it was necessary for programmers to specify primitive type information, such as whether a variable contained an integer or a float, in order for the compiler to generate the correct instructions. As compiled languages evolved, it was discovered that by adding more static declarations, the compiler could sometimes create more efficient code. For example, in PL/I procedures had to be explicitly declared to be recursive, so that the compiler could use a faster procedure prologue for the non-recursive ones.

Programmers noticed that this static declarative information could be of great value in making a program more understandable. Until then, the main benefit of declarations had been to the compiler, but with Simula¹ and PASCAL a movement was born; using declarations both to benefit human readers and compilers.

In our opinion, this trend has been a mixed blessing, especially where object-oriented languages are concerned. The problem is that the information a human needs to understand a program, or to reason about its correctness, is not necessarily the same information that a compiler needs to make a program run efficiently. But most languages with declarations confuse these two issues, either limiting the efficiency gained from declarations, or, more frequently hindering code reuse to such an extent that algorithms get duplicated and type systems subverted.

¹ SimulaTM is a trademark of a.s. Simula

Self therefore distinguishes between concrete and abstract types. Concrete types (embodied by *maps*) are completely hidden from the Self programmer. Maps are only visible to the implementation, where they are used as an efficiency mechanism. Abstract types on the other hand are notions that the programmer might think about. Self has no particular type manifestation in the language: declarative information is left to the environment. For example, one language level notion of abstract type, the *clone family*, is used in the work of Agesen et. al. [APS93] in their Self type inference work. There is no clone family object in the Self language, but such objects can be created and used by the programming environment. In order to structure complexity and provide the freest environment possible, we have layered the design so that the Self language proper includes only information needed to execute the program, leaving declarative information to the environment. This design keeps the language small, simplifies the pedagogy, and allows users to potentially extend the intensional domain of discourse.

Minimalism. Why have we tried to keep the Self language minimal? It is always tempting to add a new feature that handles some example better. Although the feature had made it possible to directly handle some examples, the burden it imposed in all reasoning about programs was just too much. We abandoned it for Self 3.0. Although adding features seems good, every new concept burdens every programmer who comes into contact with the language.

We have learned the hard way that smaller is better and that examples can be deceptive. Early in the evolution of Self we made three mistakes: prioritized multiple inheritance, the sender-path tie-breaker rule, and method-holder-based privacy semantics.¹ Each was motivated by a compelling example [CUCH91]. We prioritized

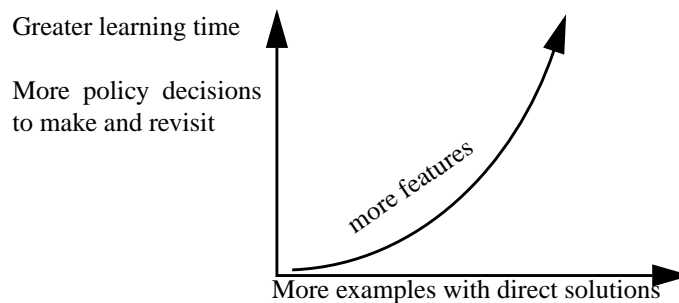


Figure 2. As more features are embedded in the language, the programmer gets to do more things immediately. But complexity grows with each feature: how the fundamental language elements interact with each other must be defined, so complexity growth can be combinatorial. Such complexity makes the basic language harder to learn, and can make it harder to use by forcing the programmer to make a choice among implementation options, a choice which may have to be revisited later.

multiple parent slots in order to support a mix-in style of programming. The sender-path tie-breaker rule allows two disjoint objects to be used as parents, for example a rectangle parent and a tree node parent for a VLSI cell object. The method-holder-based privacy semantics allowed objects with the same parents to be part of the same encapsulation domain, thereby supporting binary operations in a way that Smalltalk cannot [CUCH91].

But each feature also caused us no end of confusion. The prioritization of multiple parents implied that Self's "resend" (call-next-method) lookup had to be prepared to backup down parent links in order to follow lower-

¹ In all fairness, the first author was across the Atlantic at the time and had nothing to do with it. On the other hand, if he had not wandered off maybe these mistakes could have been avoided.

priority paths. The resultant semantics took five pages to write down, but we persevered. After a year's experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing "compiler bugs" that were merely unforeseen consequences of these features. It became clear that the language had strayed from its original path.

We now believe that when features, rules, or elaborations are motivated by particular examples, it is a good bet that their addition will be a mistake. The second author once coined the term "architect's trap" for something similar in the field of computer architecture; this phenomenon might be called "the language designer's trap."

If examples cannot be trusted, what do we think should motivate the language designer? Consistency and malleability. When there is only one way of doing things, it is easier to modify and reuse code. When code is reused, programs are easier to change and most importantly, shrink. When a program shrinks its construction and maintenance requires fewer people which allows for more opportunities for reuse to be found. Consistency leads to reuse, reuse leads to conciseness, conciseness leads to understanding. That is why we feel that it is hard to justify any type system that impedes reusability; the resultant duplication leads to a bigger program that is then harder to understand and to get right. Such type systems can be self-defeating.¹

Prototypes and Classes. There are now several fairly mature object-oriented languages based on prototypes. (For overviews see [Blas94], [DMC92], and [SLS94].) These languages differ somewhat in their treatment of semantic issues like privacy, copying, and the role of inheritance. (One notable system, Kevo [Tai93a], [Tai92], [Tai93] does not have delegation or inheritance at all.) All these languages have a model in which an object is in an important sense self-contained. Prototypes are often presented as an alternative to class-based language designs, so the subject of prototypes vs. classes can serve as point of (usually good natured) debate.

However, depending on how one defines "class," one may or may not think that classes are already present in a prototype based system. Some (e.g. [Blas94]) see a Self prototype as playing the role of class, since it determines the structure of its copies. Others note that much of the current Self system is organized in a particular way, using what we call "traits" objects in many places to provide common state and behavior for sharing among children. Such sharing is reminiscent of that provided by a class. However, classes normally also provide the description of an instance's implementation, and a "new" method for instantiation, neither of which are found in a traits object.

In a class-based system, any change (such as a new instance variable) to a class will affect new instances of a subclass. In Self, a change to a prototype (such as a new slot) will not affect anything other than the prototype itself (and its subsequent direct copies).² So we have implemented a "copy-down" mechanism in the environment to share implementation information. It allows the programmer to add and remove slots to an entire hierarchy of prototypes in a single operation. Functionality that is provided at the language level in class-based systems has risen to the programming environment level in Self. In general, the simple object model in Self means that some functionality omitted from the language may go back into the environment. Because the environment is built out of Self objects, the copy-down policy can be changed by the programmer. But such flexibility comes with a price. Now, there are two interfaces for adding slots to objects, the simple language level and the copying-down Self-object level. This loss of uniformity could be a source of confusion when writing a program that needs to add slots to objects. Only time will tell if the flexibility is worth the complication.

¹ However, by emphasizing the ability to express intuitive relationships, such as covariant specialization, over the ability to do all checking statically, it is possible to do a better job. See [MMM90].

² Self prototypes are not really special objects, they are distinguished only by the fact that, by convention, they are copied. Any copy of the prototype would serve as a prototype equally well. Some other prototype-based systems take a different approach.

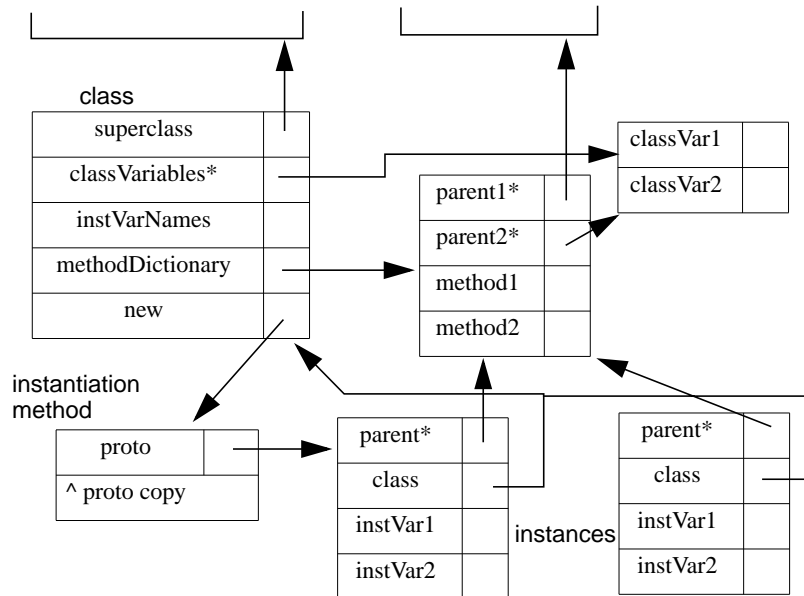


Figure 3. This figure suggests how Self objects might be composed to form Smalltalk-like class structures as demonstrated more completely by Wolczko [Wol95]. He shows that, with some caveats, Smalltalk code can be read into a Self system, parsed into Self objects, then executed with significant performance benefits, thanks to the Self's dynamically optimizing virtual machine.

A brief examination of the emulation of classes in Self will serve to illuminate both the nature of a prototype-based object model and the trade-off between implementing concepts in the language versus in the environment. In order to make a Self shared parent look more like a class, one could create a “new” method in the shared parent. This method could make a copy of some internal reference to a prototype, and so would appear to be an instantiation device. Figure 3 suggests how one might start to make a Smalltalk class out of Self objects. Mario Wolczko has built a more complete implementation of this, and has shown [Wol95] that it works quite well: he can read in Smalltalk source code and execute it as a Self program. There are certain restrictions on the Smalltalk source, but thanks to Self's implementation technology, once the code adaptively optimizes, the Self version of Smalltalk code will generally run faster than the Smalltalk version.

Do prototype-based systems like Self have classes? The answer would seem to be that if your definition of class is not satisfied by the existing language elements, you can probably build something quite easily that would make you happy. (It would be difficult to do this kind of trick in a prototype-based language that did not have an inheritance mechanism.) Of course, almost any language can emulate any other, but in this case the classes are built directly out of the prototype-based objects so directly that the classes constructed out of Self objects run faster than those built-in to Smalltalk. General meta-object issues in prototype-based languages are tackled by the Moostrap system [Mul95].

The use of traits might be seen as a carryover from the Self group's Smalltalk experience. Interestingly, it is likely that our old habits might not have done Self justice (as observed in [DMC92].) There are many other ways to organize Self objects other than by prototypes inheriting from a chain of traits parents, and many of these ways avoid a problem with the traits organization: a traits object appears to be an object but in fact cannot

respond to most of its messages. For example the point traits object lacks x and y slots and so cannot respond to `printString`, since its `printString` slot contains a method intended to be shared by point objects. We probably would have done better to put more effort into exploring other organizations. When investigating a new language, your old habits can lead you astray.

3 The User Interface and Programming Environment

Self is an unusually pure object-oriented language, because it uses objects and messages to achieve the effects of flow control, variable scoping, and primitive data types. This maniacal devotion to the object-message paradigm is intended to match a devotion to a user interface based on concrete, direct manipulation of uniform graphical objects. By matching the language to the user interface, we hope to create an experience of programming that can be learned more easily, and can be performed with less cognitive overhead.

The notion of direct manipulation has been around for many years now, and it is interesting to note that some of the earlier prototype-based systems were visual programming environments [BD81], [Smi87]. The current Self user interface [SMU95], [MS95] enhances the sense of direct manipulation by employing two principles we call *structural reification*, and *live editing*. We will define these principles, and show with an example how the interface brings the feeling of direct object experience to the task of creating objects and assembling applications.

3.1 Structural reification

The fundamental kind of display object in Self is called a “morph,” a term borrowed from Greek meaning essentially “thing.” Self provides a hierarchy of morphs. The root of the hierarchy is embodied in the prototypical morph, which appears as a kind of golden colored rectangle. Other object systems might choose to make the root of the graphical hierarchy an abstract class with no instances. But prototype systems usually provide generic examples of abstractions. This is an important part of the structural reification principle: there are no invisible display objects. Any descendant of the root morph (or any other morph for that matter) is guaranteed to be a fully functional graphical entity. It will inherit methods for displaying and responding to input events that enable it to be directly manipulated.

In keeping with the principle of structural reification, any morph can have “submorphs” attached to it. A submorph acts as though it is glued to the surface of its hosting morph. Composite graphical structure typical of direct manipulation interfaces arises through the morph-submorph hierarchy. Again, many systems provide compositing with special “group” objects which are normally invisible. But because we want things to feel very solid and direct, we chose to follow a simple metaphor of sticking morphs together as though glued to each other.

A final part of structural reification arises from the approach to submorph layout. Graphical user interfaces often require that subparts be lined up in a column or row. Self’s graphical elements are organized in space by “layout” objects that force their submorphs to line up as rows or columns. John Maloney [MS95] has shown how to create efficient “row and column morphs” as children of the generic morph. These objects are first class, tangible elements in the interface. They embody their layout policy as visible parts of the submorph hierarchy, so the user need only be able to access the submorphs in a structure in order to inspect or change the layout in some way. A possible price of this uniformity is paid by a user who does not wish to see the layout mechanism, but is confronted with it anyway.

An example in section 3.3 will illustrate how structural reification assures that any morph can be seen and can be grabbed, moved and inspected, and assures that graphical composition and layout constraints are physically present in the interface. Structural reification is an important part of making programming feel more tangible and direct.

3.2 Live Editing

Live editing simply means that at any time, an object can be directly changed by the user. Any interactive system that allows arbitrary runtime changes to its objects has a degree of support for live editing. But we believe Self provides an unusually direct interface to such live changes. The key to live editing is provided by Self's "meta menu," a menu that can pop up when the user holds the third mouse button while pointing to a morph. The meta menu contains items such as "resize," "dismiss," and "change color" which allow the user to edit the object directly. There are also menu elements that enable the user to "embed" the morph into the submorph structure of a morph behind it, and menu elements that give access to the submorph hierarchy at any point on the screen.

The "outliner" menu item creates a language-level view of the morph under the mouse¹. This view shows all of the slots in an object, and provides a full set of editing functionality. With an outliner you can add or remove slots, rename them, or edit their contents. Code for a method in a slot can be textually edited: outliners are important tools for programmers. Access to the outliner through the meta menu makes it possible to investigate the language-level object behind any graphical object on the screen.

The outliner supports the live editing principle by letting the user manipulate and edit slots, even while an object is "in use." The example below illustrates how a slot can be "carried" from one object to another, interactively modifying their language level structure.

Popping up the meta menu is the prototypical morph's response to the third mouse button click. All morphs inherit this behavior, even elements of the interface like outliners and pop-up menus themselves. But wait! Pop-up menus are impossible to click on: you find them under your mouse only when a mouse button is already down. To release the button in preparation for the third button click causes the pop-up to frustratingly disappear. Consequently, we provide a "pin down" button, which, when selected, causes the menu to become a normal, more permanent display object. The mechanism is not new, but providing it in Self enables the menu to be interactively pulled apart or otherwise modified by the user or programmer.

Live editing is partly a result of having an interactive system, but is enhanced by features in the user interface. This principle reinforces the feel that the programmer is working directly with concrete objects. The following example will clarify how this principle and the structural reification principle help give the programmer a feeling of a working in a uniform world of accessible, tangible objects.

3.3 Example of Direct Application Construction

Suppose the programmer (or motivated user) wishes to expand an ideal gas simulation, extending the functionality and adding user interface controls. The simulation starts simply as a box containing "atoms" which are bouncing around inside. Using the third mouse button, the user can invoke the meta menu, and select "outliner" to get the Self-level representation of the object (Figure 4). The outliner enables arbitrary language-level changes to the ideal gas simulation.

With the outliner, the user can start to create some controls right away. In the outliner, there are slots labeled "start" and "stop." These slots can be converted into user interface buttons by selecting from the middle-mouse-button pop-up menu on the slot (Figure 5). Pressing these buttons starts and stops the bouncing atoms in the simulation. This is an example of live editing at work: in just a few gestures, the programmer has gone through the outliner to create interface elements while the simulation continues to run.

The programmer may wish to create several such buttons, and arrange them in a row. The programmer selects "row morph" from a palette: when the buttons are embedded into the row, they immediately snap into position.

¹ Lars Bak designed the outliner framework for Self.

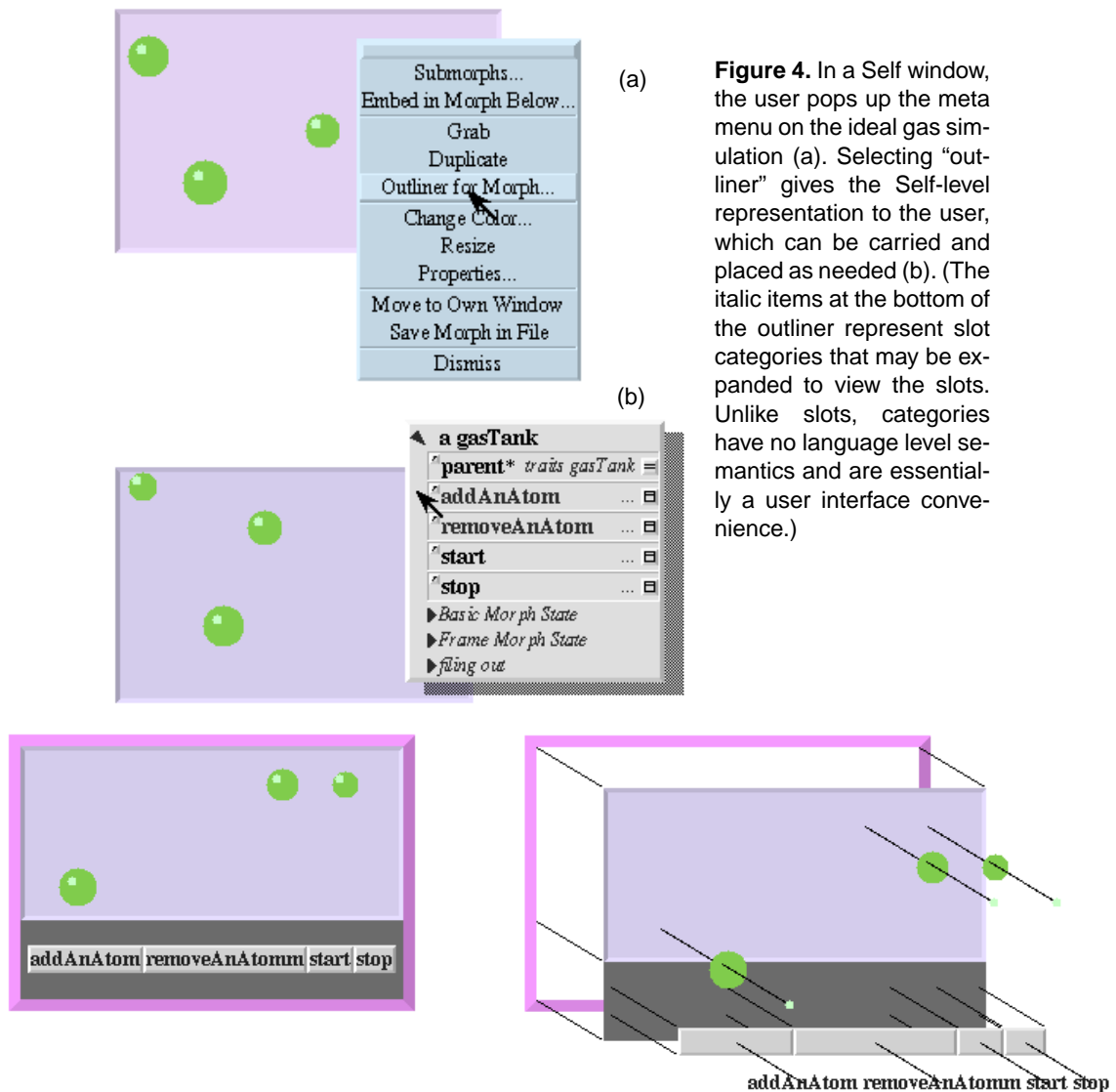


Figure 5. Composite graphical effects are achieved by embedding: any kind of morph can be embedded in any other kind of morph. The ideal gas simulation at left is a compound morph whose embedding relationships are shown at right.

Once the row of buttons is created, the programmer wishes to align the row below the gas tank. Again, the programmer can create a column frame: when the gas tank and the button row are embedded into the column frame, they line up one below the next (Figure 6)

Figure 6 also illustrates how composite graphical effects are achieved through the morph-submorph hierarchy. The interface employs morphs down to quite a low level. The labels on buttons, for example, are themselves first class morphs.

The uniformity of having “morphs all the way down” further reinforces the feel of working with concrete objects. For example, the user may wish to replace the textual label with an icon. The user can begin this task by

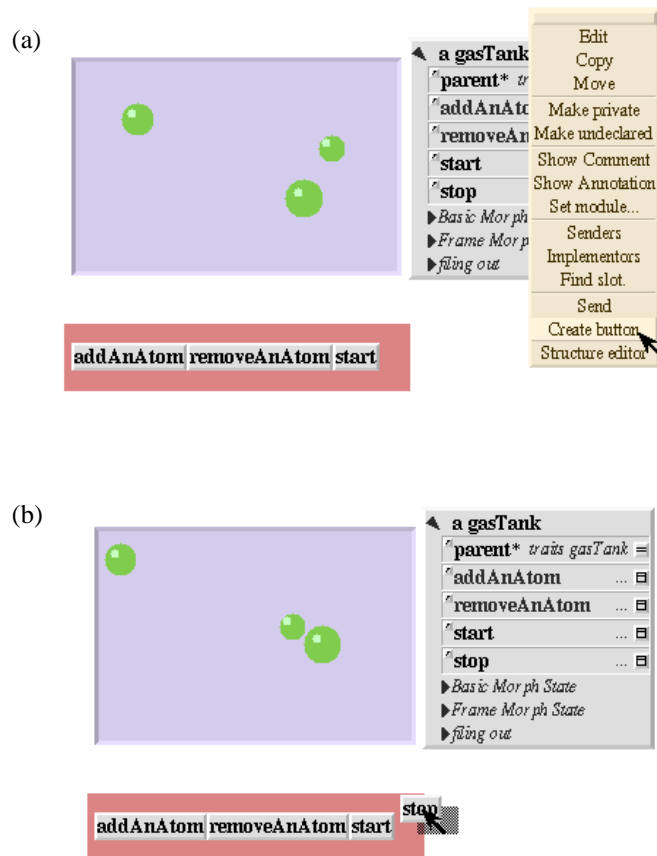


Figure 6. The middle mouse button pop up menu on the “stop” slot (a) enables the user to create a button for immediate use in the user interface (b). This button will be embedded in a row morph, so that it lines up horizontally.

pointing to the label and invoking the meta menu. There is a menu item labeled “submorphs” which allows the user to select which morph in the collection under the mouse he wishes to denote (see Figure 7). The user can remove the label directly from the button’s surface. In a similar way, the user can select one of the atoms in the gas tank and duplicate it. The new atom will serve as the icon replacing the textual label. Structural reification is at play here, making display objects accessible for direct and immediate modification.

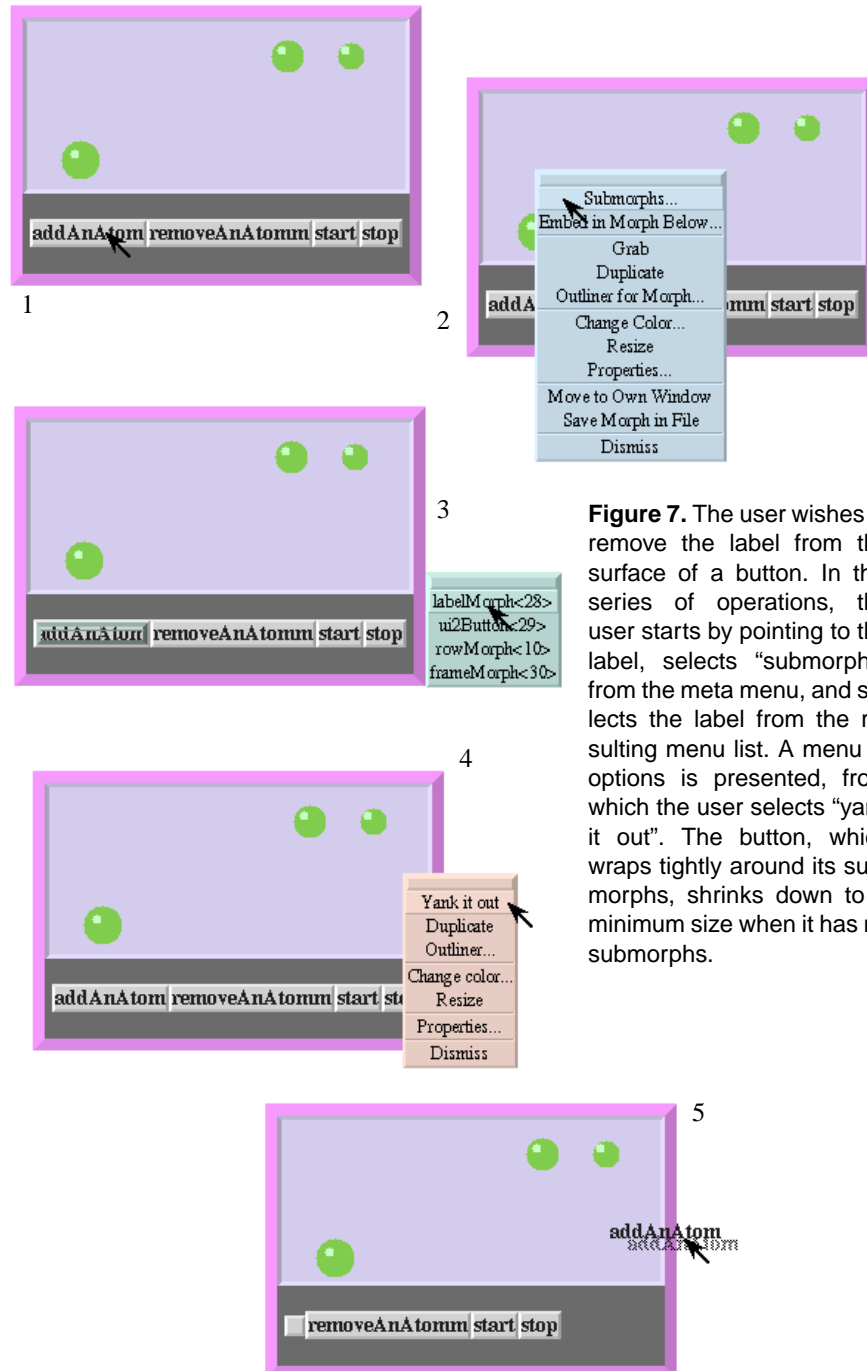


Figure 7. The user wishes to remove the label from the surface of a button. In this series of operations, the user starts by pointing to the label, selects “submorphs” from the meta menu, and selects the label from the resulting menu list. A menu of options is presented, from which the user selects “yank it out”. The button, which wraps tightly around its submorphs, shrinks down to a minimum size when it has no submorphs.

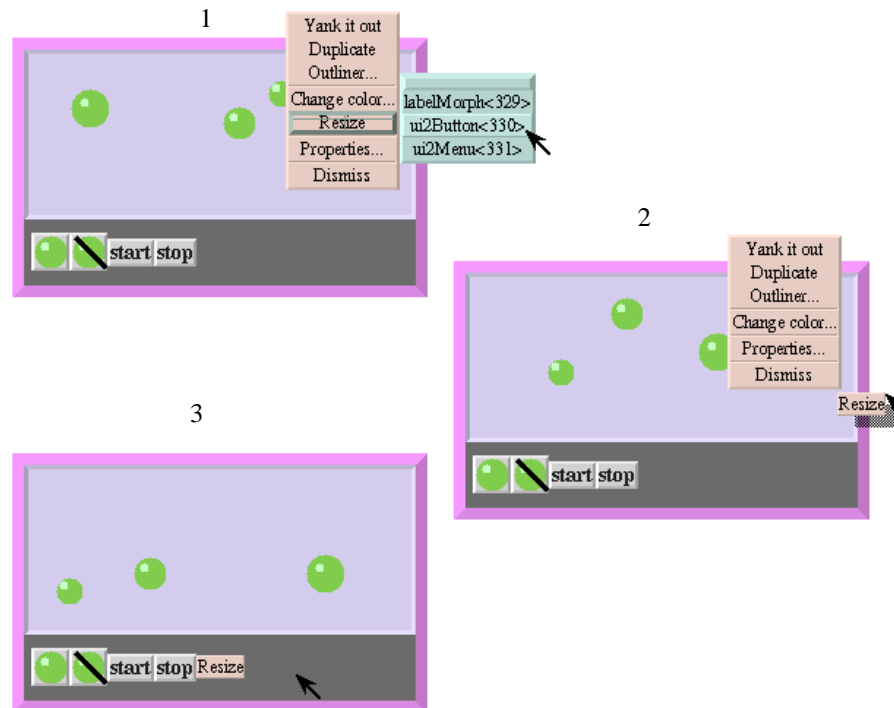


Figure 8. The environment itself is available for reuse. Here the user has created the menu of operations for the gas tank, which is now a submorph of the surrounding frame. He has “pinned down” this menu, by pressing the button at the top of the menu. He can then take the menu apart into constituent buttons: here he gets the resize button which is then incorporated into the simulation.

As mentioned above, all the elements of the interface such as pop-up menus and dialogue boxes are available for reuse. As an example, the programmer may want the gas tank in the simulation to be “resizable” by the simulation user. The programmer can create a resize button for the gas tank simply by “pinning down” the meta menu and removing the resize button from the menu, as illustrated in Figure 8. This button can then be embedded into the row of controls along with the other buttons.

Figure 9 illustrates how the programmer can modify the behavior of an individual atom to reveal its energy based upon color. A morph has a slot called “rawColor” that normally contains a “paint” object: in this example the programmer replaces that object with a method, so that the paint will be computed based upon energy level. When the change is accepted, the modified slot immediately takes effect. In Figure 10, the programmer is shown copying the slot into the atom’s parent object, so that it can be widely shared with other atoms. Of all atoms have a rawColor slot that overrides this slot in the parent. The programmer might at this point remove the rawColor slot from the prototypical atom, so that all new atoms will have this energy-based color.



Figure 9. The user has selected one atom on which to experiment. The user changes the “rawColor” slot from a computed to a stored value by editing directly in the atom’s outliner.



Figure 10. The user copies the modified rawColor slot as a first step in getting the computed method of coloring more widely shared. Because slots can be moved about readily, restructuring changes are relatively light weight, enhancing the sense of flexibility.

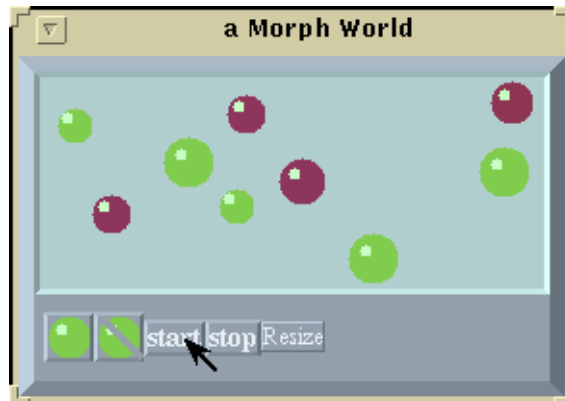


Figure 11. The completed application. The user has invoked a color changing tool that unifies colors across most submorphs, and has invoked the meta menu item “move to own window.”

Figure 11 shows the completed application. All the atoms now reveal their energy as they move, and controls for running the simulation appear in a row across the bottom of the object. The programmer has invoked a meta-menu item “move to own window” that wraps the application in its own window-system frame.

It is important to note that during this whole process, the simulation could be left running — there was no fundamental need to enter an “edit” mode, or even stop the atoms from bouncing around. The live editing principle makes the system feel responsive, and is reminiscent of the physical world’s concrete presence. Structural reification means that the parts of the interface are visible and accessible for direct modification.

3.4 Issues

While the principles of live editing and structural reification help create the sense of working within a world of tangible malleable objects, we could imagine going further. Here we discuss some of the things that interfere with full realization of our goals.

Multiple views: The very existence of the outliner as a separate view weakens the sense of directness that we are after. After all, when I want to add a slot to one of the simulated atoms, I must work on a different display object, the atom’s outliner. We have never had the courage or time to go after some of the wild ideas that would enable unification of any morph with its outliner. Ironically, Self’s first interface, Seity, probably did better on this issue [Cha95], [CUS95].

Self’s programming environment enforces the constraint that there only be one outliner on the screen at a time for a given object. If the user asks to see some object when its outliner is already on the screen, the outliner will do an animated slide over to the mouse cursor. This constraint encourages identification between the object and its outliner in the programmer’s mind. But as we mention above, particularly for morph objects, the identification does not always hold.

Because there is a difference between outliners and the objects they represent, there is a fundamental dichotomy in the system that interferes with the direct object experience goal. Is the programmer to believe that the outliner for some list object really is the list? If so, does the list have a graphical appearance as an intrinsic part of itself? Does the list have an intrinsic display location and a color? We have chosen the easy answer,

“no.” The object that represents the list to the programmer is not the same as the actual Self list object. Once we take this easy road, it is fundamentally impossible to always maintain the impression that the objects on the screen actually are the Self objects. Unfortunately, it is often clear that outliners are just display stand-ins for the real, invisible Self object, buried somewhere in the computer.

Text and object: There is a fundamental clash between the use of text and the use of direct manipulation. A word inherently denotes something, an object does not necessarily denote anything. That is, when you see the word “cow,” an image comes to mind. It is in fact difficult to avoid the image, that is the way words are supposed to work. They stand for things. However, when you manipulate a pencil, what comes to mind? It depends much more on who you are, what the context is, and so on. In other words, a pencil does not by itself denote anything. Consequently, textual notation and object manipulation are fundamentally from two different worlds. The pencil and the word denoting pencil are different entities.

Text is used quite a bit in Self, and its denotational character weakens the sense of direct encounter with objects. For example, many tools in the user interface employ a “printString” to denote an object. The programmer working with one of these tools might encounter the text “list (3, 7, 9).” The programmer might know that this denotes an object which could be viewed “directly” with an outliner. But why bother? The textual string often says all he needs to know. The programmer moves on, satisfied perhaps, yet not particularly feeling like he has encountered the list itself. The mind set in a denotational world is different than that in a direct object world, and use of text creates a different kind of experience.

Issues of use and mention in direct manipulation interfaces are discussed further in [SUC92].

3.5 Summary

The Self user interface helps the programmer feel that he or she is directly experiencing tangible objects. Two design principles help achieve this feeling. First, *structural reification*, assures that the graphical relationships at play in a particular arrangement of submorphs is manifest directly in display objects on the screen. Second, *live editing* means that there is no need for a course grained “edit mode;” rather, objects are always available for immediate and direct editing. The use of textual names for objects, and the distinction between an object and its representation are two problems that weaken the experience of directly working with objects. But on balance, we feel that the Self user interface successfully presents the illusion of being a world of readily modified, physically present objects.

4 Implementing Self

The implementation of a language is usually approached from a mathematical, or mechanistic viewpoint: what is desired is the creation of a program that interprets programs in another language (be the created program a compiler or interpreter). On the other hand, we have taken the view that the goal of the implementation is to fool the user into believing in the reality of the language. Even though Self objects have no physical existence, and there is no machine capable of executing Self methods, the implementation must strive to convince the user otherwise. That is why, despite all the tricks, the programmer can always debug at the source level, seeing all variables and single stepping, and can always change any method, even inlined ones, with no interference from the implementation.

4.1 Transparent Efficiency

The implementation techniques for Self have been presented previously so we will only summarize the briefly here. (See [Hol94], [HU94a], [HCU92], [HCU91], [Cha92], [CU91], [CU90], [CUL89], and [USCH92] for more details.)

Self presented large efficiency challenge because its pure semantics implied that every access, assignment, arithmetic operation and control structure had to be performed by sending a message. Worse yet, the Self user interface's uncompromising stance on structural reification placed further demands on efficiency: everything on the screen down to the smallest triangle is implemented by its own separate object, each with its own re-draw and active layout behavior. At the same time, in order to produce the experience of concrete objects, the system had to be as responsive as an interpreter. Self's implementors were confronted with a fundamental problem: to be responsive, the compiler could not afford to spend time on the elaborate optimizations needed for the language, no matter how effective they might be. Caught between Scylla and Charybdis, Self needed something completely different. Instead of relying on a single compiler for both speed and cleverness, Self adopted a hybrid system of two compilers: one fast, the other clever. Instead of always optimizing everything, type feedback permits the system to adaptively optimize code without introducing long pauses.

Figure 12 shows an overview of the compilation process of the system. The first time that a method is invoked,

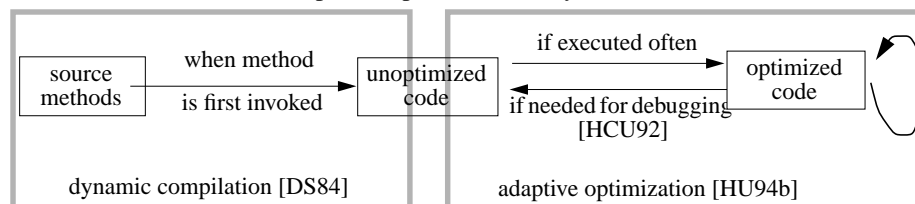


Figure 12. Compilation in the Self system (from [HU94b]).

the virtual machine uses dynamic compilation to create an “instrumented” version of the method that counts its invocations and the types of the receivers at each call site. When the invocation counter crosses some threshold, the optimizing compiler is automatically invoked and is guided by the counters at the call sites. In this way, Self feeds type information back to the compiler to adaptively optimize code.

Some language implementations force the programmer to choose between interpretation and compilation, or between different modes of compilation. Placing this burden of choice upon the programmer's shoulders can only weaken his confidence in the reality of Self, and force him to consider the difference between the program as written and what really runs. Although Self employs two compilers and a myriad of optimizations, the programmer never chooses nor even knows which have been employed on his code.

Results on two medium-size cross-language benchmarks (Richards and DeltaBlue) suggest that Self runs 2 to 3 times faster than ParcPlace Smalltalk-80¹, 2.6 times faster than Sun CommonLisp 4.0TM using full optimization, and only 2.3 times slower than optimized C++ [HU94a]. Of course, these Smalltalk and Lisp implementations may not include aggressively-optimized compilers, but the C++ language has semantics that make many more concessions to efficiency over purity, simplicity, and safety.

Most implementations strive for efficiency and employ optimizations that show through to the programmer. Tail recursion elimination, for example, optimizes methods that iterate by calling themselves at their ends, but makes it impossible to show a meaningful stack trace. This destruction of information would show through to the user, who might need to see the missing stack frames in order to debug her program. So, Self does not optimize tail-recursion. Instead, endless loops are built in as a primitive operation. There are other optimizations left undone in Self, see [Hol94] for a list.

¹ Smalltalk-80TM is a trademark of ParcPlace Systems.

4.2 Responsiveness

Many systems impose long or unpredictable pauses upon their users. But, a pause in the middle of a user task such as the addition of a slot to an object could ruin the experience of a consistent world. Such pauses, by their very existence, alert users that some mischief is afoot. If the program (be it Self or any other language) were the reality, there would be no pauses upon changing it. Since we believe that such pauses can destroy the fragile illusion of reality, we have striven to reduce them in the Self implementation. In fact, pauses for compilation were a serious problem in early versions of Self, and inspired an effort to speed up the compiler. Now, any method can be changed in a second or two. Our ultimate goal is the elimination of perceptible pauses.

In [HU94b], Hölzle and Ungar measured the compilation pauses occurring during a 50-minute session of the SELF 3.0 user interface¹ [Cha95] [CUS95]. Their analysis grouped individual pauses into clusters that would be perceived as pauses by the users. The results indicated that there were few intrusive clustered pauses; on a current-generation workstation, only 13 pauses would exceed 0.4 seconds, and on a next-generation machine, none would exceed 0.3 seconds (see Figure 13).

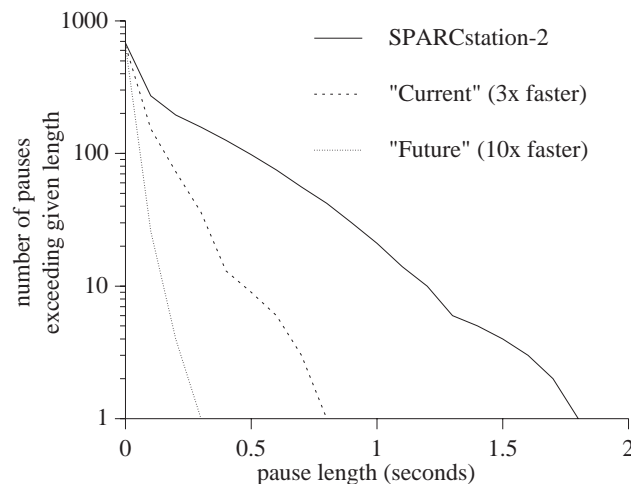


Figure 13. Compilation pauses (from [HU94b]).

Clustering pauses made an order-of-magnitude difference, and reporting individual pauses would have resulted in a distorted, overly optimistic characterization. The idea of pause clustering is one example of how our vision of providing a particular kind of experience to the programmer affects the standards that must be used to evaluate the system.

4.3 Malleability

By now, it should be clear to the reader that our philosophy of implementation as deception places additional burdens on the implementation, such as avoiding inconsistent behavior or unexplainable pauses. What may come as a surprise, though is that a requirement for malleability arises as a logical consequence of this unusual philosophy. For example, in both Smalltalk and Self, the if-then-else control structure is realized by sending a message “ifTrue:ifFalse:” to a boolean object (two arguments are included, a block to execute for true, and another for false). Each boolean simply implements this message with a message that executes the appropriate block; true’s ifTrue:ifFalse: method runs the true block and false’s runs the false block. If Self or Smalltalk objects are real, if they directly and faithfully execute, if the user is in control, then the user should be able to change these methods and observe the results. But in Smalltalk, extra performance is obtained by short-cir-

¹ The Self 4.0 user interface described in section 3 places more demands on the implementation and its pause behavior is not as good as these measurements suggest.

cutting this method in the implementation [GR83] and though he may change the method, the programmer cannot alter its behavior. This failure of malleability cannot help but raise disturbing questions and insecurities for the programmer. That is why Self does not short-circuit this or other such methods, even though providing such malleability without an efficiency loss extracts its cost from the implementation.¹

4.4 Encouraging Factoring and Extensible Control

Our non-traditional approach has led to techniques that provide a very traditional benefit: factoring is free. Suppose two methods in the same object contain several lines in common. By moving the common lines into a new method and sharing the method in both callers, the programmer can centralize them and make the program easier to change. Most implementations of object-oriented programming languages would slow down the program with an additional procedure call, providing a strong disincentive to the programmer for factoring small operations. In C++, for example the programmer must either ask explicitly for inlining (and give up virtual semantics) or pay the price of a many-statement overhead. Since Self relies on automatic inlining, no such price is paid. Consequently Self programmers feel much freer to factor programs, and the system is written in an unusually well-factored style. For example, a do-while loop is implemented with many levels of message passing before bottoming out in primitives. We believe that the performance characteristics of Self's implementation techniques encourage programmers to write programs that are easier to maintain.

In addition to free factoring, the Self implementation makes user-defined control structures as efficient as the built-in ones. Unlike for Smalltalk, there is no disincentive for the programmer to use a block; the implementation in most cases can inline it away. Since we believe that control abstraction is necessary for real data abstraction, making control abstraction free can help encourage programmers to write better programs. We strongly believe that in all languages with user-extensible control, such as Smalltalk and Beta, much benefit could be realized from adopting implementation techniques that put the user-defined control structures on the same footing as the system-defined ones.

4.5 Open Issues

Although the current Self system is in daily use by a number of people, several concerns remain about its implementation.

Overcustomization. The Self compiler creates another copy of a method for each kind of object that inherits it. Sometimes, the method is so trivial that the copies waste code space and compiler time for no good reason.

Memory Footprint. Because enough information is preserved to maintain source-level semantics, Self takes more space than other systems. Self 4.0 barely fits in a SPARCstation² with 32 Mb of real memory. Although we believe that programming time is more precious than memory cost, this resource requirement puts Self out of reach of many current users.

Real-Time Operation. Although much progress has been made in the elimination of perceptible pauses, the system still feels like it “warms up” when running the Self 4.0 environment. Hard real-time operation is not possible with today's system.

User Control. Within the Self group, a debate rages over how much control a user should have over the optimization process. On one hand, users want to be able to tell the system how much, where, and when to optimize. On the other hand, the effort to add this ability might be better spent doing a better job automatically, and giving users this control could distract them from their own tasks and destroy the fragile illusion of Self's reality. So far, we have kept the control over optimization entirely within the virtual machine, as an experiment in the philosophy of implementation as deception.

¹ In fact, at first the second author thought the cost would be too great. But Craig Chambers' compiler convinced him otherwise.

² SPARCstationTM is a trademark of SPARC International, licensed exclusively to Sun Microsystems Inc.

4.6 Summary

Confidence, comfort, satisfaction—what do these desires imply for implementation techniques? They rarely show up as topics in compiler papers, yet we believe that these goals have exerted a profound influence over Self’s implementors.

5 Conclusions

The Self language semantics, implementation, and user interface have been guided by the goal of generating a particular kind of experience for the user of the system. Programmers directly work in a uniform world of malleable objects in a very immediate and direct fashion. Self moves towards giving objects a kind of tangible reality.

The language helps give rise to this experience by its use of prototypes, which provide a copy-and-directly-modify mechanism for changes. Self’s treatment of slots with its symmetry for assignment and access reflect the deep connection between perception and manipulation at the sensory-motor level, while also enabling objects to reimplement state as behavior and reflecting an intuition about how objects are behaviorally perceived in the real world.

Self’s design departs significantly from other object-oriented languages by separating information needed to run the program from information about the programmer’s intentions. It distinguishes abstract types, used for the programmers understanding and reasoning about correctness, from concrete types, used to run and optimize the program. The former is left to the environment, the latter is left to the implementation. In our opinion, this approach avoids a number of undesirable consequences that often follow from attempts to integrate these two forms of information.

Finally, in designing Self, we have learned one lesson by making mistakes: examples can persuade the designer to include additional features which later turn out to produce incomprehensible behavior in unforeseen circumstances. This might be called “the language designer’s trap.” Minimalism, simplicity and consistency are better guides. They benefit every programmer, not just the ones who need advanced features. We suspect that many of today’s object-oriented languages could profit by dropping features.

The Self user interface and programming environment provides a direct object experience for creating objects and assembling applications by adhering to two principals: *Structure reification* makes the graphical containment structure and layout rules themselves appear as graphical objects and assures that any graphical object can be manipulated, displayed, connected to other objects, or customized. *Live editing* ensures that any object may be changed at any time without halting activities. A simple gesture takes the user from any graphical object to its programming-language-level counterpart, just as real-world objects can be taken apart. Consequently, programmers need not pore through long object libraries to find out where to start, but can simply find some graphical widget in the environment, like a button in a menu, that is similar to what they want, and proceed to dissect, inspect, modify and reassemble it. At no time must they retreat from a concrete object to some definition of an abstraction.

Two problems in the user interface interfere with achieving our goal. The existence of multiple views for a graphical object and its Self-level outliner dilutes the experience and these views should be merged. The duality between text and object goes deeper and does not readily present a solution.

The consistency and purity of the Self language together with the ubiquitous use of objects and live layout in the interface place enormous demands on the Self implementation, but more interestingly, the desire to create a particular kind of programming experience imposes its own unique requirements on the implementation. Thus, the implementation foregoes optimizations that cannot be hidden such as tail-recursion elimination. It also supports full source-level debugging, single stepping, and allows the programmer to change any method,

even basic ones such as addition and if-then-else, at any time. Since a long pause for compilation would alert the programmer to the existence of a lower level of reality, the implementation works hard to avoid such pauses. We view the implementation not as an interpreter of programs, but rather as a creator of the illusion that the Self objects are real.

Along the way Self's implementation techniques of adaptive recompilation and type-feedback achieve some traditionally-important but rarely achieved goals as well: the elimination of run-time penalty for factoring and for user-defined control structures. A programmer may chop up a method as finely as desired without slowing it down, and may introduce new abstractions that combine control and data without paying a run-time price. These characteristics encourage the create of programmers that are smaller and more malleable.

When all is said and done though, this paper can only suggest, tease, or maybe hint at what it is like to create with Self. In order to most fully appreciate the experience of interacting with a lively, responsive world of objects, effortlessly diving in to change them and create more, freely mixing data and programs, and only getting coffee when you are tired instead of when you change your program, you will have to obtain the Self 4.0 public release and try it out for yourself. May your journey be fruitful.

6 Acknowledgments

The past and present members of the Self group, highly talented individuals each, have made this body of work possible. The authors consider themselves fortunate to have known and worked with them. Sun Microsystems Laboratories has hosted the project for the past four years, for which we are deeply grateful. Special thanks to Mario Wolczko and Ole Lehrmann Madsen for comments on the draft.

7 References

- [APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. *Type Inference of Self: analysis of Objects with Dynamic and Multiple Inheritance*, in Proc. ECOOP '93, pp. 247-267. Kaiserslautern, Germany, July 1993.
- [Blas94] G. Blaschek. **Object-Oriented Programming with Prototypes**, Springer-Verlag, New York, Berlin 1994.
- [BD81] A. Borning and R. Duisberg. *Constraint-Based Tools for Building User Interfaces*, ACM Transactions on Graphics 5(4) pp. 345-374 (October 1981).
- [CUL89] Craig Chambers, David Ungar, and Elgin Lee. *An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. In OOPSLA '89 Conference Proceedings, pp. 49-70, New Orleans, LA, 1989. Published as SIGPLAN Notices 24(10), October, 1989.
- [CU90] Craig Chambers and David Ungar. *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs*. In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June, 1990.
- [CU91] Craig Chambers and David Ungar. *Making Pure Object-Oriented Languages Practical*. In OOPSLA '91 Conference Proceedings, pp. 1-15, Phoenix, AZ, October, 1991.
- [CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle, *Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self*. Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.
- [Cha92] Craig Chambers. **The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages**. Ph.D. Thesis, Computer Science Department, Stanford University, April, 1992.
- [CUS95] Bay-Wei Chang, David Ungar, and Randall B. Smith, *Getting Close to Objects*, in Burnett, M., Goldberg, A., and Lewis, T., editors, **Visual Object-Oriented Programming, Concepts and Environments**, pp. 185-198, Manning Publications, Greenwich, CT, 1995.

- [Cha95] Bay-Wei Chang, Seity: **Object-Focused Interaction in the Self User Interface**, Ph.D. dissertation, in preparation, Stanford University, 1995.
- [DS84] L. Peter Deutsch and Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 System*. In Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, 1984.
- [DMC92] C. Dony, J. Malenfant, and P. Cointe, *Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validation*, in Proc. OOPSLA '92, pp. 201-217.
- [GR83] Adele Goldberg and David Robson, **Smalltalk-80: The Language and Its Implementation**. Addison-Wesley, Reading, MA, 1983.
- [HCU91] Urs Hölzle, Craig Chambers, and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches*. In ECOOP' 91 Conference Proceedings, pp. 21-38, Geneva, Switzerland, July, 1991.
- [HCU92] Urs Hölzle, Craig Chambers, and David Ungar. *Debugging Optimized Code with Dynamic Deoptimization*, in Proc. ACM SIGPLAN '92 Conferences on Programming Language Design and Implementation, pp. 32-43, San Francisco, CA (June 1992).
- [Hol94] Urs Hölzle. **Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming**. Ph.D. Thesis, Stanford University, Computer Science Department, 1994.
- [HU94a] Urs Hölzle and David Ungar. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*. In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, Orlando, FL, June, 1994.
- [HU94b] Urs Hölzle and David Ungar. *A Third Generation Self Implementation: Reconciling Responsiveness with Performance*. In OOPSLA'94 Conference Proceedings, pp. 229-243, Portland, OR, October, 1994. Published as SIGPLAN Notices 29(10), October, 1994.
- [MMM90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen, *Strong Typing of Object-Oriented Languages Revisited*. In ECOOP/OOPSLA'90 Conference Proceedings, pp. 140-149, Ottawa, Canada, October, 1990.
- [MMN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard, **Object-Oriented Programming in the Beta Programming Language**, Addison-Wesley Publishing Co., Wokingham, England, 1993.
- [Mul95] Phillipe Mulet, **Réflexion & Langages à Prototypes**, Ph.D. Thesis in preparation, Ecole des Mines de Nantes, France, 1995.
- [MS95] John Maloney, and Randall B. Smith, *Directness and Liveness in the Morpheus User Interface Construction Environment*. In preparation.
- [Smi87] Randall B. Smith. *Experiences with the Alternate Reality Kit, an Example of the Tension Between Literalism and Magic*, in Proc. CHI + GI Conference, pp 61-67, Toronto, (April 1987).
- [SUC92] Randall B. Smith, David Ungar, and Bay-Wei Chang. *The Use Mention Perspective on Programming for the Interface*, In Brad A. Myers, **Languages for Developing User Interfaces**, Jones and Bartlett, Boston, MA, 1992. pp 79-89.
- [SLS94] R. B. Smith, M. Lentzner, W. Smith, A. Taivalsaari, and D. Ungar, *Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel)*, in Proc. OOPSLA '94, pp. 102-112 (October 1994). Also see the panel summary of the same title, in Addendum to the Proceedings of OOPSLA '94, pp. 48-53.
- [SMU95] Randall B. Smith, John Maloney, and David Ungar, *The Self-4.0 User Interface: Manifesting the System-wide Vision of Concreteness, Uniformity, and Flexibility*. To appear in Proc. OOPSLA '95.
- [Tai92] Antero Taivalsaari, **Kevo - a prototype-based object-oriented language based on concatenation and module operations**. University of Victoria Technical Report DCS-197-1R, Victoria, B.C., Canada, June 1992
- [Tai93] Antero Taivalsaari, **A critical view of inheritance and reusability in object-oriented programming**. Ph.D. dissertation, Jyväskylä Studies in Computer Science, Economics and Statistics 23, University of Jyväskylä, Finland, December 1993, 276 pages (ISBN 951-34-0161-8).
- [Tai93a] Antero Taivalsaari, *Concatenation-based object-oriented programming in Kevo*. Actes de la 2eme Conference sur la Representations Par Objets RPO'93 (La Grande Motte, France, June 17-18, 1993), Published by EC2, France, June 1993, pp.117-130

- [US87] David Ungar and Randall B. Smith, *Self: The Power of Simplicity*, Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL, October, 1987, pp. 227–242. A revised version appeared in the Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.
- [USCH92] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. *Object, Message, and Performance: How They Coexist in Self*. Computer, 25(10), pp. 53-64. (October 1992).
- [Wol95] Mario Wolczko, *Implementing a Class-based Language using Prototypes*, In preparation.

Java™: An Overview

James Gosling

Introduction by James Gosling

"Java™: An Overview" was written in early 1995, just as we were getting ready to release Java. By that time, Java had been used inside Sun for about 4 years by a small enthusiastic crew. I was looking forward to getting it out there so that others could enjoy it too. Being a pretty serious geek, I tended to think of it in pretty technical terms. This was a fairly serious problem since relatively few people could make sense of what I was babbling. My wife, Judy Borcz, is a very bright person but not a geek (she's a Wharton MBA type). When I would erupt in fits of techo-babble it didn't help her understanding at all. She pushed me very hard to write down, in terms that mattered to people like her, what was so cool about Java.

That put my head in a different place and got me thinking about Java from a different perspective. The title of the paper doesn't match the content of the paper — it isn't an overview of Java at all. A more appropriate title would have been "Java: Why Should Anyone Care?". It takes the perspective of someone looking in from the outside at the development of software products. It presents a series of vignettes in the life of a software product organization and explains how they could be impacted by the adoption of Java. The vignettes were chosen to be familiar to anyone who has been around the business.

The paper turned out to be quite important since it opened up the community of people we were talking to and made the technology more accessible to a wider audience. Many analysts and reporters got copies and used it as their first introduction to Java. It evolved over the years and became the standard Java whitepaper and is still up on the java.sun.com website.

REFERENCES:

Java Language Overview: <http://java.sun.com/docs/white/index.html>

Ken Arnold, James Gosling and David Holmes, The Java Programming Language Third Edition, Addison–Wesley, 2000.

James Gosling, Bill Joy, Guy Steele and Gilad Bracha, The Java Language Specification Second Edition, The Java Series, Addison–Wesley, 2000.

Greg Bollella and James Gosling, The Real–Time Specification for Java, IEEE Computer, 33(6), pp. 47–54, June 2000.

James Gosling and Henry McGilton, The Java Language Environment — A Whitepaper, Technical Report, Sun Microsystems, October 1995.

James Gosling, Extensions to Java for Numerical Computation, ACM 1998 Workshop on Java for High–Performance Network Computing, ACM Press, 1998.

Java: an Overview

James Gosling, February 1995

jag@sun.com

Introduction

JAVA[†] is a programming language and environment that was designed to solve a number of problems in modern programming practice. It started as a part of a larger project to develop advanced software for consumer electronics. These are small reliable portable distributed real-time embedded systems. When we started the project, we intended to use C++, but we encountered a number of problems. Initially these were just compiler technology problems, but as time passed we encountered a set of problems that were best solved by changing the language.


This document contains a lot of technical words and acronyms that may be unfamiliar. You may want to look at the glossary on page 8.

There is a companion paper to this, *WEBRUNNER an Overview*, that describes a very powerful application that exploits JAVA.

JAVA

JAVA: A simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, dynamic language.

One way to characterize a system is with a set of buzzwords. We use a standard set of them in describing JAVA. The rest of this section is an explanation of what we mean by those buzzwords and the problems that we were trying to solve.

 *Archimedes Inc. is a fictitious software company that produces software to teach about basic physics. This software is designed to interact with the user, providing not only text and illustrations in the manner of a traditional textbook, but also providing a set of software lab benches on which experiments can be set up and their behavior simulated. For example, the most basic one allows students to put together levers and pulleys and see how they act. A narrative of their trials and tribulations is used to provide examples of the concepts presented.*

Simple


We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. Most programmers working these days use C, and most doing object-oriented

[†] The internal development name for this project was "Oak". JAVA is the new official product name..

programming use C++. So even though we found that C++ was unsuitable, we tried to stick as close as possible to C++ in order to make the system more comprehensible.

JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions.

Paradoxically, we were able to simplify the programming task by making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. JAVA does automatic garbage collection —this not only makes the programming task easier, it also dramatically cuts down on bugs.

 *The folks at Archimedes wanted to spend their time thinking about levers and pulleys, but instead spent a lot of time on mundane programming tasks. Their central expertise was teaching, not programming. One of the most complicated of these programming tasks was figuring out where memory was being wasted across their 20K lines of code.*


Another aspect of simple is small. One of the goals of JAVA is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 30K bytes, adding the basic standard libraries and thread support (essentially a self-contained microkernel) brings it up to about 120K.

Object-Oriented

This is, unfortunately, one of the most overused buzzwords in the industry. But object-oriented design is still very powerful since it facilitates the clean definition of interfaces and makes it possible to provide reusable “software ICs”.

A simple definition of object oriented design is that it is a technique that focuses design on the data (=objects) and on the interfaces to it. To make an analogy with carpentry, an “object oriented” carpenter would be mostly concerned with the chair he was building, and secondarily with the tools used to make it; a “non-OO” carpenter would think primarily of his tools. This is also the mechanism for defining how modules “plug&play”.

The object-oriented facilities of JAVA are essentially those of C++, with extensions for more dynamic method resolution that came from Objective C.

 *The folks at Archimedes had lots of kinds of things in their simulation. Among them, ropes and elastic bands. In their initial C version of the product, they ended up with a pretty big system because they had to write separate software for describing ropes versus elastic bands. When they rewrote their application in an object oriented style, they found they could define one basic object that represented the common aspects of ropes and elastic bands, and then ropes and elastic bands were defined as variations (subclasses) of the basic type. When it came time to add chains, it was a*

snap because they could build on what had been written before, rather than writing a whole new object simulation.

Distributed

At one time networking was integrated into the language and runtime system and was (almost) transparent. Objects could be remote: when an application had a pointer to an object, that object could exist on the same machine, or some other machine on the network. Method invocations on remote objects were turned into RPCs (Remote Procedure Calls).

A distributed application looked very much like a non-distributed one. Both cases used an essentially similar programming model. The distributed case did, however, require that applications paid some attention to the consequences of network failures. The system dealt with much of it automatically, but some of it did need to be dealt with on a case-by-case basis.

Since then, that model has mostly disappeared: a concession to the pragmatics of living within the context of existing networks. Primarily the Internet. Consequently, JAVA now has a very extensive library of routines for easily coping with TCP/IP protocols like http and ftp. JAVA applications can open and access objects across the net via URLs with the same ease the programmers are used to accessing a local file system.

☞ *The folks at Archimedes initially built their stuff for CD ROM. But they had some ideas for interactive learning games that they'd like to try out for their next product. For example, they wanted to allow students on different computers to cooperate in building a machine to be simulated. But all the networking systems they'd seen were complicated and required esoteric software specialists. So they gave up.*

Robust

JAVA is intended for writing programs that need to be reliable in a variety of ways. There is a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and on eliminating situations which are error prone.

One of the advantages of a strongly typed language (like C++) is that it allows extensive compile-time checking so bugs can be found early. Unfortunately, C++ inherits a number of loopholes in this checking from C, which was relatively lax (the major issue is method/procedure declarations). In JAVA, we require declarations and do not support C style implicit declarations.

The linker understands the type system and repeats many of the type checks done by the compiler to guard against version mismatch problems.

As mentioned before, automatic garbage collection avoids storage allocation bugs.

The single biggest difference between JAVA and C/C++ is that JAVA has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Rather than having pointer arithmetic, JAVA has true arrays. This allows subscript checking to be performed. And it is not possible to turn an arbitrary integer into a pointer by casting.

- ☞ *The folks at Archimedes had their application basically working in C pretty quickly. But their schedule kept slipping because of all the small bugs that kept slipping through. They had lots of trouble with memory corruption, versions out-of-sync and interface mismatches. What they gained because C let them pull strange tricks in their code, they paid for in Quality Assurance time. They also had to reissue their software after the first release because of all the bugs that slipped through.*

While JAVA doesn't pretend to make the QA problem go away, it does make it significantly easier.

Very dynamic languages like Lisp, TCL and Smalltalk are often used for prototyping. One of the reasons for their success at this is that they are very robust: you don't have to worry about freeing or corrupting memory. Programmers can be relatively fearless about dealing with memory because they don't have to worry about it getting messed up. JAVA has this property and it has been found to be very liberating. Another reason given for these languages being good for prototyping is that they don't require you to pin down decisions early on. JAVA has exactly the opposite property: it forces you to make choices explicitly. Along with these choices come a lot of assistance: you can write method invocations and if you get something wrong, you get told about it early, without waiting until you're deep into executing the program. You can also get a lot of flexibility by using interfaces instead of classes.

Secure

JAVA is intended to be used in networked/distributed situations. Toward that end a lot of emphasis has been placed on security. JAVA enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption.

There is a strong interplay between "robust" and "secure". For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do have access to. This closes the door on most activities of viruses.

Not included in release 0.1 or 0.2.

There is a mechanism for defining approval seals for software modules and interface access. For example, it is possible for a system built on JAVA to say "only software with a certain seal of approval is allowed to be loaded" and it is possible for individual modules to say "only software with a certain seal of approval is allowed to access my interface". These approval seals cannot be forged since they are based on public-key encryption.


- ☞ *Someone wrote an interesting "patch" to the PC version of the Archimedes system. They posted this patch to one of the major bulletin boards. Since it was easily available and added some interesting features to the system, lots of people downloaded it. It hadn't been checked out by the folks at Archimedes, but it seemed to work. Until the next April first when thousands of folks discovered rude pictures popping up in their children's lessons. Needless to say, even though they were in no way responsible for the incident, the folks at Archimedes still had a lot of damage to control.*

Architecture Neutral

JAVA was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures. In order for an JAVA application to be able to execute anywhere on the network, the compiler generates an architecture neutral object file format — the compiled code is executable on many processors, given the presence of the JAVA runtime.

This is useful not only for networks but also for single system software distribution. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC and with the Apple Macintosh. With the PC market (through Windows/NT) diversifying into many CPU architectures, and Apple moving off the 68000 towards the PowerPC, this makes the production of software that runs on all platforms almost impossible. With JAVA, the same version of the application runs on all platforms.

The JAVA compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

 *Archimedes is a small company. They started out producing their software for the PC since that was the largest market. After a while, they were a large enough company that they could afford to do a port to the Macintosh, but it was a pretty big effort and didn't really pay off. They couldn't afford to port to the PowerPC Mac or MIPS NT machine. They couldn't "catch the new wave" as it was happening, and a competitor jumped in...*

Portable

Being architecture neutral is a big chunk of being portable, but there's more to it than just that. Unlike C and C++ there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behaviour of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32 bit IEEE 754 floating point number. Making these choices is feasible in this day and age because essentially all interesting CPUs share these characteristics.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows and the Mac[†].

The JAVA system itself is quite portable. The new compiler is written in JAVA and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.

Interpreted

The JAVA interpreter can execute JAVA bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental

[†] The Windows and Mac versions aren't complete yet.

& lightweight process, the development process can be much more rapid and exploratory.

As a part of the bytecode stream, more compile-time information is carried over and available at runtime. This is what the linker's type checks are based on, and what the RPC protocol derivation is based on. It also makes programs more amenable to debugging.

☞ *The programmers at Archimedes spent a lot of time waiting for programs to compile and link. They also spent a lot of time tracking down senseless bugs because some changed source files didn't get compiled (despite using a fancy "make" facility), which caused version mismatches; and they had to track down procedures that were declared inconsistently in various parts of their programs. Another couple of months lost in the schedule.*

High Performance

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The byte code can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those used to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader.

The byte code format was designed with this in mind, so the actual process of generating machine code is generally simple. Reasonably good code is produced: it does automatic register allocation and the compiler does some optimization when it produces the bytecode.

In interpreted code we're getting about 300,000 method calls per second on an SS10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

☞ *When Archimedes was starting up, they did a prototype in SmallTalk. This impressed the investors enough that they got funded, but it didn't really help them produce their product: in order to make their simulations fast enough and the system small enough, it had to be rewritten in C.*

Multithreaded

There are many things going on at the same time in the world around us. Multithreading is a way of building applications that are built out of multiple threads[†]. Unfortunately, writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C and C++ style.

JAVA has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm that was introduced by C.A.R.Hoare[‡]. By integrating these concepts into the language they become

[†] Threads are sometimes also called lightweight processes or execution contexts.

[‡] 1974. Hoare, C.A.R. *Monitors: An Operating System Structuring Concept*, Comm. ACM 17, 10:549-557 (October)

much more easy to use and robust. Much of the style of this integration came from Xerox's Cedar/Mesa system.

Other benefits are better interactive responsiveness and realtime behaviour. This is limited, however, by the underlying platform: stand-alone JAVA runtime environments have good realtime behaviour. Running on top of other systems like Unix, Windows, the Mac or NT limits the realtime responsiveness to that of the underlying system.

- ☞ *Lots of things were going on at once in their simulations. Ropes were being pulled, wheels were turning, levers were rocking, and input from the user was being tracked. Because they had to write this all in a single threaded form, all the things that happen at the same time, even though they had nothing to do with each other, had to be manually intermixed. Using an "event loop" made things a little cleaner, but it was still a mess. The system became fragile and hard to understand.*
- ☞ *They were pulling in data from all over the net. But originally they were doing it one chunk at a time. This serialized network communication was very slow. When they converted to a multithreaded style, it was trivial to overlap all of their network communication.*

Dynamic

In a number of ways, JAVA is a more dynamic language than C or C++. It was designed to adapt to an evolving environment.

For example, one of the big problems with using C++ in a production environment is a side-effect of the way that it is always implemented. If company A produces a class library (a library of plug&play components) and company B buys it and uses it in their product, then if A changes its library and distributes a new release then B will almost certainly have to recompile and redistribute their software. In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application vendor) then if A distributes an upgrade to its libraries then all of the users software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly.


- ☞ *Archimedes built their product using the object oriented graphics library from 3DPC Inc. 3DPC released a new version of the graphics library which several computer manufacturers bundled with their new machines. Customers of Archimedes that bought these new machines discovered to their dismay that their old software no longer worked. [In real life, this only happens on Unix systems. In the PC world, 3DPC would never have released such a library: their ability to change their product and use C++'s object oriented features is severely hindered]*

By making these interconnections between modules later, JAVA completely avoids these problems and makes the use of the OO paradigm much more straightforward. Libraries can freely add new methods and instance variables without any effect on their clients.

JAVA understands the concept of an *interface*. An interface is a concept borrowed from Objective C and is similar to a class. An interface is simply a specification of a set of methods that an object responds to. It does not include any instance variables or implementation. Interfaces can be multiply-inherited (unlike classes) and they can be used in a more flexible way than the usual rigid class inheritance structure.

Classes have a runtime representation: there is a class named *Class*, instances of which contain runtime class definitions. From an object you can find out what class it belongs to. If, in a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out. In JAVA, finding out based on the runtime type information is straightforward. For example, type conversions (casts) are checked at runtime in JAVA, whereas in C and C++, the compiler just trusts that you're doing the right thing.

It is also possible to lookup the definition of a class given a string containing its name. This means that you can compute a data type name and have it trivially dynamically linked into the running system.

 *To expand their revenue stream, the folks at Archimedes wanted to architect their product so that new aftermarket plug-in modules could be added to extend the system. This was possible on the PC, but just barely. They had to hire a couple of new programmers because it was so complicated. This also added terrible problems when debugging.*

Glossary

This paper is filled with all sorts of words and TLAs that may be hard to decipher. Here are the definitions of a few:

API	Application Programmer Interface. The specification of how a programming writing an application accesses the facilities of some object. Interfaces can be specified in JAVA and C++ using classes. JAVA also has a special interface syntax that allows interfaces that are more flexible than classes.
FTP	The basic internet File Transfer Protocol. It enables the fetching and storing of files between hosts on the internet. It is based on TCP/IP.
HTML	HyperText Markup Language. This is a file format, based on SGML, for hypertext documents on the internet. It is very simple and allows for the imbedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are imbedded using URLs.
HTTP	Hypertext Transfer Protocol. This is the internet protocol used to fetch hypertext objects from remote hosts. It is based on TCP/IP.

Internet	An enormous network consisting of literally millions of hosts from many organizations and countries around the world. It is physically put together from many smaller networks and is held together by a common set of protocols.
IP	Internet Protocol. The basic protocol of the internet. It enables the unreliable delivery of individual packets from one host to another. It makes no guarantees about whether or not the packet will be delivered, how long it will take, or if multiple packets will arrive in the order they were sent. Protocols built on top of this add the notions of connection and reliability.
JPEG	Joint Photographic Experts Group. An image file compression standard established by this group. It achieves tremendous compression at the cost of introducing distortions into the image which are almost always imperceptible.
Mosaic	A program that provides a simple GUI that enables easy access to the data stored on the internet. These may be simple files, or hypertext documents.
RPC	Remote Procedure Call. Executing what looks like a normal procedure call (or method invocation) by sending network packets to some remote host.
SGML	Standardized, Generalized Markup Language. An ISO/ANSI/ECMA standard that specifies a way to annotate text documents with information about types of sections of a document. For example "this is a paragraph" or "this is a title".
TCP/IP	Transmission Control Protocol based on IP. This is an internet protocol that provides for the reliable delivery of streams of data from one host to another.
TLA	Three Letter Acronym.
URL	Uniform Resource Locator. A standard for writing a textual reference to an arbitrary piece of data in the WWW. A URL looks like " <i>protocol://host/localinfo</i> " where <i>protocol</i> specifies a protocol to use to fetch the object (like HTTP or FTP), <i>host</i> specifies the internet name of the host on which to find it, and <i>localinfo</i> is a string (often a file name) passed to the protocol handler on the remote host.
WWW	World Wide Web. The web of systems and the data in them that is the internet.

The Design, Analysis, and Implementation of a Media Server Architecture

James G. Hanko

Introduction by James G. Hanko

In the summer of 1994, Sun's major competitors were announcing products for the emerging video on demand (VOD) market. It was determined that Sun needed a strong offering in this market to stay competitive. Since VOD would be expensive to deploy, the supplier that could reliably support a large number of customers at the lowest cost per video stream would have the greatest chance for success. After surveying the technology available at that time, we found that there were no existing systems or methodologies that would meet this goal. This technical report describes the technology that was developed at Sun to address this need.

Video on demand servers must store large amounts of video content and be able to deliver individual streams to each home, independent of the demands of other users. Rotating disk storage was the only technology available that could hold the vast amounts of data necessary to support a VOD system. However, disks have the following troubling characteristics: limited reliability (i.e., relatively high failure rate), varying performance (depending on which part of the disk is used), and non-determinism in response time to requests (due to retries and recalibration steps). Therefore, it is difficult to construct a system that can reliably deliver a large number of streams at the lowest cost per stream.

This paper details the mechanisms that were developed to address the reliability, performance, and non-determinism issues with disk subsystems in a way that reliably delivered as many streams of video as the disks were physically capable of supporting. The technology was delivered as software that controlled an off-the-shelf Sun computer and storage subsystem, and no unique hardware was required.

Sun formed the Interactive Services Group (ISG) business unit to develop a product, the Sun Media Center (SMC), based on the technology described in this paper. As a result of this technology, the product was able to meet the reliability and cost-per-stream goals.

The Design, Analysis, and Implementation of a Media Server Architecture

James G. Hanko

Sun Microsystems Laboratories

Abstract

The primary goal of a media server is to reliably deliver continuous data streams to as many clients as feasible given the available hardware resources. This paper describes the design and implementation of the Sun Media Server, and analyzes the techniques used to meet this goal.

1 Background

There is growing commercial interest in building media servers out of components originally developed for computer systems. *Media* in this context means continuous streams of time-critical data, such as digital audio and video, and a *stream* denotes the active process of reading data from the disks and delivering it to a client over a network. An effective media server can today be constructed from commodity computer systems and disk arrays to serve many clients over a high-bandwidth digital network.

A server for continuous media streams primarily acts as an engine for reading media data from disk storage and supplying it to a network connection at the proper delivery rate. In order to be successful, a server must be able to reliably supply as many clients as feasible given the hardware resources. It must also be able to accurately predict whether a particular load of clients is sustainable without disruption to the media streams.

In building a system to serve media clients from a set of disks, the parameters that constrain the number of active users are:

- the number of active streams supported by each disk
- the distribution of stream data over the disks

The number of streams supported by each disk are further constrained by:

- the amount of data transferred in each disk read
- the disk bandwidth during transfers
- the seek time between transfers

It must also be recognized that the response of a modern disk drive (as seen by a system) is non-deterministic. This is due to processing within the disk subsystem such as thermal recalibration, disk retries, error correction using error correcting codes, disk block error remapping, etc. These effects are not under direct system control and typically do not affect the data transferred, but instead affect the time period over which the transfers occur. A successful server design must be able to absorb a substantial amount of variability in disk response without disrupting the media streams.

The general problem of managing shared computing resources for time-critical applications in the face of non-determinism has been studied, and effective approaches have been developed [2], [10]. These methods generally rely on algorithms to predict or detect resource overload and mechanisms to bring the system back into equilibrium. However, in the case of a dedicated media server, aspects of the problem allow simpler methods to be used. Specifically, there is only one time-critical task that the system must perform (i.e., delivering media streams)

and the computational load in doing so is small. Therefore, the only resources that must be carefully managed are disk and network bandwidth. In addition, the load imposed on the system by a stream can be accurately predicted from the characteristics of the media data (e.g., bit rate).

This paper describes the Sun Media Server software design, analyzes the design trade-offs, and describes the implementation. The primary emphasis is on the disk scheduling and the admission control subsystems, which are collectively called the Sun Media Scheduler.

2 Assumptions

In general, there is little locality between disk access locations for different streams on a media server. This is because each data file is so large (multiple GBytes each), and it is played out over a long time (e.g., two hours). Therefore, the probability of two clients accessing the same data within a short time period is very small.

The average case disk seek times quoted by disk manufacturers are computed by dividing the sum of the time taken for all possible seeks by the number of possible seeks. The average distance across all possible seeks is 1/3 of the number of cylinders, although the time for a seek of this distance does not necessarily correspond to the quoted average seek time [3]. The worst case seek time comes from a seek across the full disk.

Disk seeks have four major components:

- 1) acceleration
- 2) constant velocity motion
- 3) deceleration
- 4) head settle time

With suitably long seeks (as in media servers) acceleration, deceleration, and head settle time can be thought of as a fixed overhead¹. Because of this, the following relationship holds:

$$S(X_1) + S(X_2) + \dots + S(X_n) \leq n \cdot S\left(\frac{1}{n} \cdot \sum_{i=1}^n |X_i|\right)$$

1. Very short seeks would take less time than this because full acceleration is not required.

where $S(x)$ is the time to seek a fraction x of the disk, and $X_1 \dots X_n$ represent a series of disk seek fractions. That is, the worst case seek time for a group of n seeks covering some total distance d occurs when each individual seek is of equal length, i.e., d/n .

A track near the outer edge of a disk is longer (i.e., has a larger circumference) than one near the inner edge, so there is potentially a greater storage capacity in the outer tracks. Because of this, modern disks are divided into several regions, with the regions at the outer edge of the disk containing more sectors than those at the inner edge. Therefore, with constant rotational velocity, the bandwidth obtainable from the outer regions is larger than from the inner ones. The worst case bandwidth occurs in the innermost region.

Finally, disk overhead is minimized by transfers of larger amounts of data at a time. This is true because larger transfers mean that a higher proportion of the time is spent transferring data, and a lower proportion is spent doing seeks. Therefore, a server using longer transfers will be able to support more clients, all else being equal.

3 Approach

On modern disks, the observed variances between worst case and average case for both seek time and disk transfer bandwidth described above can approach 2-to-1. Therefore, any scheme to maximize the number of supported streams must ensure that, over some small interval of time, the average case bandwidth and seek times are always achieved. If the worst case bandwidth or seek times are allowed to persist over an unbounded period, then the number of supported streams must be constrained by these worst case numbers. In other words, unless active measures are taken to limit worst case behavior, it can persist for an arbitrary amount of time. Then any assumption of better than worst case response will result in the disruption of some or all streams. If, however, bandwidth and seek times can be guaranteed to achieve average case numbers during bounded (short) intervals,

these parameters can be used to calculate the number of supported streams (which would be larger than with the worst case numbers).

Even if the stream admission policy used by a server is stochastic instead of deterministic, guaranteeing average case performance can be used to increase the number of supported streams. This is because the guarantee will minimize the performance variance and, thereby, increase the confidence factor for supporting larger numbers of streams.

The design of the Sun Media Scheduler can guarantee approximately average case bandwidth and seek times over a bounded (short) interval. This is achieved through the disk layout and the stream admission policy.

3.1 Disk Layout

The layout of the data on the disks is critical for server performance. In the Sun Media Server, the concept is to divide each disk into Z zones containing equal numbers of disk blocks (in the current implementation, Z is set to two). Each zone is a contiguous group of disk blocks and roughly corresponds to one or more of the above-described regions created by the disk manufacturer. For example, if $Z=2$, then one zone would contain that half of the disk blocks in the faster regions of the disk, and the other would contain the half of the disk blocks that are in the slower regions.

Each *title* (movie, video clip, etc.) is striped across D disks and Z zones. Data for a title is organized into groups of contiguous disk blocks called *extents*. The extents containing the data for the titles are laid out such that extent i of each title would be placed² in zone:

$$\left(i + \frac{i}{D}\right) \text{ modulo } Z$$

2. Note this description is for convenience only. Excessive fragmentation would occur on the last disk because all movies start on the same disk and end at $(I \bmod D)$, where I is a title's length in extents. So, on average only 1 of D movies would place their last extent on disk $(D-1)$. A similar problem would occur if all movies started in the same disk zone. To avoid this, starting extents for different movies are distributed over the set of disks and zones.

of disk:

$$i \text{ modulo } D$$

This scheme generally alternates zones on adjacent disks, with a possible repeat of zones in the transition from the last to first disk. For example, with two zones and five disks the layout $\langle \text{disk}, \text{zone} \rangle$ would be:

$$\langle 0,0 \rangle \langle 1,1 \rangle \langle 2,0 \rangle \langle 3,1 \rangle \langle 4,0 \rangle \langle 0,1 \rangle \langle 1,0 \rangle \dots$$

The server utilizes the concept of a *scheduling interval*, a configuration-specific period of time in which data for all clients is read from the disks. The media data for each title is saved on the disk so that the data needed for a stream in each scheduling interval is stored in exactly one extent. This means that titles of different bit rates (e.g., with different compression methods or parameters) will have different sized extents, but that all the extents will correspond to the same play time interval on their respective titles.

The server is designed using the so-called *push* model in which the pacing of the data to the client is completely under the control of the server. That is, there is no rate-control feedback loop between client and server, and it is the responsibility of the client to follow the server's rate. There are two major reasons for this decision. First, the server is designed to support broadcast and near-video-on-demand (a multicast service), where one stream goes to many clients, as well as a video-on-demand model where each stream only goes to one client (i.e., point-to-point). In a broadcast or multicast environment, there is no meaningful way for the server to combine rate feedback from multiple clients. Since it is necessary for clients to function in this environment, creating a separate mechanism for point-to-point communication would just introduce unnecessary complexity.

A second reason for using the push model is to maximize the capacity of the server, while minimizing resource usage. If different streams proceeded at different rates, there would be no way to prevent them from drifting to a point where they would all be accessing the same disk at the same time. Clearly one disk could not simultaneously supply as many clients as all

disks together. Therefore in order to deal with this, it would be necessary to buffer a full cycle of data from each disk for each stream. That would require an increase in buffer capacity by more than an order of magnitude. In addition, since all clients could request data faster than the nominal rate of the stream, the server capacity would have to be derated by the worst-case (or at least the expected-case) variance in rates.

By using the push model, the Sun Media Server can ensure that no stream drift occurs. The scheduler uses the concept of *slots* (described below) to organize streams into groups that share access to the disks. When two streams are placed together in a slot, the use of the push model ensures that they will never move out of phase with each other, and they will never need to access a disk that is being used by another slot. This minimizes the buffer requirements because a *just-in-time* strategy can be used. In addition, it allows the stream admission algorithm to predict disk loading into the indefinite future both precisely and easily.

3.2 Stream Admission and Scheduling

The Sun Media Scheduler organizes sets of streams into *slots* in order to manage the disk bandwidth and seek times. Let B represent the total bandwidth that can be accommodated by one disk in the server, and Z be the number of zones used. Streams can be added to a slot only so long as all the streams in the slot use a total bandwidth less than or equal to B . The number of slots is equal to the number of disks in the system. Streams in one slot always access the same disk in each time period. Within a slot, streams are divided into Z groups, where all members of a group always access the same disk zone. The total bandwidth used by streams in each group is kept approximately equal. In other words, each zone of every disk is accessed every time period by a group of streams using bandwidth of approximately:

$$\frac{B}{Z}$$

Streams in each slot proceed together from disk to disk in lock-step. When the slot makes the transition from one disk to another, the disk zone accessible to each stream changes in accordance

with the layout policy. This maintains the balance between zone accesses, while allowing every stream to access all areas of the disks.

Within each slot, the read requests for a disk are ordered by disk block. The scheduler seeks to the end of the ordered set of requests nearest the current head position, then proceeds through the set to the nearest subsequent position. This is, effectively, the well-known *elevator* or *Scan* algorithm performed on each slot. Such a configuration of groups using the Scan algorithm has been called a *group sweeping scheme* [1].

At any given time, only one slot may read data from a given disk. Each disk is assigned to one slot in every time period. All of the streams in the slot read their required data from that disk and then send it out over the network in the next time period. This *just-in-time* approach minimizes the buffer occupancy (and therefore total memory requirements) compared to traditional scheduling schemes [1], [8].

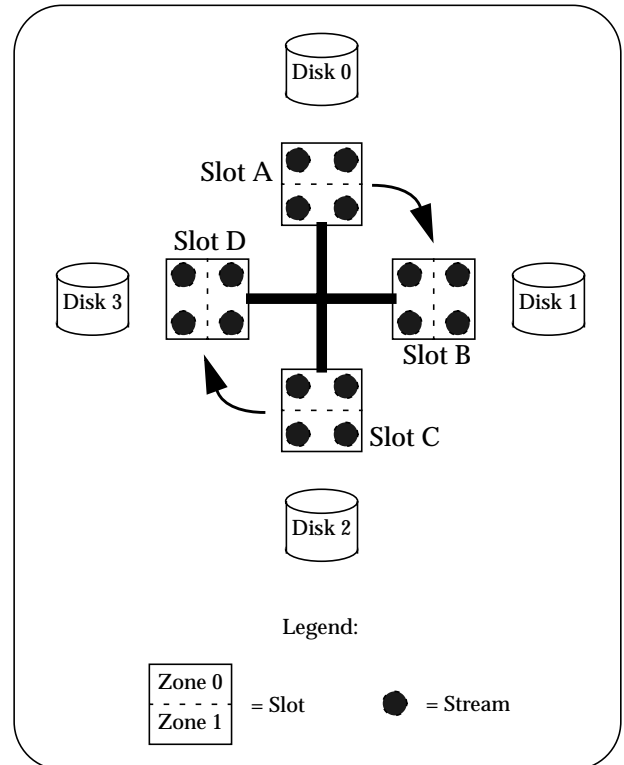


FIGURE 1. Scheduler Operation

The operation of the Sun Media Scheduler can be visualized as a carousel in which each slot is

briefly attached to a disk in order to access information, and then moves on. Figure 1 shows how a system with four disks and four slots would operate. In the position shown, Slot A is attached to Disk 0. While it is attached, it will read data for all four streams in slot A from both zones of the disk (two streams are using each zone). Similarly, Slot B is reading from Disk 1, etc. When the time period is up, Slot A will advance to Disk 1, Slot B to Disk 2, etc., and the data read in the first time period will be sent to the clients via the network. When Slot A moves to Disk 1, the zones that the individual streams in Slot A access on the disk adjust to match the media layout policy; this transition is the same for all titles.

Slot	Stream	Disk, Zone access by time period						
		t0	t1	t2	t3	t4	t5	t6
0	0	0,0	1,1	2,0	0,1	1,0	2,1	0,0
	1	0,0	1,1	2,0	0,1	1,0	2,1	0,0
	2	0,1	1,0	2,1	0,0	1,1	2,0	0,1
	3	0,1	1,0	2,1	0,0	1,1	2,0	0,1
1	4	1,0	2,1	0,0	1,1	2,0	0,1	1,0
	5	1,0	2,1	0,0	1,1	2,0	0,1	1,0
	6	1,1	2,0	0,1	1,0	2,1	0,0	1,1
	7	1,1	2,0	0,1	1,0	2,1	0,0	1,1
2	8	2,1	0,0	1,1	2,0	0,1	1,0	2,1
	9	2,1	0,0	1,1	2,0	0,1	1,0	2,1
	10	2,0	0,1	1,0	2,1	0,0	1,1	2,0
	11	2,0	0,1	1,0	2,1	0,0	1,1	2,0

Table 1: Slot Accesses to Disks and Zones

An alternative view of the Sun Media Scheduler's operation is shown in Table 1. In this example, there are three disks, two zones, and twelve streams shown over seven time periods (scheduling intervals). As can be seen from the table, the disks are accessed in all zones every time period. This activity is balanced among the zones to achieve the average case disk bandwidth. Note that every stream has access to every zone of all the disks, but at differing times. The access pattern of period t6 is a repeat of period t0. In general, the access patterns in the

Sun Media Scheduler repeat after $disks \cdot zones$ scheduling intervals.

The scheduler is configured to use the Sun Data Pump [5], [9] to deliver the data streams to the network interface and to regulate their rates. In order to avoid the problem of having separate clocks in the scheduler and in the Pump, potentially moving at differing rates, the scheduler uses a feedback mechanism from the Data Pump to control the advance of the scheduler time periods. This means that the scheduler's interval is derived from timing information provided by the pump, ensuring that they advance at identical rates over the long-term.

When a new stream is requested, the admission policy attempts to place it in the slot that will next be assigned the disk containing the first block of the stream. However, if that slot is full, the admission policy scans backwards for a slot that will soon have access to the proper disk and zone and which has sufficient bandwidth for the stream.

Once a new stream has been admitted, the scheduler ensures that it begins in phase with the other streams in the system. This is accomplished by delaying the sending of its first block of data until the last block of data for the previous time period has been sent for a majority of the active streams. This minimizes the spread between the streams in the server and also minimizes buffer usage and eliminates start-up jerkiness for new streams.

The bandwidth allocation for each slot is based on the amount of I/O that each disk can perform in one scheduling period, de-rated by the amount of time lost to seeks.

3.3 Work-Ahead

In order to deal with the inherent non-determinism of disk subsystems, a feature called *work-ahead* has been designed into the Sun Media Scheduler. When the disk accesses for one slot in one time period have been completed, the disk is made available to begin work early for the next slot to access the disk. Work-ahead may occur on a disk independent of the other disks' activities.

The admission control subsystem ensures that the full system load does not equal or exceed the total capacity of the disk subsystem. Because of this, there is a long-term rate mismatch such that, over time, the disk subsystem can always supply more data than needed. The work-ahead feature exploits this mismatch, so that in steady state the system is approximately one full time period ahead of where it needs to be to supply the clients. That is, it builds up *slack time* in the disk schedule.

Because of the work-ahead strategy, the admission subsystem can deliberately create short-term overloads, for example by putting too many streams in one slot, as long as it maintains the long term load below the disk capacity. This can be used to minimize the time a new client must wait to get the stream data. In addition, the slack can be exploited to maximize the number of clients overall.

For example, if the disk capacity is such that a disk can serve 4.5 clients each time period, the admission algorithm can alternate slots with four streams and slots with five streams. Although the slots with five streams cannot read the disk in one time period, each pair of slots completes within two time periods. Because of the slack built by work-ahead, the clients see no disruption in service.

Work-ahead also gives the server a level of immunity from the non-determinism of the disk subsystem. Typical disruptions such as sector retries, bad block forwarding, etc. will delay a disk transfer from 20 to 100 milliseconds. Delays introduced by thermal recalibration typically range from 50 to 300 milliseconds³. By setting the scheduling interval of the server to 1/2 second (500 milliseconds), the server is capable of absorbing substantial disruptions without affecting the client streams by consuming the built-up slack. As long as the disruptions do not persist (as they do in a failing disk), server will quickly rebuild its slack and suffer no service disruption.

3. Disks specifically designed for media servers have thermal recalibration times near the low end of the range.

The work-ahead feature has been constrained in the Sun Media Scheduler to allow accesses only to the next disk in the sequence, because each additional step of work-ahead can increase the start-up time for new streams. Once a slot has finished using a disk and has passed it on, it is no longer possible for a new stream in that slot to access the disk. Therefore, the stream must be admitted to a slot later in the cycle and suffer an increased start-up delay. In addition, each step of work ahead creates a requirement for more memory buffers to hold the accessed data until it is needed.

Table 2 shows a timeline of work-ahead (W), read (R), and send (S) operations for one slot in a hypothetical system with twelve data disks, D0-D11 over 15 time periods t0-t14. The table shows that the slot sends the data that it read in time t1 from disk D0 out to the network in time period t2. Each slot will normally contain several active media streams. The order of reads for streams in the slot is determined by the elevator algorithm.

Disk	Time period														
	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14
D0	W	R	S										W	R	S
D1		W	R	S									W	R	
D2			W	R	S									W	
D3				W	R	S									
D4					W	R	S								
D5						W	R	S							
D6							W	R	S						
D7								W	R	S					
D8									W	R	S				
D9										W	R	S			
D10	S										W	R	S		
D11	R	S										W	R	S	

Legend:

- W - Work-ahead read
- R - Read disk
- S - Send over network

Table 2: Single Slot Timeline In Normal Operation

Table 2 represents the activity of only one slot in the scheduler. Other slots in the system follow a similar pattern, although each subsequent slot has its time line shifted one position to the right of its immediate predecessor. For example if Table 2 represents slot 0, then slot 1 will be assigned disk D0 in time period t2. Note that work-ahead is permitted only if the slot assigned the disk for reading has completed all I/O it requires from the disk. Note also that a given slot may be simultaneously accessing one disk as its assigned disk and another for work-ahead.

3.4 Buffering

The simplest method of buffer management would be to have two buffers per stream. One would play out while the other is filling from the disk. For work-ahead to be effective, a third buffer would be needed per stream.

A better but more complex method requires approximately one buffer per stream plus one buffer per disk at a minimum. In this case, the buffer is made up of a number of *chunks* that are individually managed and that are kept in a common pool until needed. At the beginning of every period, each active stream in the system has a full buffer of data (read in the previous period), and the pool contains enough free chunks so that the first data needed in that time period from each disk can be read. As the time period progresses, each stream will free up the chunks containing data that have been sent, and these are used for subsequent disk transfers⁴. In a worst-case disk access scenario, more chunks will be ready in time for subsequent disk transfers. In the case where the disk performance is better than expected, the disk process can wait until enough chunks are available. Again, for work-ahead to operate effectively, additional chunks, up to the equivalent of one additional buffer, should be available per stream.

The Sun Media Scheduler uses the second method to handle buffer fragmentation due to varying bit rates, and to provide additional flexibility to cover disk timing transients. It allocates enough buffer chunks to have

4. Extra chunks are required if the number of chunks per buffer is fewer than the number of streams per slot.

approximately 2.75 full buffers per stream, which creates approximately one full time period of slack time. Each chunk is 128 KBytes.

4 Disk Failure Recovery

A commercially viable media server must have a high reliability factor. It would be unacceptable to most users if the system were unavailable often or for extended periods. In addition, any reduction in capacity (e.g., due to “graceful degradation” in a failure case) will have the effect of causing total failure to some subset of clients. This section looks at the disk error recovery requirements and outlines the strategy for achieving them in the Sun Media Scheduler.

4.1 MTBF Calculation

Current generation disks have MTBF of 500000 or more hours. In operational mode (after infant mortality and before end-of-life) most components follow an exponential error distribution. If this is assumed for disks, we have:

$$Prob(X = x) = \begin{cases} \lambda \cdot e^{(-\lambda \cdot x)} & x > 0 \\ 0 & x \leq 0 \end{cases}$$

and:

$$Prob(X \leq x) = \begin{cases} 1 - e^{(-\lambda \cdot x)} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

and the mean (i.e., MTBF) is $1/\lambda$ [4].

Therefore, a MTBF of 500000 makes $\lambda = 1/500000$.

The probability that a single disk will survive one full time period x is:

$$1 - (1 - e^{(-\lambda \cdot x)}) = e^{(-\lambda \cdot x)}$$

and the probability that all of n independent disks survive in time period x is:

$$(e^{(-\lambda \cdot x)})^n = e^{(n \cdot (-\lambda \cdot x))} = e^{(-(n \cdot \lambda) \cdot x)}$$

This corresponds to a second exponential distribution with MTBF of $(n \cdot \lambda)$.

So, for example, the MTBF of 60 disks is $500000/60 = 8333$ hours or 347 days.

Note that the probability of 60 disks surviving one hour is the same as the probability of one disk surviving 60 hours. This is because the exponential distribution has no “memory”.

If it is assumed that repair takes less than eight hours, the chance of a second failure during that time is less than one tenth of one percent. Therefore, single point failure tolerance is sufficient for the class of media server addressed by this design.

4.2 Recovery Method

In order for a media server to be able to continue delivering data in the face of the failure of any single disk in the system, some redundant information must be available from which to reconstruct the data on the failed disk. Clearly, full mirroring, where all data is replicated on a redundant set of disks, can accomplish this with no additional buffer memory and with minimal impact on the media scheduler. However, mirroring doubles the cost of the disk subsystem, which is already a major component of the server’s cost.

The RAID technique of building a *parity* disk with the XOR (exclusive-OR) of all the data on the other disks can be used to reconstruct any failed disk. It does, however, require additional storage and can require up to twice as many disk accesses (i.e., cut usable bandwidth in half). In order to reconstruct a block from a failed disk, one must first read data from the parity disk and all other data disks involved in the parity set. Then either this information is retained until it is needed (requiring more memory), or it is re-read when needed (increasing disk accesses).

The worst case extra buffering requirement occurs when the failed disk is the first one accessed in each cycle. Then buffering for the entire cycle must be available because every disk must be read before the resulting data can be reconstructed and sent. Therefore the buffering requirement is $(streams \cdot (disks + 1))$. The extra one is needed to read in the next parity information while playing out the reconstructed data.

Using parity across all disks is not, in general, feasible because of the explosion of memory requirements and the processing power necessary to perform parity reconstruction over a large data set. However, if the total number of disks is divided into separate *clusters*, and the goal is to recover from a single disk failure in any cluster, the buffering requirements are reduced substantially.

For example, assume 30 disks and 150 streams using 256KB buffers, then 1.2GB of memory would be required $(150 \cdot 31 \cdot 256KB)$ if the parity data encompassed all disks. However, if five clusters of six disks each were separately managed using the techniques outlined earlier, then the additional buffering for dealing with a single cluster failure is 55MB (i.e., $30 \cdot 7 \cdot 256KB$). This is because streams only need additional buffers during the time they pass through the failed cluster. Afterwards, the extra buffers are available for the next streams as they pass through. In this case, there are at most 30 (i.e., $150/5$) streams in the failed cluster at any time.

In addition to the increased storage capacity, there needs to be additional processor power to perform the XOR operation. It is interesting to note that every byte in every stream in the failed cluster will be XOR’ed once. If we assume 4Mbit/sec streams, then $(30 \cdot 4094304)/8$ bytes must be processed. This is under 16 MBytes/second and is easily performed by a single processor. Reducing the size of the cluster would make this even easier.

An important assumption of these calculations is that the parity data is kept on a per-stream basis instead of a per-disk-block basis. The bandwidth management scheme being used scatters the disk blocks of a stream around each disk (in zones). By structuring the parity on a per-stream basis, only data that is already needed for the stream is combined with the parity information to reconstruct the lost data. In essence, data from the non-failed disks in the parity set will be used both to reconstruct the failed data and as stream data on its own. In order to avoid worst-case bandwidth on parity reads, the parity data must be zoned similarly to the stream data. A simple way to do this is to layout the redundant blocks

in the same zone as the blocks of one of the data disks (e.g., the first disk of the cluster).

These calculations further assume that the failed disk is replaced and the data reconstructed off-line or with available bandwidth I/O. For example, the missing data could be read from tapes or optical juke-box.

Upon the failure of a disk, there would be a single “glitch” in all the affected streams while the system transitions from just-in-time buffer management to the pre-reading required for recovery. After that, the streams would play out at full rate. The media server is able to recover from any single disk failure without diminishing the active stream capacity. That is, the system is able to support as many users with one disk failure as with none. However, the server is not able to operate with two or more disk failures.

In the Sun Media Server, data for each title is striped across all the disks. However, the error recovery information is grouped into *redundancy sets* (the clusters described above). For example, if there were five disks in each redundancy set, four would be used as data and one used for parity on the four (i.e., 4 data + 1 parity). So, if there were 30 data disks, they would be organized with disks 0-3 containing data and disk 4 containing the parity for 0-3. The media data would continue on disks 5-8 with parity information for 5-8 on disk 9, and so on.

This is similar to, but subtly different from, traditional RAID techniques. Some RAID levels allow greater performance to be obtained during non-failure mode. However, since the server must be able to maintain the same number of streams during failure, it can't use the additional bandwidth available with those RAID schemes. In addition, intermixing media data and parity information on a disk would make the scheduler's slotting scheme much more problematic. Finally, the organization of parity data by title or file, even when the data blocks are spread around the disks, is a significant departure from traditional RAID techniques.

4.3 Scheduling

When a disk fails, its place in the schedule is effectively taken by the parity disk for its redundancy set. In addition, the parity information is organized by media stream using the same layout constraints as the media data. Therefore, the parity block read effectively is substituted for the stream's data block on the failed disk. As a result, the disk bandwidth requirements remain the same after the failure; only a substitution has occurred.

For each stream, recovering the data that was contained on the missing disk requires that the data from all remaining disks in the redundancy set be XOR'ed together with the parity data. Because the failed disk could be the first one in the redundancy set, all the other disks must be read first and the XOR performed before the data can be sent. This requires that these disks be read earlier than they would have been without failure⁵. Therefore, the scheduling during a failure is somewhat different than during normal operation. In addition, the number of buffers available in the system must be larger to hold the buffers participating in the XORs. These issues are addressed in a later section.

4.4 Media Reconstruction

It is assumed that the scheduler will not be responsible for reconstructing lost data and placing it back onto disk. External software will take care of rebuilding the data. There are several reasons for this:

- There may not be another disk available to recover onto until one is manually added.
- The scheduler won't, in general, know when all streams have been regenerated. In particular, titles that are not sent to any clients would never be reconstructed.
- It unnecessarily complicates the scheduler.

Instead, system management software is expected to use the Available-Bandwidth I/O facility (see Section 5 on page 11) to reconstruct the lost data without perturbing the active media

5. In non-failure mode, the scheduler reads disks in a just-in-time manner to minimize buffer requirements.

streams. A set of system management tools has been built for the Sun Media Server to perform these tasks [9].

4.5 Example

Assume 12 data disks D0-D11, and three parity disks P0, P1, and P2 providing parity for disks D0-D3, D4-D7, and D8-D11, respectively. In normal mode, this would operate the same as in the example in Table 2, with the parity disks idle.

4.5.1 Steady State With One Failed Disk

Now, assume a failure on disk D5, with its parity data on disk P1. First we look at the steady state, long after the failure occurred (shown in Table 2). Later, we'll examine the transient conditions that occur when the disk fails.

As a result of the failure, the server will substitute reads to disk P1 for reads to disk D5. In order to recover the data in time, we read disks D4-D7 (with P1 replacing D5) at the same time as disks D0-D3; D0-D3 are used for the normal just-in-time delivery and D4-D7 are used for recovery. Note that D4-D7 must be all in memory before the XOR can be done. We use a full time slot to do the XOR because there will be several streams in the slot and this gives sufficient time to process them all before the data is needed. Note that, at this point, there is no longer a 1-to-1 relationship between slots and disks. However, each disk is only performing work for one slot per time period. A slot will sometimes be using two disks per period and then immediately afterward the slot will use no disks for the same amount of time. In essence, we "borrow" the disk from a slot ahead in the train because the slot is not using it; when a slot that borrowed the disk gets to that spot it "lends" its disk to a slot behind in the train. So, it all works out as shown in Table 2.

Note that during time period t5 the recovered data for disk D5 is built (in place) in the parity buffer that had been read from disk P1. During time periods t6 through t8 the stream sends the recovered data along with the other buffers that were read previously, and requires no additional disk I/O. Beginning at time period t9 it begins normal operation on the next redundancy set. As illustrated in Table 2, the diagonal line of sends

(one per time period) is maintained. This ensures that there are no interruptions in the delivery of the data streams.

	Time period														
Disk	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14
D0	W	R	S										W	R	S
D1		W	R	S										W	R
D2			W	R	S										W
D3				W	R	S									
D4	W	R				⊕	S						W	R	S
D5								S'							
D6			W	R		⊕			S						W
D7				W	R	⊕				S					
D8									W	R	S				
D9										W	R	S			
D10	S										W	R	S		
D11	R	S										W	R	S	
P0															
P1		W	R			⊕		↑						W	R
P3															

Legend:

- W - Work-ahead read
- R - Read disk
- S - Send over network
- ⊕ - XOR for D5 data recovery
- S' - Send recovered data (from ↑)

Table 3: Timeline In Disk Failure Recovery Operation

4.5.2 Failure Transients

There are several transient conditions that occur when a disk fails. Streams reading from the redundancy set immediately before the one containing the failure (disks D0-D4 in the example) at the time of the failure will drop all data from the failed set (disks D5-D7) for one pass through the set (because they didn't do a read-ahead but must lend their disks). However, from then on they will proceed without interruption. In the example, these streams

would lose 2 seconds of video (i.e., four disks with 1/2 second of video each).

Streams using the disks in the redundancy set containing the failed disk at the time the disk fails will drop the data from the failed disk only. Because the subsequent group of streams will drop all the data from the set containing the failure (see above), the streams in the failure set can continue to use the non-failed disks in the just-in-time manner. They will only lose the data from the failed disk, and only if they hadn't read it before the failure. In the example, these streams would lose at most 1/2 second of video.

All other streams will be unaffected by the failure.

Finally, the admission control algorithm must be modified to avoid placing new streams in slots associated with the redundancy set containing the failed disk or the one immediately before it. In this way, newly admitted streams will be unaffected by the failure.

4.5.3 Buffer Requirements

Only the disks involved in error recovery use extra buffers. When the stream is outside the redundancy set containing the failure, they use just-in-time scheduling. The minimum buffer space required is:

$$(streams + data_disks) + (streams / redundancy_sets \cdot disks_in_redundancy_set)$$

In other words, using the Sun Media Server's disk layout and scheduling scheme gives a buffer requirement in normal operation of $(streams + data_disks)$. The extra buffering needed to handle failure recovery is calculated as $(streams / redundancy_sets \cdot disks_in_redundancy_set)$.

For example, assume 150 streams, 30 disks, 256KB buffers, and five disks in each redundancy set (4 + 1 parity). Then the minimum total buffer space for a system with cluster redundancy is $((150 + 24) + (150/6 \cdot 5)) \cdot 256K = 79$ MBytes. However, this amount of buffering would not be sufficient to permit the work-ahead feature to operate.

Reducing the cluster size would improve the memory requirements. However, it would require more disks to support a given configuration, or would reduce the maximum number of titles that a given group of disks could hold.

5 Available-Bandwidth I/O

The scheduler supports commands to allow applications to read and write the managed disks within the constraints of the media schedule. These operations are performed as soon as possible without perturbing the active media streams. As a result, the scheduler will use any excess capacity of the system to perform the reads and writes as quickly as possible.

With this facility, external software can perform media reconstruction after disk failures in the most efficient manner for all titles, and then notify the scheduler that the recovery is complete. Note that this may often be accomplished more quickly than if the scheduler itself were doing it because all of the excess bandwidth can be applied at once. The same mechanism can also be used to add new titles to the system during normal operation. To ensure timely completion of these operations, it is possible to create a desired amount of excess bandwidth in the system by limiting stream admissions.

The available-bandwidth I/O mechanism works by associating a list of outstanding requests with each disk. When I/O is completed for a slot, any disk bandwidth available to the slot that is currently unneeded by it will be used to service requests on the list. After this has completed, the normal work-ahead processing begins for the disk.

6 Analysis

6.1 Seek Time

Given a set of N ordered seeks and an algorithm that would seek to the nearest end of the set then proceed through the set to the nearest subsequent position (an elevator algorithm), the worst case

total seek distance for the set would be (as a fraction of the disk):

$$\text{total seek distance} \leq \frac{1}{2} + (N-1) \cdot \frac{1}{N-1} \leq 1.5$$

case 1: $N = 1$; The worst case seek is the whole disk.

total seek ≤ 1

case 2: $N > 1$; The starting disk position is in the middle of the group. Then one seek of at most one half of the disk is used to get to the nearest end of the group from the current head position. After that, there are $(N-1)$ seeks of average $\leq 1/(N-1)$ of the disk. That is, the worst case occurs when the starting position is in the middle of the disk and the set spans the disk.

total seek $\leq 1/2 + (N-1) \cdot 1/(N-1) = 1.5$

case 3: $N > 1$; The starting disk position is outside the group. Then there are N seeks of no more than $1/N$ of the disk average.

total seek $\leq N \cdot (1/N) = 1$

Therefore, if $\text{seek}(x)$ is the time to seek fraction x of the disk, then for the set of N accesses:

$$\text{total seek time} \leq N \cdot \text{seek}\left(\frac{1.5}{N}\right)$$

$$\text{average seek time} \leq \text{seek}\left(\frac{1.5}{N}\right)$$

If the number of streams in each slot, N , is greater than or equal to five ($N \geq 5$), then over each slot the average seek time is guaranteed to be less than the time to seek across the disk's average seek distance. That is, $\text{seek}(1.5/N) < \text{seek}(1/3)$.

Finally, in steady state operation of a media server which uses a two-way elevator algorithm as described above, it can be seen that, on average, each time period will experience no more than one full seek of each disk. Let $L(i)$ be the lowest block number needed from the disk in time period t_i , and $H(i)$ be the highest block number needed from the disk in time period t_i . Then, for any sequence of time periods $t_n, t_{n+1}, t_{n+2}, \dots$, and assuming (without loss of generality) that at the beginning of t_n the disk head is closer to $L(n)$ than $H(n)$, that the disk will seek no

further than to the extreme points shown by the following pattern:

$$\begin{aligned} &L(n) \\ &\max(H(n), H(n+1)) \\ &\min(L(n+1), L(n+2)) \\ &\max(H(n+2), H(n+3)) \\ &\dots \end{aligned}$$

So, in steady state operation, each disk will perform, on average, no more than one full seek each time period. This means that the average seek time will be less than $\text{seek}(1/N)$, while the instantaneous worst-case is bounded at $(1.5/N)$. Therefore, a server using this method and serving a moderate load of clients will achieve the average case seek of the disks.

6.2 Rotational Latency

The worst-case rotational latency for a disk is the time it takes to make one revolution of the disk. Modern high-performance disks rotate at 7200 RPM, so the worst-case latency is approximately 8.3 milliseconds. The average-case latency (assuming random accesses) is one half the worst-case latency, or approximately 4.2 milliseconds. Because the locations accessed in a media server are not correlated (they represent independent streams), average-case latencies can be expected and the worst-case is unlikely to persist across several accesses. Because of this and the fact that the difference between worst-case and average-case rotational latencies are small, it is convenient to treat the average rotational latency as another component of the seek overhead in server capacity calculations (e.g., see Drawing 1 on page 14).

6.3 Bandwidth

Each slot contains streams using roughly equal bandwidth from each zone. For example, if all streams use the same bandwidth b , each slot will contain approximately $N = B/(b \cdot D)$ streams. For any given slot, the number of streams in the worst case zone is $\lceil N/Z \rceil$, and there are at least $\lfloor N/Z \rfloor$ in the best zone. (If there are more than two zones, the others will also contain at least $\lfloor N/Z \rfloor$ streams.) So given the worst case bandwidth for transfers from each zone, the worst case bandwidth for each slot can be calculated. This is better than the disk's worst

case and, with typical modern disks, approaches the average case for the disk.

In addition, because the layout of the disks alternates between zones on each data file, each stream will have traversed each zone in every Z accesses. Therefore, each stream will achieve approximately average case disk bandwidth internally in every Z time periods (except, perhaps from the last to first disk; however, even this effect cancels over a longer interval). As a result, the server achieves an essentially complete balance between the zones every Z time periods, assuring the average-case disk bandwidth overall.

Similarly, the admission algorithm ensures, over each set of adjacent slots, that the bandwidth utilization is balanced by zone. Such a set of slots are called a *slot group*. Across the set of slots in each slot group, the bandwidth used by streams in any zone will not exceed the sum of the bandwidths of the slots divided by the number of zones. Then, as the slot group passes over each disk any imbalance in zone usage for the disk is cancelled within the number of time periods equal to the size of the largest slot group. Therefore, each disk will achieve approximately average case bandwidth internally across each slot group.

6.4 Admission

A stream can be assigned to any slot; however it must not transfer until the first data block it needs is on the disk assigned to its slot and the block is in the proper zone. Therefore the worst case time to obtain the first block of data is $D \cdot Z \cdot t$, where t is one scheduling period.

The start-up delay that a stream will experience if admitted to a given slot can be calculated given the current disk assigned to the slot and knowledge of the transitions from disk-to-disk and zone-to-zone. Using this calculation method, the admission control subsystem allows each admission to be constrained by a maximum start-up delay. If the stream cannot be admitted into a slot which can provide the first block within the delay constraint, the admission can be rejected. This can be used to allow the client to find an alternate server that will supply the data in time.

Finally, although the admission process has been described as *first come first served* (FCFS), there are other reasonable policies. FCFS is appropriate where all clients are of approximately equal importance, as when the server is used to deliver movies into a large number of homes. However, if importance values can be assigned to clients, the admission algorithm can use them to maximize the overall value delivered. For example, an appropriate policy could be to preempt a running stream of a lower importance in order to admit a later one with a higher importance.

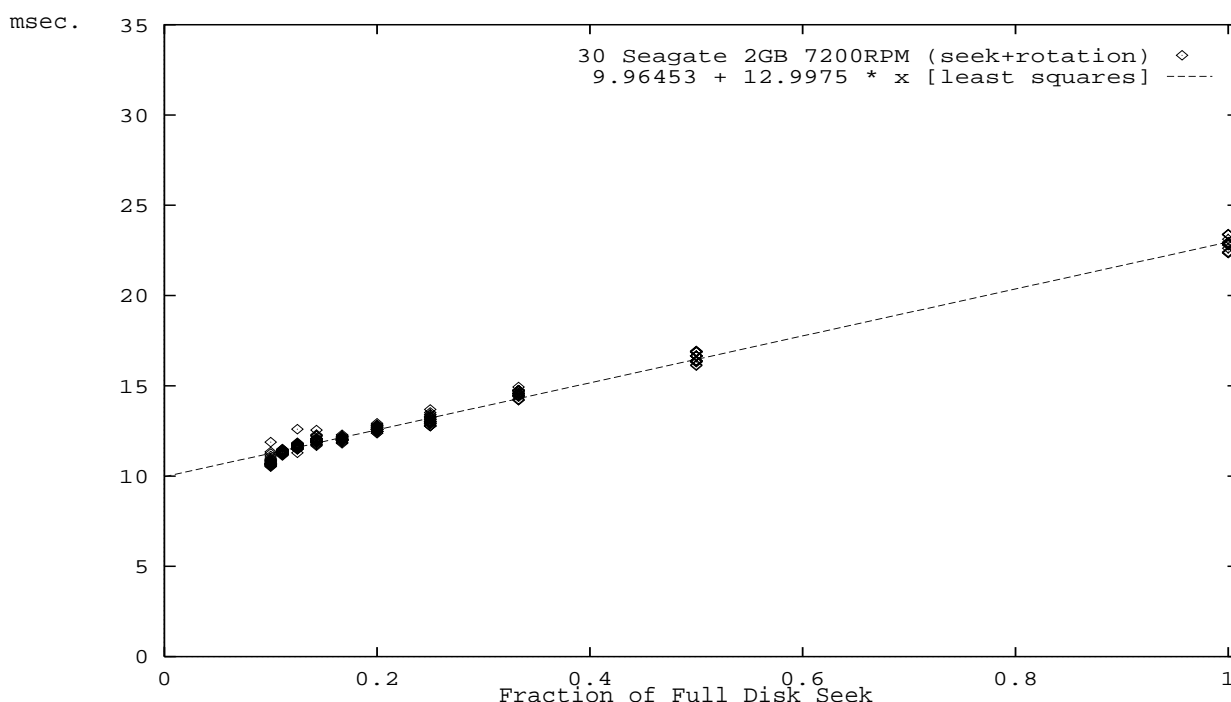
7 Implementation

The Sun Media Scheduler has been implemented as a Solaris device driver which interacts with the Data Pump (also a device driver). The system uses Solaris' real-time scheduling class to ensure that disk scheduling decisions are made in a timely manner. The scheduling and admission subsystems work from a configuration specification provided by a user-mode configuration utility. The actual parameters are derived from the Media File System [9] specification, augmented by a configuration file in the form of a Tcl [7] script. The configuration parameters include:

- the scheduling interval
- the number of data and parity disks
- disk device identification (major and minor device numbers)
- the access order of the data disks
- the zone transition order
- seek time information for the disks
- the worst-case bandwidth of each zone
- total network bandwidth
- the amount of buffering to use
- site-imposed limits on the number of streams or total bandwidth

Because of this facility, the software can be configured for any size system without changing the scheduling and admission module.

Although the design of the Sun Media Scheduler allows stream delivery guarantees to be made, it



DRAWING 1. Measured Seek and Rotational Latency Times

is not necessary to operate a server in this mode. The disk bandwidth values used in stream admission are conservative bounds based on the average of the worst case bandwidth in each zone. However, in practice the actual observed bandwidth will be essentially equal to the disks' overall average. In addition, the system assumes one worst-case seek per scheduling interval on each disk, even though the actual seek will nearly always be less than this. So, if absolute server capacity is very important and if clients can accept occasional times where data delivery is a little late, then the server can be configured to admit streams beyond the guaranteed capacity. This is accomplished in the Sun Media Scheduler by setting an *overbooking* configuration parameter to an amount appropriate to the installation.

The first product to use the Sun Media Server software is the SMS-1000. This server is based on the SPARCcenter 1000 computer and a modified SPARCstorage Array. The SMS-1000 is rated to deliver a sustained 400 Mbits/second, serving individual streams of 1-8 Mbits/second. It can survive the failure of any single disk while maintaining the full delivery rate.

The SPARCcenter 1000 in the SMS-1000 is configured to have two processors, six SCSI controllers, four Sun SAHI ATM interface cards (155 Mbit/second OC-3), and at least 128 MBytes of memory. Approximately 100 MBytes of memory is used as buffers by the Sun Media Scheduler.

The SPARCstorage Array contains up to 30 disks, each running at 7200 RPM and holding 2 GBytes of data. It has been modified to remove the FibreChannel controller, and instead bring six differential SCSI chains directly to the SCSI controllers in the SPARCcenter 1000. This was necessary because the FibreChannel controller could not supply the full bandwidth of the disks.

The disks are organized into six redundancy sets containing four data disks and one parity disk. Each redundancy set occupies one SCSI chain. This ensures that the load on each SCSI chain remains balanced even in the event of a disk failure.

Drawing 1 shows a plot of the combined seek time and average rotational latency (in milliseconds) measured over a large number of trials with 30 identical disks available in the SMS-

1000. As can be observed from the drawing, the assumption of a fixed overhead and a distance-related seek time is valid for these disks.

The transfer bandwidth for these disks ranged from 4.9 MBytes/second at the outer rim to 4.5 MBytes/second near the inner edge. The system can be configured with two zones with worst-case bandwidth of 4.7 MBytes/second and 4.5 MBytes/second. Because of the roughly even response of these disks, the bandwidth management scheme provides a limited gain.

By contrast, the standard Sun 1 GByte 5400 RPM disks range from 4.2 MBytes/second at the outer rim to 2.4 MBytes/second at the inner edge. A server configured with these disks would use two zones of 3.5 MBytes/second and 2.4 MBytes/second. Here the bandwidth difference between the zones is 45%.

In the SMS-1000, the set of disks on each SCSI chain overloads the SCSI controller in simultaneous accesses. That is, when all 24 data disks are active at the same time, the effective bandwidth drops due to SCSI contention. The result is that each disk can provide approximately 3.0 MBytes/second of bandwidth.

The SMS-1000 is usually configured to use 1/2 second scheduling intervals. Typical titles are recorded at 4 Mbits/second. Therefore, the buffer for each such stream in a time interval is approximately 250 KBytes. As a result, the 7200 RPM disks can support approximately 5.2 of the 4 Mbit/second streams each time period using simultaneous accesses. With 4 Mbit streams, each disk spends approximately 86% of its time transferring data, and approximately 14% of its time seeking to the next location. If higher bandwidth titles are used, a larger percentage of the time is spent in transfers.

In the SPARCcenter 1000, processors access memory across the high-bandwidth multiprocessor system bus (the XDBus™) through direct-mapped caches. This design, combined with the higher latencies associated with this type of bus, can lead to *cache thrashing* on operations like parity reconstruction. In order to control the system bandwidth required for disk failure recovery, an optimized XOR

calculation facility was developed. This feature uses cache blocking to ensure that the memory bandwidth is used most effectively. One (64 byte) cache block for each buffer participating in the error recovery is read in turn into the processor and accumulated into the calculation. When this is done, the resulting cache block is written to the buffer for the recovered data. The use of this facility ensures that the absolute minimum system bus bandwidth is consumed.

In normal mode, the Sun Media Scheduler and Data Pump consume approximately 15-25% of one 60 MHz SuperSPARC processor when serving a full load of 400 Mbits/second (with debugging checks turned on). Disk failure recovery (XOR) processing further consumes 40-50% of one processor at full load. Therefore, a fully loaded SMS-1000 will not fully occupy one processor even in disk failure recovery mode.

The basic structure of the SMS-1000 is shown in Figure 2. The ATM network is used to connect to set-top boxes, desktop workstations, or a combination of both. The network provides point-to-point connections or creates the spanning trees necessary to support broadcast and multicast transmissions. Future versions of the Sun Media Server will support other delivery mechanisms, including Fast Ethernet.

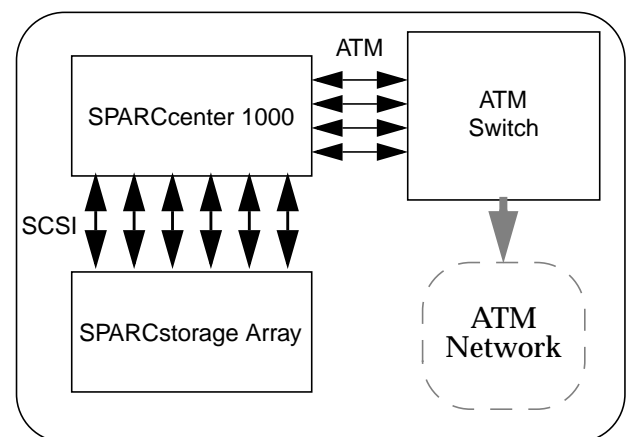


FIGURE 2. SMS-1000

The SMS-1000 has demonstrated sustained, reliable delivery of over 400 Mbits/second in both normal operation and during disk failure recovery mode. Therefore, if 4 Mbits/second

MPEG-2 streams are used, the server can easily support up to 100 users indefinitely.

Additional applications of the Sun Media Server software are planned based on the SPARCstation 20 platform with 1GByte disks for small-scale systems, and the SPARCcenter 2000 for large-scale systems. The SPARCcenter 2000 version can be configured with up to 60 disks, with each disk holding 9 GBytes of data and delivering approximately 4.6 MBytes/second.

8 Simulator

A discrete event simulator was developed for the Sun Media Scheduler as an aid for the debugging and analysis of the scheduling and admission control processes. The simulator framework provides accurate behavioral models of the disks, the network, the Data Pump, and the kernel synchronization services that the scheduler uses.

The design of the simulator allows approximately 90% of the actual Sun Media Scheduler code to be run in the simulation environment. This is accomplished by imitating the programming interface of the various subsystems used by the scheduler.

Facilities have been included in the framework to allow creation of scenarios involving the admission and shutdown of multiple clients, transient disk slowdowns, and disk failures and restorations. The simulator produces a log file based on trace-points that can be used to verify the correctness of the run. These trace-points can also be monitored in a live Sun Media Server using the Solaris kernel trace facility (vtrace).

The simulator has been especially useful in debugging the Sun Media Scheduler because the kernel debugging environment is rather unforgiving. Some errors in kernel code can cause a machine to seize up, with no additional information available. In contrast, the same code running on the simulator can be monitored with the standard user-mode source level debuggers.

The simulator is also useful for characterizing the capacity of hypothetical configurations. By giving the disk characteristics and the bandwidth

of the titles for a system, an accurate estimate of the number of supported streams can be derived. In addition, it can be used to determine the amount of buffer memory that should be included in a system to provide the desired level of protection from transient disk errors.

Finally, the simulator has been useful in ensuring the reliability of the Sun Media Scheduler software. Because the simulator runs at approximately ten times real time, it was possible to log over one year of run time using randomly-generated scenarios of various client loads, disk problems, etc. before the alpha shipment of the SMS-1000 software.

9 Conclusions

The design presented in this paper and used in the Sun Media Server can guarantee approximately average case bandwidth and seek times from a set of disks and deliver data at that rate to a set of clients. It is able to do so in the face of substantial non-determinism in the response of the disks themselves and even in the event of complete failure of any single disk. It achieves this with only a moderate investment in both memory and processor resources.

10 References

- [1] J. Gemmell et al., "Multimedia Storage Servers: A Tutorial," *IEEE Computer*, May 1995, pp. 40-49.
- [2] J. Hanko et al., "Workstation Support for Time-Critical Applications", *Proceedings of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, November, 1991.
- [3] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufman Publishers, 1990, pp. 557.
- [4] R. Khazanie *Basic Probability Theory and Applications*, Good-year Publishing, 1976.
- [5] K. Mandal, "Data Pump for Video On Demand Applications," Sun internal document, 1994.
- [6] J. D. Northcutt et al., "A High Resolution Workstation," *Signal Processing: Image Communications*, Elsevier, August 1992, pp. 445-455.
- [7] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Co., 1994.
- [8] F. Tobaji et al., "Streaming RAID™ - A Disk Array Management system For Video Files", Starlight Networks Inc., 1993.

- [9] J. Voll, J. Hanko. and K. Mandal, "Video-on-Demand Server Software", *Proceedings of the First SunSoft Technical Conference*, April 1995, pp. 179-192.
- [10] G. Wall et al., "Bus Bandwidth Management in a High Resolution Video Workstation", *Proceedings of the Third International Workshop on Network and Operating Systems Support for Digital Audio and Video*, November, 1992.

11 Glossary

admission policy - An algorithm for deciding whether to admit a new stream into the server and, if so, how it is to be managed.

chunk - An area of memory for holding media data that is separately allocated and freed.

client - The receiver of a stream of media data.

Data Pump - A kernel module that provides optimized zero-copy access to disk and network devices, as well as rate-regulated network delivery.

disk zone - An area of a disk with a particular transfer bandwidth.

extent - A contiguous group of disk blocks holding stream data for one scheduling interval.

media stream - A continuous stream of time-critical data.

parity - A value used in error correction or detection that is based on the bit-wise exclusive-or of data bits.

redundancy set - A set of disks for which redundant information is kept that enables data recovery after a disk failure.

rotational latency - The delay from the time a disk's read/write head arrives on the requested track until the desired location rotates into position under the head and can be accessed.

scheduling interval - A site-selected fundamental time period in which a disk read is completed for all streams.

seek - The process of moving a disk's read/write head to the track containing a specific location

slot - A a group of streams that access the same disk(s) at any point in time.

slot group - A group of slots over which bandwidth used from all zones is balanced.

stream - The active process of reading data from the disks and delivering it to a client over a network.

thermal calibration - A process performed by some disk drives to compensate for apparent disk geometry changes caused by thermal effects.

work-ahead - A mechanism that allows the media server to begin work early for a subsequent scheduling interval when work for the previous one completes.

12 Acknowledgments

The zoned bandwidth management and disk layout techniques evolved out of a discussion with Steve Kleiman. The disk error recovery method similarly evolved from discussions with Jerry Wall. The Data Pump was designed and implemented by Kallol Mandal, who also assisted in hooking the Sun Media Scheduler into the device driver framework. Jim Voll developed the Media File System (MFS) tools to create, delete, rebuild, and manipulate titles, as well as the libraries for accessing the server facilities. Chris Moeller and Mike DeMoney created a video-on-demand system with menus and full VCR control using the SMS-1000 and Open TV™ set-top boxes. Kenneth Wong developed an external daemon program to monitor the disks and provide an early indication of disk problems.

Solaris MC: A Multi-Computer OS

**Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff,
Moti Thadani**

Introduction by Yousef A. Khalidi

This technical report describes a prototype distributed operating system for closely-coupled computer clusters. The prototype system, called Solaris MC, was built as a set of extensions to the base Solaris™ Operating Environment UNIX® system and provided the same ABI/API as the Solaris OE, running unmodified applications. We used several techniques to extend Solaris OE using a CORBA-compliant object oriented system. Objects communicated through a runtime system that borrowed from Solaris OE doors and Spring subcontracts. Solaris MC provided a single-system image: a cluster appeared to the user and applications as a single computer running Solaris. Solaris MC was designed for high availability: if a node failed, the remaining nodes and cluster services remained operational.

After completing the Solaris MC prototype, the team and technology were transferred out of Sun Labs to form the core of a new product group chartered with building Sun's next generation clustering product. This effort resulted in Sun™ Cluster 3.0 (Full Moon) product which was released in November of 2000. Sun Cluster 3.0 is Sun's most powerful and comprehensive cluster solution ever. Sun Cluster 3.0 focuses on delivering integrated availability, scalability, manageability and ease of use with the core Solaris OE (<http://www.sun.com/clusters>).

We learned many lessons starting from the prototype work in Sun Labs, through product development of Sun™ Cluster 3.0 and customer deployment of the final product. The initial emphasis on single-system image was later relaxed in the final product as we balanced it against the increasing requirements of high-availability.

Many of the initial design decisions we made in Solaris MC were proven correct, including the use of object-oriented techniques, and seamless recovery of system services. Finally, our early decision to extend Solaris without breaking any applications was key to making the transition from a prototype system to a major Sun software product.

REFERENCES:

Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani, "Solaris MC: A Multicomputer Operating System," Proceedings of Usenix 1996, January 1996, pp. 191–203.

Ken Shirriff, Jose Bernabeu Auban, Yousef A. Khalidi, Vlada Matena, "Single System Image: The Solaris MC Approach," Proceedings of PDPTA '97, July 1997, pp. 1097–1105.

Ken Shirriff, "Building distributed process management on an object-oriented framework," Proceedings of Usenix 1997, January 1997, pp. 119–131.

Jose M. Bernabeu-Auban, Vlada Matena, and Yousef A. Khalidi, "Extending a Traditional OS Using Object-Oriented Techniques," USENIX Conference on Object-Oriented Technologies, 1996.

Solaris MC: A Multi-Computer OS

Yousef A. Khalidi
Jose M. Bernabeu
Vlada Matena
Ken Shirriff
Moti Thadani

SMLI TR-95-48

November 1995

Abstract:

Solaris MC is a prototype distributed operating system for multi-computers (i.e., clusters of nodes) that provides a single-system image: a cluster appears to the user and applications as a single computer running the Solaris™ operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same ABI/API as Solaris, running unmodified applications. The components of Solaris MC are implemented in C++ through a CORBA-compliant object-oriented system with all new services defined by the IDL definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts. Solaris MC is designed for high availability: if a node fails, the remaining nodes remain operational. Solaris MC has a distributed caching file system with UNIX consistency semantics, based on the Spring virtual memory and file system architecture. Process operations are extended across the cluster, including remote process execution and a global /proc file system. The external network is transparently accessible from any node in the cluster. The prototype is fairly complete—we regularly exercise the system by running multiple copies of an off-the-shelf commercial database system.



M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:

yousef.khalidi@eng.sun.com
josep@iti.upv.es
vlada.matena@eng.sun.com
ken.shirriff@eng.sun.com
moti.thadani@eng.sun.com

Solaris MC: A Multi-Computer OS

Yousef A. Khalidi Jose M. Bernabeu Vlada Matena Ken Shirriff Moti Thadani

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Solaris MC¹ is a prototype operating system for a multi-computer, a cluster of computing nodes connected by a high-speed interconnect. The Solaris MC operating system provides a single system image, making the cluster look like a single machine to the user, to applications, and to the network. By extending operating system abstractions across the cluster, Solaris MC preserves the existing Solaris ABI/API and runs existing Solaris 2.x applications and device drivers without modification.

The decision to design a cluster operating system was motivated by trends in hardware technology. Traditional bus-based symmetric multiprocessors (SMP) are limited in the number of processors, memory, and I/O bandwidth that they can support. As processor speed increases, traditional SMPs will support an even smaller number of CPUs. Powerful, modular, and scalable computing systems can be built using inexpensive computing nodes coupled with high-speed interconnection networks. Such clustered systems can take the form of loosely-

coupled systems, built out of workstations [1], massively-parallel systems (e.g., [24]), or perhaps as a collection of small SMPs interconnected through a low-latency high-bandwidth network.

The key to using clustered systems is to provide a single-system image operating system allowing them to be used as general purpose computers. Cluster systems in the past have been mostly used for custom-built parallel and distributed applications, and sometimes as specialized database systems. However, to fully exploit the potential of clustered systems, we believe that they have to be usable as *general purpose computers*, running existing applications without modification. Moreover, clustered systems have to be easy to administer and maintain. The fact that the computer is actually built out of multiple computing nodes should be invisible to the user. Finally, since clustered systems are built out of many components, the clustered system should be *highly-available* and should be able to tolerate the failure of any one component.

Our goals are to make a cluster of nodes that may or may not share memory appear as a single general purpose multiprocessor. It should be seen as a single machine by appli-

1. Solaris MC is the internal name of a research project at Sun Microsystems Laboratories. More information on the project can be obtained from <http://www.sunlabs.com/research/solaris-mc>.

cations, users, and administrators. We want this while preserving object code compatibility (the ABI), minimizing changes to kernel code, requiring minimal or no change to device drivers, and supporting high availability.

Solaris MC has several interesting features. It:

- Extends existing Solaris operating system

Solaris MC is built on top of the Solaris operating system. Most of Solaris MC consists of loadable modules extending the Solaris OS, and minimizes the modifications to the existing Solaris kernel. Thus, Solaris MC shows how an existing, widely-used operating system can be extended to support clusters.

- Maintains ABI/API compliance

Existing the application and device driver binaries run unmodified on Solaris MC. To provide this feature, Solaris MC has a global file system, extends process operations across all the nodes, allows transparent access to remote devices, and makes the cluster appear as a single machine on the network.

- Supports high availability

The Solaris MC architecture provides fault-containment at the level of an individual node in the multi-computer. Solaris MC runs a separate kernel on each node. A failure of a node does not cause the whole system to fail. A failed node is detected and system services are reconfigured to use the remaining nodes. Only the programs that were using the resources of the failed node are affected by the failure. Solaris MC does not introduce new failure modes into UNIX.

- Uses C++, IDL, and CORBA in the kernel

Solaris MC illustrates how the CORBA (*common object request broker architecture*) object model can be used to extend an existing UNIX operating system to a distributed OS. At the same time, it also shows the advantages of implementing strong interfaces for kernel components by using IDL (*interface definition language*). Finally, Solaris MC illustrates how C++ can be used for kernel development, coexisting with previous code.

- Leverages Spring technology

Solaris MC illustrates how the distributed techniques developed by the Spring OS [15] can be migrated into a commercial operating system. Solaris MC imports from Spring the idea of using a CORBA-compliant object model [18] as the communication mechanism, the Spring virtual memory and file system architecture [7, 10, 9], and the use of C++ as the implementation language. One can view Solaris MC as a transition from the centralized Solaris operating system toward a more modular and distributed OS like Spring.

Solaris MC uses ideas from earlier distributed operating systems such as Sprite [19], LOCUS [20], OSF/1 AD TNC [26], MOS [2], and Spring. One key difference from other systems is that Solaris MC shows how a commercial operating system can be extended to a cluster while keeping the existing application base. In addition, Solaris MC uses an object-oriented approach to define new kernel components. Solaris MC also has a stronger emphasis on high availability. Finally, Solaris MC uses new techniques for making the cluster appear as a single machine to the external network.

The remainder of this paper is structured as follows. Section 2 explains the global file system. Section 3 describes how process management is globalized. Section 4 explains how I/O devices are made global,

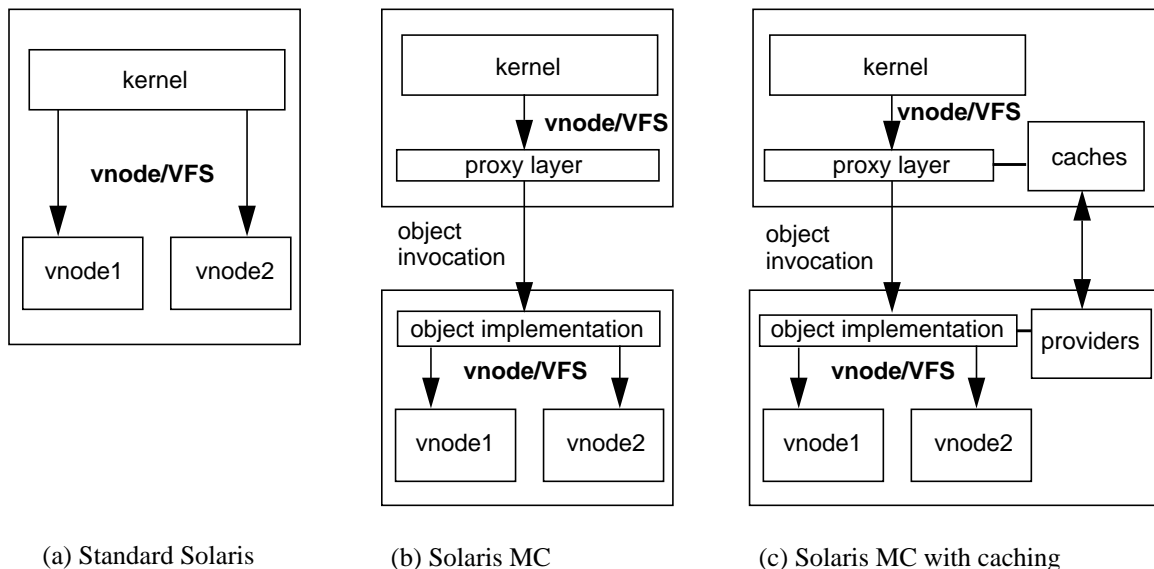


Figure 1. Extending File System Interfaces for Solaris MC. (a) In Solaris, the kernel accesses files through the VFS/vnode operations. (b) In Solaris MC, the VFS/vnode operations are converted by a proxy layer into *object invocations*. The invoked object may reside on any node in the system. The invoked object performs a local VFS/vnode operation on the underlying file system. Neither the kernel nor the existing file systems have to be modified to run under Solaris MC. (c) Caching is used in Solaris MC to improve performance. Solaris MC supports caching of file pages, directory information, file attributes, and mount points.

and Section 5 explains how network operations are made transparent. Section 6 discusses the object-based communication model of Solaris MC, and explains CORBA and IDL. Section 7 briefly describes how Solaris MC provides high availability. Section 8 provides the current status of Solaris MC, Section 9 compares Solaris MC to other distributed operating systems, and Section 10 concludes the paper.

2 Global File System

Solaris MC uses a global file system to make file accesses location transparent—a process can open a file located anywhere in the system and processes on all nodes can use the same pathname to locate a file. The global file system uses coherency protocols to preserve the UNIX file access semantics even if the file is accessed concurrently from multiple nodes. This file system, called the proxy file system (PXFS), is built on top of the existing Solaris file system at the *vnode* [11] interface. This interface allows PXFS to

be implemented without kernel modifications. The PXFS file system provides extensive caching for high performance using the caching approach from Spring [7], and provides zero-copy bulk I/O movement to move large data objects efficiently. This section discusses these features of PXFS in more detail.

PXFS interposes on file operations at the *vnode/VFS* interface and forwards them to the *vnode* layer where the file resides, as shown in Figure 1. Besides files, PXFS also provides access to other types of *vnodes*, such as directories, symbolic links, special devices, streams, swap files, fifos, and Solaris doors.² Because PXFS is built on top of the existing file system, it can leverage off the existing file system code. This is an important difference from distributed file

2. Solaris doors is a new IPC mechanism in Solaris 2.5 that is based on the Spring IPC mechanism [15].

systems such as Sprite or Spring that rewrite the entire file system.

PXFS uses extensive caching on the clients to reduce the number of remote object invocations. Figure 2 shows the objects used in the file paging and attribute caching protocols. The design of PXFS was influenced by the Spring file system and its caching architecture [7, 17, 16]. A client cache is implemented through a *cached* object on the client to manage the cached data and a *cacher* object on the server to maintain consistency. For data, the client has a *memcache* object and the server has a *mempager* object. For attributes, the client has a *attrcache* object and the server has a *attrprov* object.

As an example, suppose a process on Client 1 wishes to page in a page from a file. A *memcache* is a vnode in addition to being an IDL object, so it can accept GETPAGE and PUTPAGE operations from the Solaris virtual memory system. The *memcache* vnode is used as the paged vnode for the VOP_MAP operations on the proxy vnode. *Memcache* searches the local cache for the page. If it is not available, *memcache* requests the page from the associated *mempager*. The *mempager* checks the other *mempagers* to see if another client has the page, to maintain consistency. Finally, the page is obtained from the backing server vnode. Thus, PXFS has control over global page coherence.

The PXFS coherency protocol is token-based and allows a page to be cached read-only by multiple caches or read-write by a single cache. If a dirty page is transferred from one node to another, it is first written to the stable storage on the server to avoid losing updates due to crashes of unrelated nodes. Similarly, an attribute cache is also protected by a reader-writer token. The token is also used to enforce atomicity of read/write system calls on regular files. Token

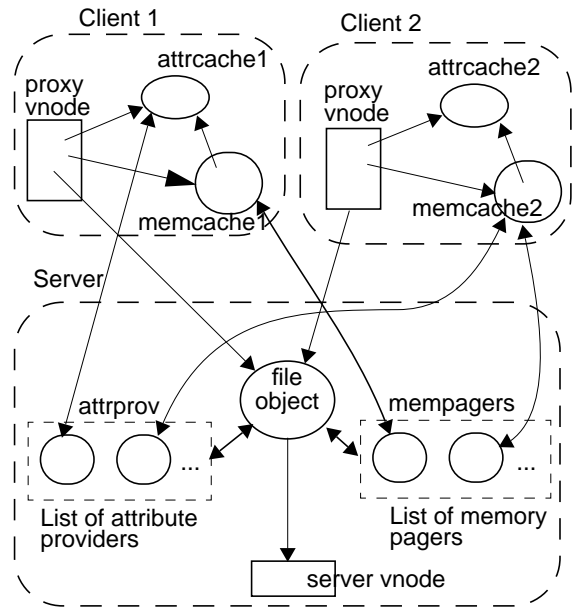


Figure 2. File Paging and Attribute Caching. Each client has a *memcache* object to cache data and an *attrcache* object to cache attributes. The server has corresponding *mempager* and *attrprov* objects to provide the data and attributes. The file object is an IDL object implementing the file protocol. The server vnode provides the underlying file storage.

management is integrated with data transfer for better performance.

Directory caching and caching of mount points is done in a fashion similar to attribute caching. Directory operations that create or remove objects are implemented as write-through to be reflected synchronously in stable storage on the server.

PXFS has a “bulkio” object handler to perform zero-copy transfers between nodes of large data (file pages, uioread/uiowrite data) if the hardware interconnect has sufficient support. For example, if a process takes a page fault, it allocates a page in the local cache and invokes the *page_in* method on the *mempager*. The server then allocates a kernel buffer and reads the data from the disk into the buffer. The data is then transferred using the bulkio handler directly into the page on the client. If the underlying hardware supports shared memory, the server can

map the client page and read data from the disk directly into the page without the need for an intermediate buffer on the server. By using a separate handler for bulk I/O, no changes to the PXFS client or server code are necessary to port PXFS to a different interconnect; only the bulkio handler has to be ported to take full advantage of the hardware.

3 Global Process Management

Global process management in Solaris MC extends OS process operations so that the location of a process is transparent to the user. While the threads of a single process must be on the same (possibly multiprocessor) node, a process can reside on any node. The design goals of process management are to support POSIX semantics for process operations while providing good performance, supplying high availability, and minimizing changes to the existing Solaris kernel. This section discusses the implementation of process management and how it transparently provide signals on global process ids, distributed waits, the /proc file system, and process migration.

Process management is implemented in a kernel module above the existing Solaris kernel code that manages the global view of processes. As illustrated in Figure 3, this layer consists of a virtual process (vproc) object for each local process, and a node manager for the node. The vproc maintains state such as the parent and children of the process. The node manager keeps track of the local processes and the other nodes. Additional objects manage process groups and sessions.

The global process layer interacts with the rest of the system in several ways. First, process-related system calls are redirected to this layer. Second, a small number of hooks were added to the kernel to call this layer

when appropriate. Finally, the vproc layers on different nodes communicate through IDL interfaces. Process management was made more difficult by the lack of an existing kernel interface (analogous to vnodes for the file system). We are exploring if the vproc interface can be extended to a flexible kernel interface useful for other system extensions.

Process identifiers (pids) in Solaris MC use a single global pid space and encode the home node of the process in the top bits. Thus, an arbitrary process can be located from its pid by contacting the home node, which knows the current location; this location can then be cached. The signal delivery code, for instance, uses the pid to deliver signals to a process no matter where it resides. The pid encoding also ensures that processes on different nodes will not be created with the same pid. The same pid is used inside and outside the kernel; Solaris MC does not use distinct local (internal) pids and global (external) pids.

Waits pose problems for a cluster because a parent and child may be on separate nodes.

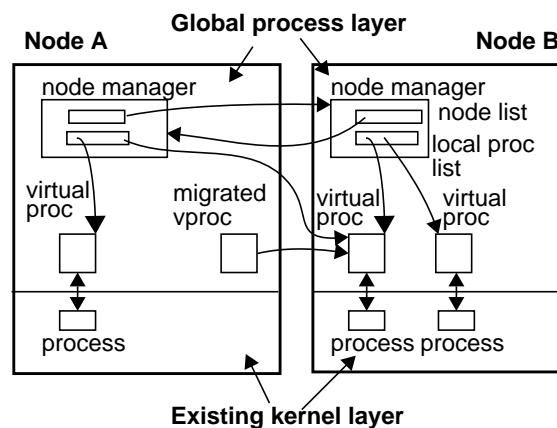


Figure 3. The data structures of the global process layer. Each node has a node manager object that has a list of all processes created or residing on the node and a list of the other nodes. Each process has a virtual process (vproc) object associated with it. When a process migrates, the old vproc is left behind to forward any operations. The vprocs keep track of the parent/child relationships of the processes.

In Solaris MC, distributed waits are implemented by having the child inform the parent of each state change (exit, stopped, continued, or debugged). The parent keeps track of the state of each child and wait operations use this local copy.

In the Solaris OS, the `/proc` pseudo file system provides access to each process in the system; this is used by `ps` and the debugger, for instance. In Solaris MC, the `/proc` file system is extended to cover all processes in the cluster. Code for `/proc` in the `pxfs` file system merges together local `/proc_local` file systems into a global `/proc`. Thus, directory operations on `/proc` show the process entries in all the local `/procs`, and lookup operations are redirected to the appropriate node.

Solaris MC currently supports remote execution of processes and will soon support remote forks and migration of existing processes. For a remote fork or migration, most of the process's state will be moved automatically through the consistency mechanism of `pxfs`. A "shadow `vproc`" is left behind when a process migrates; any operations received by the shadow `vproc` are forwarded to the `vproc` on the node where the process resides. The policy decisions on load balancing will be built on top of the migration mechanisms; one possibility is to use a migration daemon, as in Sprite [5], that will decide which nodes should receive processes. However, we believe that the main use of process migration will be for planned shutdown of cluster nodes rather than fine load balancing across the cluster; load balancing will largely be managed by placement of processes at exec time.

Global process management in Solaris MC will support high availability. That is, the failure of a node will not interfere with processes on another node. While the processes on a failed node will die, the rest of the system will continue after a recovery

phase. Parents and children will be notified appropriately of process failures. A new node will take over as home node for the failed node, and migrated processes that originated on the failed node will now use the new node as home.

4 I/O Subsystem

The I/O subsystem makes it possible to access any I/O device from any node in the multi-computer without regard to the physical attachment of devices to nodes. Applications are able to access I/O devices as local devices even when the devices are physically attached to a node different from the one on which the application is running. Several areas require attention to ensure this access:

- Device configuration: the Solaris OS provides dynamically loadable and configurable device drivers. Solaris MC transparently provides a consistent view of device configurations through a distributed device server that is notified when a new device is configured into the system on a particular node. When the device driver corresponding to the newly configured device is invoked on a different node, it is loaded on that node using the DDI/DKI device interfaces defined for the Solaris OS. Different nodes in the system may have different devices attached and different sets of drivers/modules loaded in kernel memory at any point in time.

The device server distributes the functionality of the Solaris `modctl()` interface, which handles the loading and unloading of dynamically loadable modules. Module configuration routines such as `make_devname()` add the new device names to the device server. Module control interfaces such as `mod_hold_dev_by_major()`, `ddi_name_to_major()`, and

ddi_major_to_name() look-up the distributed device database rather than local data structures.

- Uniform device naming: Device numbers provide information about the location (i.e., node number) of the device in the system in addition to the type of device and the instance or unit number of the device. The operating system associates a location with every device special file. When a device is opened, the *open()* is directed to the node to which the physical device is attached.
- Providing process context to device drivers: Device drivers require access to process context for data transfer and credentials checking. In Solaris MC, the calling process may be on a different node than the node on which the driver executes. Consequently, the process context in which the driver runs is different from the process context of the calling process. The operating system provides a logical equivalence between the two processes in order for device drivers to be able to function without modification.

The Streams framework poses additional problems, which are not discussed in detail here due to space limitations. Solaris MC allows Streams device drivers and modules that use procedural interfaces to work unchanged in the new environment. Some modules, however, do not strictly obey the Streams interface; they may either be modified to run on Solaris MC, or they may be confined to one node in the cluster.

5 Networking

The networking subsystem in Solaris MC creates a single system image environment for networking applications. The operating system ensures that network connectivity is the same for every application, regardless of which node the application runs on. This

goal is achieved with minimal impact on the existing network subsystem implementation and without any changes to applications.

We considered three approaches for handling network traffic. The first approach was to perform all network protocol processing on a single node. This approach, however, is not scalable to large numbers of nodes. The second approach was to run network protocols over the interconnection backplane. This approach requires each node to have a separate network address, which prevents transparency. The third approach, which we took, was to use a packet filter to route packets to the proper node and perform protocol processing on that node.

Our approach creates the illusion that the set of real network interfaces available in the system is local to each node in the system. Applications are unaware of the real location of each network device, and their view of the network is the same from every node in the system. When an application transmits data over an illusory network device on a node, the framework forwards the outgoing network packet to the real device. Similarly, on the input side, the framework forwards packets from the node on which the real network device is attached to the node where the appropriate application is running.

The advantages of our design are (a) protocol processing is not limited to those nodes that have network devices, (b) only one new module is written to handle networking for most protocol stacks, and (c) changes to the protocol stacks are minimized.

There are three key components of the Solaris MC networking subsystem:

- Demultiplexing of incoming packets to the “correct” node: Incoming packets are first received on the node that has the network adapter physically attached to it.

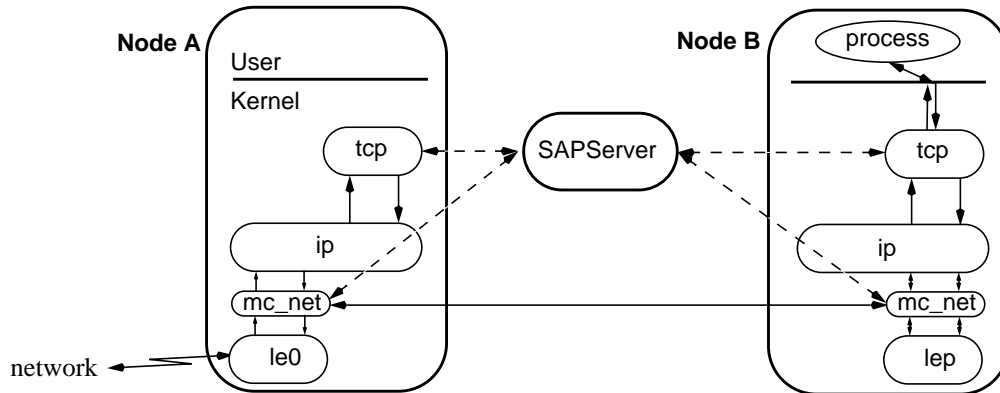


Figure 4. Multi-computer Networking Set-up. The *mc_net* packet filter makes the *le0* network device appear local to the application process. TCP/IP protocol processing occurs on node B, preventing node A from becoming a bottleneck. Solid lines show data traffic and dotted lines show service access port control communication.

The data may, however, be addressed to an application running on a different node. Solaris MC includes an enhanced implementation of the programmable Mach packet filter [14, 25], which extracts relevant information from each packet and matches it against state information maintained by the host system. Once the destination node within the multi-computer system is discovered, the packet is delivered to that node over the system interconnect.

- Multiplexing of outgoing packets from various nodes onto a network device: All protocol processing for outgoing packets is performed on the node on which the endpoint for the network connection exists. The layer that passes data to the device driver makes use of remote device access (transparently) to send data over the physical medium.
- Global management of the network name space: Network services are accessed through a service access point (or sap). (For TCP/IP, the saps are simply ports.) Providing a single system image of the sap name space requires coordination between the various nodes. In Solaris MC, a database that maps service access points to nodes within the multi-computer is maintained by the SAPServer, which

ensures that the same sap is not simultaneously allocated by different nodes in the system.

The structure of the networking system is shown in Figure 4. The *mc_net* module is the packet filter that creates the illusion of a local lower stream corresponding to a remote physical network device in the system. The *mc_net* module is pushed above the cloneable network device driver by the Solaris MC network configuration utilities. The network stack, with the exception of the *mc_net* module, is oblivious of the location of the network device within the multi-computer system. In the figure, the SAPServer is shown independent of a node for clarity; in reality, it is provided on one or more of the nodes of the system.

Solaris MC networking also provides the ability to replicate network services to provide higher throughput and lower response times. This is achieved by extending the API to allow multiple processes to register themselves as servers for a particular service. The network subsystem then chooses a particular process when a service request is received. For example, *rlogin*, *telnet*, and *http* servers are by default replicated on each node. Each new connection to these services is sent to a

different node in the cluster based on a load balancing policy (currently, a simple round-robin load distribution policy). This allows the cluster to be used as an HTTP server, for example, with all nodes handling requests in parallel.

Other features of the Solaris MC networking subsystem are management of global state in the network protocols, such as network statistics maintained for network management agents, and network state information acquired from routers or peers on the network. In the former case, the network management agents are modified to collect information from all the component nodes of the multi-computer, while in the latter case, information collected on any node is broadcast to the other nodes.

6 Communication and Programming Infrastructure

Solaris MC is built from a set of components on top of the Solaris basic kernel. Those components include most OS services, from file system support to global process management and networking management. The programming and communications framework provides support for implementing the components and the communication between components. The framework includes a programming model, a compiler, and run time support for component implementation.

6.1 Programming model

Solaris MC components require a mechanism for accessing them both locally and remotely, and to determine when a component is no longer used by the rest of the system. At the same time, it is essential that each new component have a clearly specified interface, permitting its maintenance and evolution. These two requirements led us to decide on the adoption of an object-oriented approach to the design of Solaris MC.

From the available possibilities we decided to adopt the CORBA [18] object model, as the best suited for our purposes. CORBA is an architecture with mechanisms for objects to make requests and receive responses in a heterogeneous distributed environment, somewhat similar to RPCs. CORBA provides a strong separation between interfaces and implementations. In CORBA, an interface is basically a set of operations, and each object accepts requests for the operations defined by the associated interface. How a given object implements an interface is up to the implementor of the particular object. CORBA also includes reference counting. In order to perform a request on an object, the client code must obtain a reference to that object, allowing the system to keep track of the number of references.

Interfaces are defined by using CORBA's IDL [23]. IDL allows the definition of interfaces by specifying the set of operations the interface accepts (similar to C function declarations), as well as the set of exceptions any given operation may raise. Interfaces can be composed using *interface inheritance* mechanisms, including multiple inheritance. Client and server object implementation code can be written using any programming language for which a mapping from IDL has been established. Currently, there are a few such programming languages, including C and C++. We decided to use C++ as it provided the best match for the CORBA object model.

Every major component of Solaris MC is defined by one or more IDL specified object type. All interactions among the components are carried out by issuing requests for the operations defined in each component's interface. Such requests are carried out independently of the location of the object instance by using our own ORB (*Object Request Broker*), or run time. When the invocation is local (within the same address

space), it proceeds like a procedure call. When the invocation crosses domains (across address spaces or nodes), the invocation proceeds essentially as an RPC, where the client code uses stubs, and the server (implementation) code uses skeleton code to handle the call.

The stubs used by the client code, as well as the skeletons used by the server code are generated automatically from the IDL interface definition by a CORBA IDL to C++ compiler. We currently use a modified version of the Fresco IDL to C++ compiler [13].

6.2 The run time system

The different components of the system communicate using the services of the ORB. The main functions of the Solaris MC ORB are reference counting, marshaling/unmarshaling support, remote request support (RPC), and communication fault recovery. The three main goals of our ORB architecture are to provide an efficient object invocation mechanism, easy configuration of clusters, and support for high availability.

Solaris MC's ORB is composed of three layers: the handler, the xdoor, and the transport layer (Figure 5). Each object reference is associated with a handler. The handler is responsible for preparing inter-domain requests to the object whose reference it handles. A handler is also in charge of performing marshaling of its associated references, as well as of local (to an address space) reference counting. The handler layer implements the subcontract paradigm [21], providing a flexible means of introducing multiple object manipulation mechanisms, such as zero-copy.

In order to perform an invocation on its object, a handler is associated with one or more *xdoors*. The xdoor layer implements an RPC-like inter process communication mechanism. This layer extends and builds on

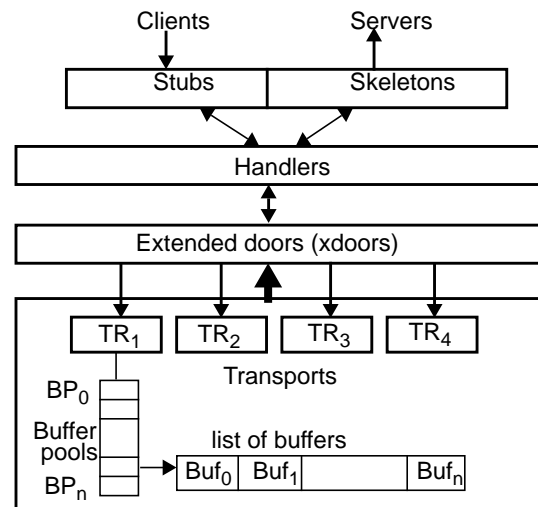


Figure 5. The different layers of the ORB.

the functionality of the Solaris doors mechanism. With *xdoors*, it is possible to perform arbitrary inter-domain (intra- or inter-node) object invocations following an RPC scheme. References to *xdoors* are carried out with invocations and replies, permitting the ORB to locate particular instances of implementation objects. The *xdoor* layer implements a reference counting mechanism for ORB and application structures. Together, the handler and *xdoor* layers support efficient parameter passing for inter-address-space, inter- and intra-node invocations.

The transport layer defines the interface that a transport (such as Ethernet or Myrinet [4]) has to satisfy to be used by the *xdoor* layer. Transports implement reliable *sends* of arbitrary length messages. It is also possible to register receive functions with a transport. A transport has a set of buffer pools with lists of buffers, waiting to be filled by incoming messages. Arriving messages are stored in buffers from the buffer pools specified in the message's headers. Thus, the transport's interface is both simple and sufficiently powerful to support highly efficient object invocation, providing message delivery with scatter and gather capabilities through the buffer pools. A Solaris MC system can

support several ORB transports at the same time. While our main goal is to support closely connected clusters, this capability makes it easy to configure the system to have some long distance links within the cluster.

Together, the three layers provide support for efficient parameter marshaling and passing, making it possible to implement zero-copy schemes for inter-node communications when the communications hardware supports it.

To support high availability, the xdoor layer never aborts an outstanding invocation unless a failure in the cluster is detected. In that case, outstanding invocations to failed nodes are aborted, and an exception is raised which can be caught by the handler layer, or by the component code itself. Services needing high availability make use of the information in the exception either directly or by means of special handlers. In addition to this failure reporting, the xdoor layer implements a reference counting algorithm that can recover after node failures. For efficiency, the algorithm is optimistic in nature, performing most of its work when an actual node failure is detected.

The ORB permits both kernel- and user-level communication. The ORB is implemented as a loadable kernel module to be used by the code residing in the kernel, and as a library, to be used by code executing in a user-level process. Most of the code used in both cases is identical, differing mostly in the xdoor and transport layers. Calls which target objects served by a user-level process are routed through the kernel xdoors.

6.3 C++ in the kernel

All Solaris MC extensions are implemented in C++, and are incorporated into a Solaris kernel as loadable modules. In order to mix the C++ code with the existing kernel, it was necessary to create a special loadable module containing the C++ run time support.

The basic task of the loadable module is to provide some new relocation types to the kernel dynamic linker. It also supports the “new” and “delete” operators, as well as the C++ exception handling mechanism, which in turn is used to implement our ORB’s exceptions.

So far we have had no problems with code size or speed, finding no major difference with C compiled code. We use C++ not only for writing the components defined with the IDL interfaces, but also for any other code performing auxiliary or complementary functions. We make conservative use of C++ features, only using those with sufficient advantages for their cost. Thus, we do not use virtual base classes (the current compiler implementation makes them very space-inefficient), or return objects by value. On the other hand, we find C++ exceptions extremely useful. Exceptions are extensively used throughout our code to signal abnormal conditions and errors.

7 Support for High Availability

Solaris MC integrates the support for high availability into the operating system. Solaris MC divides the responsibility for high availability into several layers: failure detection and membership service, object and communication framework, reconfiguration of system services, and reconfiguration of user level programs. A more complete description of the high availability support in Solaris MC will be described in a forthcoming technical report. Here we provide a brief description of the architecture and some examples.

At the lowest level, a cluster membership monitor (CMM) detects a communication or node failure. The CMM informs the ORB that a cluster reconfiguration is in progress. The CMM then uses a distributed membership protocol to reach a global agreement on the current cluster configuration. Once an

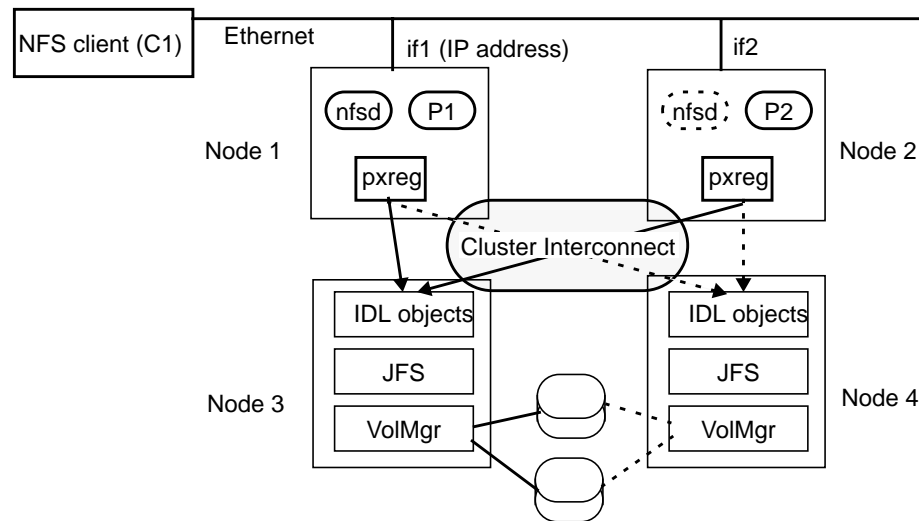


Figure 6. An example of high availability in Solaris MC. The system is configured as a NFS server with redundancy.

agreement is reached, the ORB is contacted. In turn, the ORB invalidates all client xdoors for objects residing in the failed node. Furthermore, the ORB runs a reference recovery protocol which removes all object references held by processes on the failed node.

Distributed system services can learn about a failure in two ways. First, they may get an exception indicating the failure of the node servicing a currently invoked object reference. Second, they can use special handlers which are notified when the xdoor they selected to perform an invocation has been invalidated as a result of a failure. For example, when a node caching file pages crashes, the pager object receives an *unreference* upcall. The pager then cleans up any locks held by the cache, allowing other nodes to access the file.

Figure 6 illustrates one system configuration to make Solaris MC system services highly available. The figure shows a four node system configured as an NFS server connected to an Ethernet. A file system is stored in a disk volume mirrored on two disks. The disks are dual ported and

connected to a pair of nodes. One node is designed as the primary server for the file system (Node 3); the other is a backup (Node 4). A journaling file system is used to minimize file system recovery time. The file system is exported via NFS to remote clients (C1). The file is also accessed by processes P1 and P2 running locally on the cluster. The network interface *if1* is the primary interface for the cluster IP address; *if2* is a backup.

If Node1 crashes, Solaris MC relocates the IP address to *if2* on Node 2. The crash is transparent to P2 and the remote NFS client. If Node 3 crashes, the backup volume manager on Node 4 takes over and recovers the disk volume. The backup IDL objects on Node 4 continue to serve the file system. The crash and takeover are transparent to P1, P2, and any remote client.

A special object handler is used to implement the file objects on Node 3 to facilitate transparent failover to the backup object. The special handler stores enough information in the object reference to perform a switchover from a primary object on Node 3 to a backup object on Node 4. The switchover is transparent to the holder of the

object reference (e.g., a pxfs proxy vnode). In the example we assume that a journaling file system makes changes to the file non-volatile state recoverable. As for the volatile state (i.e., locks and tokens), there are basically three strategies to preserve it across a takeover by the backup node: backup solicits the volatile state from clients during takeover (long takeover), backup has an up-to-date volatile state (high run time overhead), or the volatile state is in stable storage (high run time overhead if no hardware support).

The focus on high availability differentiates Solaris MC from layered cluster software products, such as AT&T LifeKeeper, IBM HACMP 6000, HP MC/Service Guard, and Sun SPARCclusterTM PDBTM. Solaris MC is much easier to configure for high availability than the layered products, making the administration model scalable to a high number of nodes. Tight integration of key high availability protocols (such as the node membership protocol) with the low level communication framework results in faster failure detection and recovery than is achievable in layered systems.

8 Status

Most of the architecture described in this paper has been implemented: communication and object support, including the ORB and object reference counting; C++ support in the kernel; the global file system; most of process management, including remote exec, wait, signals, and the /proc file system; global networking support for TCP/IP, including extensions to the API to allow more than one server to service incoming connections on the same port; access to remote I/O devices, with no modifications to device drivers; a group membership and status monitor; and a set of extensible performance evaluation tools.

The system was initially developed on Solaris 2.4 and has since been moved to Solaris 2.5 with little effort. The prototype is fairly complete—we regularly exercise the system by running parallel makes and multiple copies of a commercial database server.

All implementation work is done in C++, and all new interfaces are defined using IDL. With the exception of a few minor changes to the kernel proper, the bulk of Solaris MC extensions consist of a set of loadable modules or user-land servers.

The hardware prototype currently consists of sixteen dual-processor SPARCstationTM 10 and SPARCstation 20 machines, partitioned into two or more clusters. The developers' workstations act as front-end machines to the Solaris MC cluster. We use a variety of networks as the system interconnect, including 100baseT ethernet and Myrinet [4].

9 Related Work

Solaris MC uses ideas from earlier distributed operating systems such as Chorus Mix, LOCUS, MOS, OSF/1 AD TNC, Spring, Sprite, and VAXclusters. There are, however, significant differences in our approach, compared to previous systems:

- Solaris MC shows how a commercial operating system can be extended to a cluster while keeping the existing application base.
- Solaris MC emphasizes high availability.
- Solaris MC uses an object-oriented design. The system was built with the CORBA object model.
- Solaris MC uses new ideas from Spring, especially filesystem and virtual memory ideas.

For example, unlike systems such as Spring, and Sprite, Solaris MC builds on a commer-

cial operating system while maintaining binary compatibility with a large existing application base. Also, most of the other systems, with the notable exception of VAXclusters, do not emphasize high availability. Finally, Solaris MC introduces object-oriented techniques based on the CORBA object model, and builds on the experience of the Spring system.

10 Conclusions and Future Work

We have built a prototype operating system that provides a single-system image for distributed tightly-coupled hardware systems. The prototype consists of a set of extensions to a commercial operating system. The resultant system thus leverages all existing applications of the OS and enables the OS to extend to a new class of computers. By extending an existing operating system, rather than writing an entirely new one, we were able to leverage off the existing OS code base and device drivers, dramatically reducing the development effort.

We made the early decision to build our system using the Solaris system in order to leverage the large investment in application and system software. Several Solaris features made our job easier, including loadable modules and dynamic linking, both for the kernel and user processes; the basic system VFS and DDI interfaces; the multi-threaded architecture of the system; and the new door IPC mechanism. On the other hand, there are portions of the system that are difficult to extend to a clustered system, including the Streams framework because many existing modules do not adhere to the framework and assume that they are running on a shared memory system. Some UNIX semantics are also difficult (but not impossible) or are inefficient to extend to a clustered system, including POSIX controlling terminal and

sessions semantics, file descriptor sharing, and fork semantics.

Our experience so far with the use of IDL and CORBA to design and implement Solaris MC is very positive. The use of IDL gives us a collection of clearly defined interfaces, and the IDL to C++ translator conveniently creates the glue we need to perform arbitrary service requests from our components. An additional advantage of using IDL is the little effort it took us to adapt Spring technology to Solaris MC.

The structure of Solaris MC ORB allows us to create special handlers to efficiently transmit bulk data and other user-defined structures between nodes. These handlers make use of the transport layer to avoid extra copying of large amounts of data.

On the other hand, we find the CORBA model lacking in two areas. First, there is no straightforward way to perform asynchronous object invocations. Secondly, there is no support for performing group invocations to a set of objects supporting a common interface, a feature that would be useful for the cluster membership monitor. We had to go outside the CORBA model for group multicast.

More work remains for the future. High availability support needs more work. We plan to port a volume manager and provide more support for system administration. We also plan to move to faster interconnect hardware, and to measure and analyze the system performance. Finally, we plan to experiment with heterogenous clusters.

Acknowledgments

We would like to acknowledge Madhu Talluri, Remzi Arpaci, Francesc Munoz Escoi, and Aman Singla for their contributions to the project.

References

- [1] T. E. Anderson, D. E. Culler, D. A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, February, 1995.
- [2] A. Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software—Practice & Experience*, vol. 15(8), August 1985.
- [3] N. Batlivala, et al., "Experience with SVR4 over CHORUS," *Proceedings of USENIX Workshop on Microkernels & Other Kernel Architectures*, April 1992.
- [4] N. J. Boden, D. Cohen, R. E. Felderman, A.E. Kulawik, C. L. Seitz, J. N. Seizovic, W. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, February 1995.
- [5] F. Douglass and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software—Practice & Experience*, vol. 21(8), August 1991.
- [6] Intel Corporation, *Intel Paragon XP/S Supercomputer Spec Sheet*, 1992.
- [7] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993. Also Sun Microsystems Laboratories Technical Report SMLI-TR-93-18.
- [8] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of Winter '93 USENIX Conference*, January 1993. Also Sun Microsystems Laboratories Technical Report SMLI-TR-92-3.
- [9] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-TR-93-09, February 1993.
- [10] Yousef A. Khalidi and Michael N. Nelson, "A Flexible External Paging Interface," *Proceedings of the Usenix conference on Microkernels and Other Architectures*, September 1993. Also Sun Microsystems Laboratories Technical Report SMLI-TR-93-20.
- [11] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of '86 Summer Usenix Conference*, pp. 238-247, June 1986.
- [12] N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-Coupled Distributed Systems," *ACM Transactions on Computer Systems*, vol. 4(2), May 1986.
- [13] Mark Linton and Douglas Pan, "Interface Translation and Implementation Filtering," *Proceedings of the USENIX C++ Conference*, 1994.
- [14] C. Maeda, B. N. Bershad, "Protocol Service Decomposition for High-performance Networking," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.
- [15] James G. Mitchell, et al., "An Overview of the Spring System," *Proceedings of Comcon Spring 1994*, February 1994.
- [16] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi, "A Framework for Caching in an Object-Oriented System," *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, December 1993. Also Sun Microsystems Laboratories Technical Report SMLI-TR-93-13.
- [17] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, September 1993.
- [18] Object Management Group, *The Common Object Request Broker: Architecture*

and Specification, Revision 1.2, December 1993.

[19] J. Ousterhout, A. Cherenon, F. Douglas, M. Nelson, and B. Welch, “The Sprite Network Operating System,” *IEEE Computer*, February 1988.

[20] G. Popek and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[21] G. Hamilton, M. L. Powell, and J. G. Mitchell, “Subcontract: A flexible Base for Distributed Programming,” *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[22] Glenn C. Skinner and Thomas K. Wong, “Stacking Vnodes: A Progress Report.” *Proceedings of the Summer 1993 Usenix Conference*, 1993.

[23] Sun Microsystems, Inc. *IDL Programmer’s Guide*, 1992.

[24] Thinking Machines Corp., *The Connection Machine System: CM-5*, 1993.

[25] M. Yuhara, C. Maeda, B. N. Bershad, J. E. B. Moss, “Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages,” *Proceedings of Winter ‘94 USENIX Conference*, January 1994.

[26] Roman Zajcew, *et al.*, “An OSF/1 UNIX for Massively Parallel Multicomputers,” *Proceedings of Winter ‘93 USENIX Conference*, January 1993.

SpeechActs: A Spoken Language Framework

Paul Martin, Fredrick Crabbe, Stuart Adams, Eric Baatz, and Nicole Yankelovich, (IEEE Computer, Vol. 29, Number 7, July 1996)

Introduction by Paul Martin and Nicole Yankelovich

SpeechActs was a prototype system designed for traveling professionals who required access to online information while they were away from their computer. SpeechActs provided a natural, speech-only interface to a suite of integrated applications, including e-mail and calendar. Other applications provided speech access to dynamic data feeds for weather forecasts, stock quotations, currency exchange data, and international time. Without having to train the system, a user could telephone SpeechActs from an airport, a hotel room, or a colleague's office and speak requests naturally without having to memorize commands. For example, SpeechActs understood phrases such as "I'd like mail please," "How's the weather in Chicago?" or "What's on Bob's calendar the day after tomorrow?"

The SpeechActs paper selected for this collection introduces the SpeechActs Framework and focuses on the techniques used to create a conversational, consistent "sound and feel" across applications. The paper also touches on compiling speech-recognizer and natural language grammars from a single source, dynamically altering grammars for better recognition, and handling recognition errors with medium-grained semantic analysis.

While the SpeechActs project never had a direct impact on Sun's products, it was unique for its time in the speech field. In 1996, when this paper was published, almost all existing speech applications had been built by speech vendors and worked only with the vendor's specific technology. SpeechActs was one of the first large-scale, continuous speech projects that considered recognizers and synthesizers as pluggable components. In addition, the grammar work done as part of the SpeechActs project had a substantial influence on portions of the Java™ Speech API.

SpeechActs never had a large user population, but those that did use the prototype system were heavily studied through laboratory usability tests and longitudinal field trials. This data enabled the team to develop an in-depth understanding of conversational speech user interface design techniques. The lessons learned from this research are still being referenced in the current speech user interface design literature and have had a substantial impact in the field.

"SpeechActs: A Spoken-Language Framework." © 1996 Sun Microsystems, Inc. and IEEE. Reprinted, with permission, from Computer, Vol. 29, Number 7, July 1996 (0018-9162/96).

REFERENCES:

Using Natural Dialogs as the Basis for Speech Interface Design, Nicole Yankelovich, submitted to MIT Press as a chapter for the upcoming book, Automated Spoken Dialog Systems, edited by Susann Luperfoy.

<http://www.sun.com/research/speech/publications/mit-1998/MITPressChapter.v3.html>

How do Users Know What to Say? Nicole Yankelovich, ACM Interactions, Vol. III, Number 6, November/December 1996.

<http://www.sun.com/research/speech/publicationsacm-interactions-1996/Interactions.html>

The "Casual Cashmere Diaper Bag": Constraining Speech Recognition Using Examples, Paul Martin, Proceedings of the Association of Computational Linguistics, Madrid, July 7, 1997.

<http://www.sun.com/research/speech/publications/acl97/cashmere.html>

Office Monitor, Nicole Yankelovich, Cynthia D. McLain, CHI '96 Conference on Human Factors in Computing Systems, Vancouver, British Columbia, Canada, April 14-18, 1996.

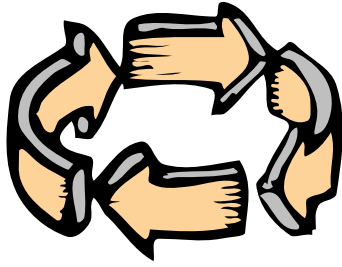
<http://www.sun.com/research/speech/publications/chi96/CHI96.html>

Designing SpeechActs: Issues in Speech User Interfaces, Nicole Yankelovich, Gina-Anne Levow, Matt Marx, CHI '95 Conference on Human Factors in Computing Systems, Denver, CO, May 7-11, 1995. SMLI 94-0394.

<http://www.sun.com/research/speech/publications/chi95/CHI95.html>

Speechacts: A Testbed for Continuous Speech Applications, Paul Martin, Andrew Kehler, Proceedings of the AAAI Workshop on Integration of Natural Language and Speech Processing, pages 65-71, Cambridge, MA, August. 1994 MIT Press.

<http://www.sun.com/research/speech/publications/aaai-workshop94.ps>



SpeechActs: A Spoken- Language Framework

Paul Martin
Frederick Crabbe
Stuart Adams
Eric Baatz
Nicole Yankelovich
Sun Microsystems Laboratories

SpeechActs is a prototype testbed for developing spoken natural language applications. In developing SpeechActs, our primary goal was to enable software developers without special expertise in speech or natural language to create effective conversational speech applications—that is, applications with which users can speak naturally, as if they were conversing with a personal assistant.

We also wanted SpeechActs applications to work with one another without requiring that each have specific knowledge of other applications running in the same suite. For example, if someone talks about “Tom Jones” in one application and then mentions “Tom” later in the conversation while in another application, that second application should know that the user means Tom Jones and not some other Tom. A *discourse management* component is necessary to embody the information that allows such a natural conversational flow.

Because technology changes so rapidly, we also did not want to tie developers to specific speech recognizers or synthesizers. We wanted them to be able to use these speech technologies as plug-in components.

These constraints—integrated conversational applications, no specialized language expertise, and technology independence—led us to a minimalist, modular approach to grammar development, discourse management, and natural language understanding. This approach contrasts with those taken by other researchers working on spoken-dialogue systems. We believe we have achieved a degree of conversational naturalness similar to that of the outstanding Air Traffic Information Systems dialogues,¹⁻³ and we have done so with simpler natural language techniques.

At the same time, SpeechActs applications are unique in their level of speech technology independence. Currently, SpeechActs supports a handful of speech recognizers: BBN’s Hark,⁴ Texas Instruments’ Dagger,⁵ and Nuance Communications’ recognizers⁶ (derived from SRI’s Decipher). These recognizers are all *continuous*—they accept normally spoken speech with no artificial pauses between words—and *speaker-independent*—they require no training by individual users. For output, the framework provides text-to-speech support for Centigram’s TruVoice and AT&T’s TrueTalk. The system’s architecture makes it straightforward to add new recognizers and synthesizers to the existing set.

Like several other research systems, SpeechActs supports multiple, integrated applications. For example, the Chatter system developed at the Massachusetts Institute of Technology⁷ offers e-mail reading, voice mail access, telephone dialing, Rolodex access, and user-activity information. Chatter, however, was created as a single, monolithic system. Applications in a monolithic system can be very tightly integrated, allowing the user’s conversation to flow seamlessly from one activity to another (Chatter: “Your next message is from Lisa.” User: “What’s her address?”).

Users interact with SpeechActs applications using natural, continuous speech, as if they were conversing with a personal assistant.

SpeechActs trades off this type of seamless integration for extensibility. A single, monolithic system cannot be expanded indefinitely without running into grammar size limits, vocabulary conflicts, and unacceptably high *perplexity*. Perplexity (too many equally likely word choices) is a major cause of recognition errors. SpeechActs users, therefore, must explicitly switch between applications rather than change the subject in a more natural, implied manner. Despite this inconvenience, the user's conversational context is maintained across applications, as we illustrate in this article.

HELP FOR BUSINESS TRAVELERS

The current suite of SpeechActs telephone-based applications targets business travelers, letting them read electronic mail, look up calendar entries, retrieve stock quotes, set up notifications, hear national weather forecasts, ask for time around the world, and convert currency amounts. The dialogue below captures the flavor of a SpeechActs conversation. In this example, a business traveler has telephoned SpeechActs and entered his name and password. (We've published a more complete discussion of the design issues and user testing of the SpeechActs conversational interface elsewhere.⁸)

SpeechActs
lets business
travelers read
e-mail, consult
their calendars,
and ask for
time around
the world.

SA: Welcome to SpeechActs. Would you like mail, calendar...
User: I'd like mail please.
SA: You have 47 old messages and 78 new messages; 14 are high priority. The first new message is from Paul Martin regarding "Sailing this weekend."
User: Skip that one.
SA: Message 2 is from Nicole Yankelovich regarding "Change in itinerary."
User: Let me hear it.
SA: "Ric, can you meet me in Paris on the first Friday after New Year's instead of in Brussels next week?"
User: Switch to Calendar.
SA: Switching to Calendar. Today you have "Lunch with visitors."
User: What do I have the Friday after New Year's?
SA: On Friday, January 5th, you have no appointments.
User: How about the day before that?
SA: On Thursday, January 4th, you have "Remember to call florist."
User: What does Sandy have on Wednesday?
SA: On Wednesday, December 11th, at 11:00 Sandy has "Staff Meeting."
User: No, I meant the Wednesday after New Year's.
SA: Sorry. On Wednesday, January 3rd, Sandy has "Peter's Birthday."
User: What's on Tom's calendar the next day?
SA: Do you mean Tom Ricardo or Tom Jones?
User: I mean Tom Jones.

SA: On Thursday, January 4th, at 9:00 Tom Jones has "Meet with architect." From 3:00 to 4:00 he has "Performance review meeting."
User: Switch back to Mail.
SA: Switching back to Mail. You left off with message 2 from Nicole Yankelovich.
User: Send a message to Tom.
SA: Preparing message to Tom Jones...

Notice that SpeechActs weaves feedback into each conversational unit so that users can detect errors. Notice also that the user's reference to Sandy is unambiguous, but the system knows about two different Toms (for this user). SpeechActs remembers which Tom is currently being discussed so that the user doesn't have to identify "Tom" more than once in a session. The sample dialogue also illustrates how the user can correct just a portion of the query when the system selects the incorrect Wednesday.

In addition, this example illustrates the Calendar's extensive use of relative dates. Our user studies revealed that this is how people talk about dates when they do not have a calendar in front of them. Because relative dates are rather tricky from a natural language-understanding perspective, SpeechActs includes a modular date grammar and a discourse specialist for dates. These shared resources let developers of subsequent applications (such as notification and weather) use relative dates ("Give me a reminder call next Monday" or "What's the weekend forecast for Boston?") without addressing the accompanying complex language issues.

Other prewritten grammar modules support irregular verb constructions, user names, place names, numbers, clock times, and money. These grammars can be included in any application. When necessary, corresponding *discourse specialists* handle calculations and track prior conversational references. Besides easing development, using common modules gives the applications a shared "sound and feel."

SYSTEM STRUCTURE OVERVIEW

Figure 1 shows a diagram of information flow in SpeechActs. The framework comprises an audio server, the Swiftus natural language processor, a discourse manager, a text-to-speech manager, and a set of grammar-building tools. These pieces work in conjunction with third-party speech components and the components supplied by the application developer. In this article, we place Swiftus, the discourse manager, and the grammar tools in context. For a more comprehensive architectural overview, see our earlier account.⁸

The audio server presents raw, digitized audio (via a telephone or microphone) to a speech recognizer. When the speech recognizer decides that the user has completed an utterance, it sends a list of recognized words to Swiftus. The speech recognizer recognizes only those words contained in the relevant *lexicon*—a specialized database of annotated vocabulary words.

Swiftus parses the word list, using a grammar written by the developer, to produce a set of *feature-value pairs*. These pairs encode the semantic content of the utterance that is relevant to the underlying application. For example, a calendar application requires pairs such as the following:

```

USERID=pmartin,
DATE=6 January 1996, and
ACTION=appointment-lookup.

```

The feature-value pairs pass through a series of discourse manager *snooper functions*, which scan for pairs that require special action such as requests to end the session or switch to a different application.

If none of the snoopers intervene, the feature-value pairs are passed to the application for processing. As it processes the pairs, the application may ask the discourse manager for help from discourse specialists or access to information about the current conversational context. Among other things, discourse specialists turn relative date references like “the Wednesday after New Year’s” into absolute day-month-year references like “3 January 1996.”

The discourse manager also maintains a stack of information about the current conversation (the *discourse stack*), including data that lets an application resolve references that use pronouns (“Send me a reminder”), deictic references (“What does Eric have tomorrow?”), or partial information (“And Nicole?”). Both the discourse manager and the application can respond to the user by sending a text string to the text-to-speech manager, where it is eventually transformed into digitized audio, which the audio server sends to the user via the telephone, a speaker, or headphones.

GRAMMAR TOOLS

Although any SpeechActs application can recognize and act upon a single defined set of user utterances, there are two grammars that specify the legal set of utterances: A speech recognition grammar determines what words were

spoken, and a natural language processing grammar extracts the meaning from those words. Each grammar has a corresponding engine: A speech recognizer and Swiftus.

Lexicon

The lexicon used by SpeechActs is a computer-readable dictionary containing all the words (in their various forms) that are needed for the current application. Each entry is a word sense, so a word like “bank” might have two noun entries (side of a river and financial institution) plus two verb senses (making a financial transaction and deflecting a ball). If a word sense never occurs in an application, the developer can omit it from that application’s lexicon. Besides its spelling, each word sense has a set of feature-value pairs that defines such things as its part of speech (noun, verb, and so on), its root form (“was” and “am” are both forms of “to be”), and other information useful to the application (such as semantic features like “clothing” for the word “shirt”). The lexicon may contain additional features, such as defining synonyms (instead of creating the entry for each separately) or specifying when a word changes its form irregularly (for example, the plural of “child” is “children”). The developer may also explicitly include other lexicons so that an application can easily use previously developed lexicons for special purposes.

An example entry for the word “show” in a calendar application might look like this:

```

(show
((category verb) (root show)
(semantic display)
(irregular (past-participle shown)))

```

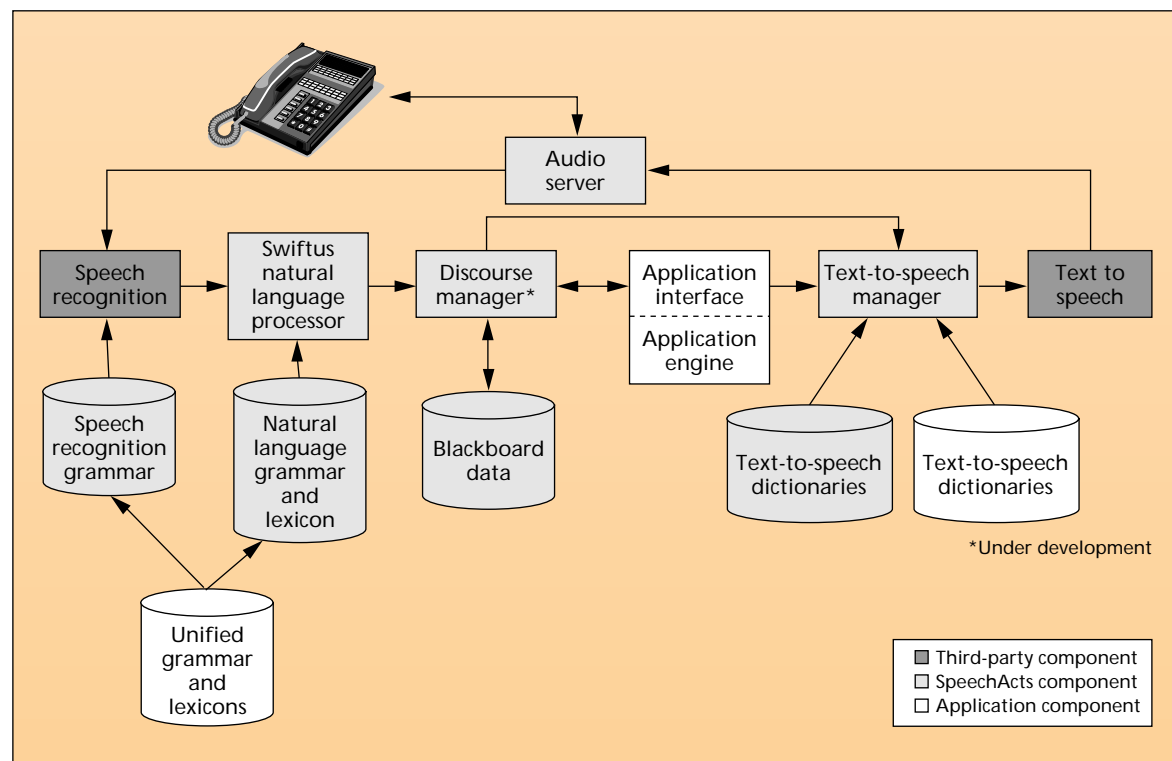


Figure 1. Diagram of information flow in SpeechActs.

From this entry, the lexicon loader would automatically produce “shows,” “showed,” and “shown” as derivative forms. Because the speech recognizer requires a full word list, the *morphology* must be done in advance, not on the fly.

Unified Grammar

Current continuous-speech recognizers require grammars that specify every possible utterance a user could say to the application. The constraints on word choice imposed by these grammars reduce perplexity and thus lower the recognition error rate. Recognizer grammars are commonly specified using Backus-Naur Form, yet the details of each formalism vary widely. To substitute one recognizer for another, developers must write a new version of the grammar. Worse still, for the Swiftus semantic grammar to handle the same user utterances as the recognizer, it must be closely synchronized with the recognizer grammar.

We solved this problem by inventing a Unified Grammar,⁹ which lets a developer write a single, recognizer-independent grammar specification for a SpeechActs application.

UNIFIED GRAMMAR RULES. Along with its lexicon, a Unified Grammar is a collection of rules. The typical Unified Grammar rule consists of a *pattern*—such as a BNF rule—followed by *augmentations*, which are statements written in a Pascal-like form. Augmentations take the form of

- *head declarations*—an easy way to pass groups of feature-value pairs;
- *tests*—further constraints on the matches based on the features of the pattern elements; and
- *actions*—a means to pass feature values out of the parse.

Here is a commented example of a Unified Grammar rule that matches utterances such as “What is on Nicole’s calendar?”

```
{CalendarQuestion := # PATTERN

    "what" root=be ("in" | "on")
    namePossessive sem=calendar;

                                # AUGMENTATIONS
                                # Head Declaration
    head namePossessive;
                                # Tests
    be.past-participle != t;
    be.ing-form != t;
                                # Actions
    action := `lookup; }
```

Each word of the pattern matches a word (or compound word) of the utterance. The pattern requires the first word to be “what,” then any form of “to be,” then “in” or “on.” The fourth word is whatever results from a rule called *namePossessive*, and the last is any word in the lexicon with the semantics of “calendar,” such as “schedule.” The first augmentation sets the head of the rule (its main source of information) to be the results from the *namePossessive* rule. Then, the tests ensure that the verb is not “being” or “been.” The “be” forms we want are “was,” “is,” “are,” and “_s” (for a contraction), and so on. The action augmentation adds a feature “action” with the value of “lookup.”

UNIFIED GRAMMAR COMPILER. From a Unified Grammar, the Unified Grammar compiler produces a grammar for a specified speech recognizer and a corresponding grammar for Swiftus. The major complication is that the recognizer and Swiftus grammars are fundamentally different from one another. Recognizer grammars, used only to constrain word sequences, explicitly represent constraints down to the level of all possible terminals, while Swiftus grammars, in order to extract a data structure comprising the semantic contents of a full parse of the sentence, must be more general and use augmentations. The Swiftus grammar is a simple transformation of the Unified Grammar, so we will focus instead on how the compiler produces the speech recognizer grammar.

Reducing a grammar containing arbitrary augmentations to pure BNF would require precomputing all those augmentations, which is provably not possible. However, a tool to reduce augmented grammars is so useful that we have created an engineering approximation. The augmentations are composed of tests that commonly compare pattern element features to one another or to constant values, and actions that set feature values, either by copying from lexical feature values or through more complex computations.

The Unified Grammar compiler works by rewriting the augmented rules and the feature-based patterns using only words from the lexicon or patterns that correspond eventually to some set of these words. While a top-down expansion to all the allowed word sequences would fit this description, the compiler also strives to maintain a compact form for an expression without losing its constraining power.

Retaining constraints is the purpose of the compiler; otherwise a single trivial rule such as `sentence := word*` could cover every grammar. So whenever case analysis of the actual entries in the lexicon can yield a tight BNF rule to replace an augmented one, it is used. For example, a rule like

```
ToBeVerb := root=be
```

could be rewritten after searching the lexicon for all such words as

```
ToBeVerb := "be" | "_s" | "is " | "am " |
             "as" | "were" | "being " |
             "been"
```

When the tests refer to properties that have been set by actions in other rules, the compiler must pursue those

Current continuous-speech recognizers require grammars that specify every possible utterance a user could say to the application.

rules to determine the conditions that cause them to set these values. Whenever the code analysis becomes too complex, or when the augmentation contains an explicit indicator that further analysis is forbidden, the compiler produces a rule that allows the pattern portion but does not impose the restrictions represented by the unanalyzable actions or tests. This is the escape hatch that makes it possible to process any Unified Grammar.

The new rules created by rewriting are given descriptive names that allow the compiler to reuse rules that would otherwise be created more than once in a compilation. For example, if a portion of some pattern allowed any word whose root form was “do,” a substitute rule that simply collected all the winners from the lexicon would be written

```
eq_root_do := “do” | “does” | “did “ |
             “done” | “_d”
```

If an additional requirement for only past-participle forms were added as a test augmentation

```
do.past-participle = t
```

then a further restricted rule would be created, using only the allowed word:

```
eq_root_do_eq_past-participle_t :=
    “done”
```

Other cases where an equivalent restriction arose would use this same rule, recognizing it by the name-construction rules.

Swiftus

Swiftus, the natural language processor, was designed to meet several competing objectives.

- *Real-time performance.* The semantic representation must be generated in real time to facilitate conversation.
- *Accurate understanding.* Methods such as simple keyword matching cannot be used because they miss nuances needed to make the proper response.
- *Toleration of misrecognized words.* State-of-the-art speech recognition systems have a significant error rate, especially for structural words, which may be short and/or deemphasized. For example, traditional natural language understanders place too much credence on the distinction between the words “the” and “a.”
- *Wide variation among applications.* Different applications require different knowledge and different strategies. An inflexible natural language processor will not work well with some applications.
- *Ease of use.* Someone besides the developers must find it straightforward to create new applications.

To meet these objectives, Swiftus is designed to perform flexible, *medium-grained* semantic analysis, somewhere between coarse keyword matching and full, in-depth semantic analysis.

SWIFTUS OPERATIONS. Swiftus uses the lexicon supplied for the Unified Grammar and the rule set produced by the Unified Grammar compiler. It is implemented as a pattern recognizer, and it applies the augmented context-free grammar using a top-down, finite-state-automata parser.

The patterns the Unified Grammar compiler has made from the rules are quadruples: pattern name, pattern body, tests, and actions. Patterns are represented as Lisp expressions. Swiftus converts the word list from the speech recognizer into a sequence of word senses. It then parses this sequence by recursively expanding the rule quadruples until at least one sense of each word has been consumed. The rules are efficiently represented as a finite-state automata, in which traversing an arc to another state requires matching the patterns and passing the tests.

At each state, the pattern body is compared with the input to find all matches. If a match occurs, Swiftus then tries the test expressions as additional match requirements, testing such things as the word root, part of speech, or the value of a specified feature in the lexical entry. When both the pattern and the test expressions have passed, Swiftus evaluates the action expressions and uses their values to produce the result form.

USING SWIFTUS. Each Swiftus grammar is designed to handle just one application, so when the user changes applications, Swiftus is reset to the relevant lexicon and Swiftus grammar.

By performing a full parse but limiting semantic analysis

to simple feature-value pairs and a limited number of grammar rules, Swiftus can complete its analysis quickly yet preserve the essential content. Because the matching need not be restricted to specific words, a well-designed grammar can successfully extract the semantics despite many common speech misrecognitions. For example, if the recognizer mistakes one

If the recognizer mistakes one preposition for another, the grammar can be written so that the sentence will still be understood.

preposition for another, the grammar can be written so that the sentence will still be understood. This choice of how specific to make the grammar’s tests lets an application writer create widely ranging grammars, from quite simple to fairly sophisticated. Finally, since the lexicon and grammars are switched with the applications, each can be fine-tuned to respond best for its particular application without interfering with the sharing of grammars by applications.

DISCOURSE MANAGEMENT

Using just Swiftus to process what a user said, an application would succeed only if the user fully specified a command every time he spoke (“Show the calendar for Eric Baatz on January 4, 1996.”) To support more natural speech, we must provide at least rudimentary discourse management.

SpeechActs provides a discourse manager to keep track

of some of what has already been said. The discourse manager tracks the conversation's current structure, using a simplified version of the theories of discourse-segment pushing and popping.¹⁰ To provide a more natural conversation, we use additional techniques,¹¹ including prompt design and error-correcting mechanisms.

Application-level discourse

At the coarsest level, a discourse is represented as a data structure consisting of functions for handling user input. The discourse manager maintains a stack of these structures, and the top one handles the default discourse for the current application. When an application deems it necessary to enter a new dialogue, it informs the discourse manager, which pushes a structure for the new dialogue onto the discourse stack.

For example, when SpeechActs starts, a small application called Login is started by pushing its discourse structure onto the stack. After logging in, the user chooses a “real” application, which is pushed onto the discourse stack. After Swiftus has processed a user utterance and the resulting feature-value pairs are passed to the discourse manager, the pairs are sent to the functions identified by the dialogue structure at the top of the discourse stack (unless a discourse manager snooper claims the utterance). When the discourse manager is told that the current dialogue has been resolved, it pops it off the stack, leaving the underlying dialogue at the top. Additionally, there are ways to abort the current discourse, and they may pop more than one discourse structure from the stack. New discourse structures are started explicitly by the current application or implicitly by the discourse manager to support the current discourse. Consider this example dialogue:

At the coarsest level, a discourse is represented as a data structure consisting of functions for handling user input.

SA: Please state a currency which you'd like conversion to.
User: What's the rate for the franc?
SA: Do you mean Belgian, French, or Swiss?
User: Swiss.
SA: There are 1.15 Swiss francs in one dollar or .87 dollars in one Swiss franc.

In this example, when the user asks for the franc's conversion rate, the currency exchange application does not have enough information to reply, so it asks the discourse manager for disambiguation. If previous context established which franc, then the specialist for this disambiguation would just insert the needed information. In this example, however, the specialist asked the user a question and then pushed a new discourse structure chosen for resolving countries onto the discourse stack. When the user replies, the feature-value pairs are not returned to the currency exchange application, but to the handler for the disambiguation discourse. If the user's response resolves the ambiguity, the new information is passed on to the

application and the disambiguation discourse is popped. Otherwise, functions within the disambiguation discourse structure will try to elicit the information from the user until the issue is resolved. The discourse structure will eventually be popped off the stack when the ambiguity is resolved or when the user cancels that activity.

Discourse across applications

The other main form of discourse management depends on the cross-application context stack. To allow the conversations with SpeechActs to feel more natural, we keep a simple stack of referenced items so that the system seldom needs to rely on entering a subdialogue.

The stack is currently implemented as an ever-growing list, although a more sophisticated model will be needed when we attempt to support longer conversations. As a particular concept is being discussed, the application can push the concept onto the top of the stack. Items can be pushed on as single lexical entries, which are later expanded into feature-value pairs, or as feature-value pairs representing a sentence or phrase. The context stack stays current between various applications, so representing the ideas as feature-value pairs allows flexible access by applications other than the one that pushed them on. Because it is not always clear which of several items should take precedence, multiple items can be pushed onto the same level of the stack. When an application needs to resolve a reference, it asks the discourse manager to search the context stack for the most recent entry of a particular type—that is, the entry with a feature matching one of a set of possible values. If one or more items on the same level match, all those that matched are returned.

Continuing with the preceding example, when the user asks for a conversion between Swiss and Belgian francs, both Switzerland and Belgium are placed as locations on the top of the stack. When the user asks for the rate “there,” the reference is ambiguous, and the application asks the discourse manager to search the stack for a place. Both Switzerland and Belgium are returned, so the situation remains ambiguous. But now, when the disambiguation discourse is entered, the choices are more limited:

User: How about 1 franc in Belgian francs?
SA: There are 25.58 Belgian francs in 1 Swiss franc.
User: How much is 57 dollars there?
SA: Do you mean Belgium or Switzerland?
User: Belgium.
SA: There are 1,450.65 Belgian francs in 57 U.S. dollars.

After this ambiguity is resolved and the feature-value pairs for the Belgian franc are put onto the top of the context stack, a later call from the International Time application for a location can easily extract “Belgium” as the most recent geographic reference on the stack:

User: What time is it there?
SA: The current time in Brussels, Belgium, is 9:46 in the morning.

CHALLENGES

Modular, reusable grammars, coupled with discourse specialists and the simple feature-value pair knowledge representation, have allowed SpeechActs application developers to create dialogues that flow naturally. However, some substantial issues still need to be addressed before SpeechActs can simulate a human conversation convincingly. The more tractable of these issues include pacing, error recovery, and defining the functional boundaries of an application.

Conversational pacing

Pacing is perhaps the least obvious challenge. Humans attach meaning to pauses in a conversation, and current speech systems produce pauses that are inappropriate by human convention. Although the speech recognizers used with SpeechActs have approximately real-time performance, their response delays are sometimes long enough to disrupt conversational pacing. Worse, the delays are not consistent. A longer utterance or a more complex grammar requires more processing time. The resultant pacing gaps signify either that the system is still doing recognition or that the user has not been heard,¹¹⁻¹² but users are not good at guessing which. Besides the obvious solution of somehow speeding up recognition, we are considering the use of nonspeech audio cues to fill pauses. For example, if a certain sound or musical motif were used to let the user know that the system is working, then silence would unambiguously mean that the system had not heard the user.

Explicit error corrections

Recognition errors are quite common, partly because users sometimes speak too soon, causing the first part of their utterance to be clipped, and partly because some sequences sound similar. We have tried very hard to make the system robust in the face of these errors,¹¹ but more work is required. First, the sort of user-initiated error correction illustrated in the example at the beginning of this article was handcrafted by the Calendar developer. A desirable addition to the discourse services would be a generic error-correction specialist that any application could call on; this would both ease the developers' burden and increase the consistency of error-correction behavior across applications.

Such a specialist should allow:

- *Full-replacement corrections*: "I said how much is that in French francs?"
- *Partial-replacement corrections*: "No, I meant this Wednesday."
- *Elimination of options*: "No, I didn't mean that one."
- *Undoing of state-changing mistakes*: "Undo that" or "Oops! Go back."
- *Probing the system state*: "What was I doing?" or "Where were we?"

User prompting

Another issue in designing dialogues is how to let users know an application's boundaries. A speech-only application resembles a command-line interface in that it hides the application's functionality. Speech is too slow an out-

put channel to give users extensive spoken help. In lieu of help, in our applications we have tried to establish a common ground with the user that suggests possible next utterances. While this approach has proven quite effective in getting users to speak legal utterances, it falls far short of letting them know the range of legal utterances possible at a given moment. Currently, new users are given reference cards that list example utterances for each application. Though helpful, this is not a satisfactory solution.

A different approach to user help involves using fairly lengthy prompts, initially, to teach users an application's functionality and options. As the user gains experience, we

progressively shorten the prompts and provide less detail (both within a single session and across sessions). For example, the first time a user records an outgoing e-mail message, the prompt might be "Begin recording your message after the tone. Pause for several seconds when done." The second time, the prompt might be shortened to "Record then pause." If these prompts

SpeechActs is both a proof of concept and an effective system that about a dozen people now depend upon when they travel.

were played in conjunction with an audio cue, eventually the cue alone would suffice.

To help application designers provide this tapered prompting, we provide functions to keep track of which elements in a list of progressively shorter prompts have already been used. To further automate tapering, we need a mechanical way to derive shorter prompts from longer ones, and the record keeping to remember not just that a user is "experienced," but exactly what parts of the system he has mastered.

AS AN EXISTING SET OF APPLICATIONS, SpeechActs is both a proof of concept and an effective system that about a dozen people now depend upon when they travel. Powerful enough to be useful it is easy to use with little training.

As a framework for building speech applications, SpeechActs' contributions include the Unified Grammar to create synchronized grammars for speech recognition and semantic parsing, reusable plug-in speech components, and the Swiftus natural language processor. SpeechActs also includes important discourse management techniques. Both the discourse stack and a simple context queue in SpeechActs model the current state of the discourse so that SpeechActs can respond naturally. These simple, straightforward components combine to make SpeechActs a powerful framework in which to design speech applications. ■

Acknowledgments

SpeechActs has been a group effort. We thank Andy Kehler, Gina-Anne Levow, Matt Marx, and Cynthia McLain for their contributions to the design and implementation,

and especially Bob Sproull for his support and architecture ideas.

References

1. M. Bates et al., "The BBN/HARC Spoken Language Understanding System," *Proc. Int'l Conf. Acoustics, Speech, and Signal Processing*, Vol. II, IEEE Press, Piscataway, N.J., 1993, pp. 111-114.
2. W. Ward and S. Issar, "Recent Improvements in the CMU Spoken Language Understanding System," *Proc. Human Language Technology Workshop*, Morgan Kaufmann, San Mateo, Calif., 1994, pp. 213-216.
3. V. Zue et al., "Pegasus: A Spoken Language Interface for On-Line Air Travel Planning," *Proc. Human Language Technology Workshop*, Morgan Kaufmann, San Mateo, Calif., 1994, pp. 201-206.
4. G. Smith and M. Bates, "Voice Activated Automated Telephone Call Routing," *Proc. Ninth IEEE Conf. Artificial Intelligence for Applications*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 143-148.
5. C. Hemphill, "DAGGER, Directed Acyclic Graphs of Grammars for Enhanced Recognition," *Users Guide and Reference Manual*, tech. report, Texas Instruments, Dallas, Tex., 1993.
6. V. Digalakis and H. Murveit, "Genomes: Optimizing the Degree of Mixture Tying in a Large Vocabulary Hidden Markov Model Based Speech Recognizer," *Proc. Int'l Conf. Acoustics, Speech, and Signal Processing*, IEEE Press, Piscataway, N.J., 1994, pp. 1,537-1,540.
7. E. Ly, *Chatter: A Conversational Telephone Agent*, master's thesis, Massachusetts Inst. of Technology, Cambridge, Mass., 1993.
8. N. Yankelovich and E. Baatz, "SpeechActs: A Framework for Building Speech Applications," *Proc. AVIOS 94*, American Voice Input/Output Soc., San Jose, Calif., 1994, pp. 179-188.
9. P. Martin and A. Kehler, "SpeechActs: A Testbed for Continuous Speech Applications," *Proc. AAAI-94 Workshop on Integration of Natural Language and Speech Processing*, MIT Press, Cambridge, Mass., 1994, pp. 65-71.
10. B. Grosz and C. Sidner, "Attention, Intentions, and the Structure of Discourse," *Computational Linguistics*, July-Sep. 1986, pp. 175-204.
11. N. Yankelovich, G. Levow, and M. Marx, "Designing SpeechActs: Issues in Speech User Interfaces," *Proc. CHI 95*, Addison-Wesley, Reading, Mass., 1995, pp. 369-376.
12. L. Stifelman, "VoiceNotes: A Speech Interface for a Hand-Held Voice Notetaker," *Proc. InterCHI 93*, ACM Press, New York, 1993, pp. 179-186.

Paul Martin is co-principal investigator of Sun Microsystems Laboratories' Speech Applications Project, where he designs and builds tools and prototypes that use state-of-the-art speech recognition systems to "understand" natural language. Martin has worked on the problem of communicating with computers for two decades, at the Stanford AI Lab, Xerox Parc, SRI, MCC, and IBM. Martin received a PhD in artificial intelligence from Stanford University, and a BS in electrical engineering and computer science from North Carolina State University.

Frederick Crabbe is a PhD candidate in artificial intelligence and cooperative, multiagent natural language at the University of California, Los Angeles, and an intern at Sun Microsystems Laboratories' Speech Applications Project, where he develops models and tools for discourse in SpeechActs. He has worked at the US Air Force's Rome Laboratories and Los Alamos National Labs. His interests include neural network models of natural language processing and of general cognition. Crabbe received an MS in computer science from UCLA and an AB in computer science and philosophy from Dartmouth College.

Stuart Adams is a consultant who integrates speech, telephony, and natural language processing technology to build speech-aware applications. He has been involved in the development of various speech applications for other companies, including the development of software to provide phone access to daily newspaper text for the blind and the development of hardware and software for a speech-enabled palm-top computer. Adams received an MS in computer science from Penn State University.

Eric Baatz is a staff engineer at Sun Microsystems Laboratories' Speech Applications Project, where he programs the SpeechActs Framework and applications. He has held a variety of technical and managerial jobs at BluePoint Technologies, Cadre Technologies, Mosaic Technologies, Computer Corporation of America, and Digital Equipment Corporation. Baatz received an MS in computer science from Northwest University.

Nicole Yankelovich is co-principal investigator of Sun Microsystems Laboratories' Speech Applications Project, where she has project management responsibilities and designs speech user interfaces. She has worked on user-interface design in the context of an integrated, multiuser hypertext system at Brown University's Institute for Research in Information and Scholarship (IRIS). She has published a variety of papers on hypertext, user-interface design, and speech applications, and she has served on the organizing and program committees of conferences such as Hypertext, CHI, UIST, ASSETS, and CSCW.

Contact Martin at Sun Microsystems Laboratories, 2 Elizabeth Dr., Chelmsford, MA 01824; paul.martin@east.sun.com; for more information and references, see <http://www.sunlabs.com/research/speech>.

Early Experiences with Persistent Java

Mick Jordan

Introduction by Mick Jordan

This paper was presented at the First International Workshop on Persistence and Java, which was the first of three workshops on this theme that were organized by the Forest research group at Sun Labs in collaboration with Professor Malcolm Atkinson's group at Glasgow University, Scotland. The goal of these workshops was to bring together researchers in the field of persistence in the context of the exciting and dynamic environment of the evolving Java™ platform.

The research collaboration between Sun Labs and Glasgow was designed to apply and test the hypothesis that the persistence of state in the Java platform would best be provided as an orthogonal property, transparent to the programmer, in contrast to the traditional layered approaches through persistence-specific APIs. The collaboration, which began in October 1995, continued for five years during an era of intensive development of the Java™ platform. Although orthogonal persistence is in essence a very simple concept, it proved surprisingly difficult to communicate. Towards the end of the project we settled on the more descriptive concept of "continuous computation," an idea that resonates well with the requirements of reliability and availability. The main significance of the research was the application of orthogonal persistence to a mainstream programming language and the prospect of adoption beyond the laboratory.

The paper reports on the initial experiences of the very first prototype of orthogonal persistence, based on the JDK™1.0.2 virtual machine, built at Glasgow and delivered in June of 1996, on a sample of available applications, including the Java compiler and a web server. The fact that these applications ran, essentially unchanged, was a very positive result. However, the paper hints at the myriad of details that still needed to be solved before the promise of transparency could truly be realized. Handling these details in the face of the extraordinary pace of the Java platform development occupied much of our time in the ensuing years.

Technically, the research did eventually succeed in building a very high performance platform that provided almost complete transparency. Unfortunately, we were unsuccessful in persuading the Java community that orthogonal persistence should become a part of the platform. The desire for orthogonality is recognized by many people, especially developers, but our efforts ultimately foundered on the virtual machine modifications that are required for a complete solution. The consequence in the platform today is layered solutions with increased complexity for the developer. This is currently deemed an acceptable price but it remains to be seen whether this trade-off will change in the future.

REFERENCES:

An Orthogonally Persistent Java, M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis and S. Spence, ACM SIGMOD Record, 25(4), December 1996.

Proceedings of the First International Workshop on Persistence and Java, M.P. Atkinson and M.J. Jordan, editors, Sun Microsystems Laboratories Technical Report TR-96-58, November 1996.

Proceedings of the Second International Workshop on Persistence and Java, M.P. Atkinson and M.J. Jordan, editors, Sun Microsystems Laboratories Technical Report TR-97-63, December 1997.

Advances in Persistent Object Systems – Proceedings of the Eighth International Workshop on Persistent Object Systems and the Third International Workshop on Persistence and Java, R. Morrison, M.J. Jordan and M.P. Atkinson, editors, Morgan Kaufmann, August 1998.

Orthogonal Persistence for Java – A Mid-Term Report, M.J. Jordan and M.P. Atkinson. In Morrison et al., pages 335–352.

A Review of the Rationale and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform, M.P. Atkinson and M.J. Jordan, Sun Microsystems Laboratories Technical Report TR-2000-90.

Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine, B. Lewis, B. Mathiske, N. Gafter, Sun Microsystems Laboratories Technical Report TR-2000-93.

Orthogonal Persistence for the Java Platform – Specification and Rationale, M.J. Jordan and M.P. Atkinson, Sun Microsystems Laboratories Technical Report TR-2000-94.

First International Workshop on Persistence and Java

Mick Jordan
Malcolm Atkinson

SMLI TR-96-58

November 1996

Abstract:

These proceedings record the First International Workshop on Persistence and Java, which was held in Drymen, Scotland in September 1996. The focus of this workshop was the relationship between the Java languages and long-term data storage, such as databases and orthogonal persistence. There are many approaches being taken, some pragmatic and some guided by design principles. If future application programmers building large and long-lived systems are to be well supported, it is essential that the lessons of existing research into language and database combinations are utilized, and that the research community develops further results needed for Java.

The initial idea for the workshop came from Malcolm Atkinson, who leads the Persistence and Distribution Research group at Glasgow University. The idea was one of the first fruits of the collaborative research program that was initiated between Sun Microsystems Laboratories (Sun Labs) and the Glasgow group in the fall of 1995. Sun Labs sponsored the workshop to cover the attendees' local costs



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:

mick.jordan@eng.sun.com
malcolm.atkinson@eng.sun.com

Early Experiences with Persistent Java™

Mick Jordan

mick.jordan@Eng.Sun.COM

*Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043 USA*

Abstract

This paper reports on initial experiences with an orthogonally persistent variant of the Java platform, called Persistent Java (PJava). We review and reflect on the design of PJava and discuss compatibility with Java. The features and limitations of an initial prototype are discussed. The experiences gained in running four distinct applications on PJava are described in detail.

1 Introduction

Persistent Java (PJava) [1], [2] is an experimental persistent programming environment for the Java programming language. PJava provides orthogonal persistence [3], an approach that provides equal rights to persistence for all data types with the absolute minimum of additional programming effort. PJava attempts to support a wide range of applications, ranging from simple, almost completely transparent, uses of persistence to sophisticated applications that require extensible transaction models. A major goal of PJava is for arbitrary Java code to participate unchanged in PJava applications.

We report on our early experiences with the first prototype of PJava, called PJava₀, which is based on the Sun Microsystems™ Java Development Kit (JDK), release 1.0.2 [4]. PJava₀ was designed and implemented by the PJava team at Glasgow University, with support provided by Sun Microsystems Laboratories under its Collaborative Research program.

We begin by providing a brief overview of the design of PJava, followed by a description of the PJava₀ prototype. We then reflect on some of the design choices of PJava and comment on how they relate to the specification of the Java platform, which has been evolving in parallel with the PJava effort. Following a discussion of the limitations of PJava₀ we describe some common pitfalls that beset programmers who are unfamiliar with orthogonally persistent systems.

We then discuss four applications that have been run in the PJava environment, focusing on suitability, ease of

programming, performance and extra functionality.

2 PJava Design

In this section we provide a brief overview of the PJava design. PJava provides orthogonal persistence for Java, according to the following principles:

- **Orthogonality:** The principle of orthogonality states that all values whatever their type have equal rights to persistence - longevity or brevity. PJava applies this principle to all Java types, *including* Class types.
- **Persistence Independence:** The principle of persistence independence states that all code should have the same form irrespective of the longevity of the data on which it acts. PJava accepts arbitrary, unmodified Java code.
- **Persistence Identification:** The principle of persistence identification says that there should be a straightforward and consistent mechanism for determining the longevity of values. In PJava the mechanism is persistence by reachability from *root* objects explicitly registered with the class `PJavaStore`.

PJava applications are associated with a store, represented to the programmer as an instance of the `PJavaStore` class. An application is initiated by associating the PJava interpreter with a store and a class to invoke; the latter may be omitted if the application has previously registered the initial class with `PJavaStore`. Objects are candidates to become persistent if they are reachable, directly or indirectly, from root objects that are explicitly registered with `PJavaStore`. By default, successful termination of a PJava application causes all objects reachable from the roots to be atomically and durably updated in the store. Applications that terminate exceptionally make no changes to the stable store. It is possible to call `PJavaStore.stabilizeAll` during program execution in order to achieve a global stabilization. It is the responsibility of the application to ensure

that a stabilization reflects a semantically consistent state.

An application can register callback methods with `PJavaStore` that will be called prior to the execution of the initial class. These can be used for a variety of application specific tasks, for example, verifying global consistency constraints.

Objects in a store are persistently bound to their associated class, including its bytecodes. Class name lookup is always performed first in the store and only if that fails are the normal mechanisms of the Java system employed.

In addition to global stabilization, PJava provides a rich and extensible transaction interface through the `TransactionShell` class. To simplify the programmer's task, a variety of standard transaction models are provided as specializations of this class, for example, traditional flat transactions and nested transactions. The extensibility is provided as a collection of modular building blocks, for example lock management, that can be composed to form new transaction kinds.

The nature of a PJava store is implementation dependent. However, it will usually be a file or disk partition. A PJava interpreter may execute with a store, called *store mode*, or without a store, called *non-store mode*.

3 Reflections on the PJava Design

In this section we report on our experiences with those aspects of PJava that are independent of the `PJava0` implementation.

3.1 Programming Interfaces

From the programmer's perspective, the bulk of the PJava design is concerned with the extended transaction model. Since this is not implemented in `PJava0`, we cannot report experiences with it. The only other class that is visible to programmers is `PJavaStore` and our experiences with that have led to some redesign. The most significant change relates to support for multiple stores. Initially, there was a notion that a PJava application might be able to access multiple stores, represented by multiple instances of the `PJavaStore` class. For the reasons described in section 7.1, this has been abandoned. Other minor usability changes have been made.

3.2 Persistence Identification

PJava identifies persistent objects by reachability from named roots registered with the `PJavaStore` class. This requires additional code to be written and also requires that this namespace be managed by the applica-

tion. Another approach, which would be more transparent, would be to define the roots as the set of `Class` instances in the virtual machine, this set being implicitly defined from the classes loaded by the interpreter over time.¹ In this approach, class fields declared as static would be viewed as fields of the corresponding `Class` object, and would provide an implicit persistent root table. By associating appropriate semantics for the `static transient` combination² of modifiers, some of the design issues discussed in [1] surrounding static initialization could be avoided. On the other hand, failure to use `transient` appropriately would cause data to be made quietly persistent. As we shall see later, the PJava design choice turns out to be very pragmatic for `PJava0`.

3.3 Compatibility with Java

We would like the PJava interpreter to execute "normal" Java applications unchanged. By normal, we mean those applications that make no use of the `PJavaStore` or `TransactionShell` classes. In other words a PJava interpreter running in non-store mode should pass all of the standard compliance tests. A minimal requirement for this is that PJava accept standard class files. In fact, `PJava0` accepts unmodified class files generated by a normal Java compiler and requires no pre-processing or post-processing steps. When run in non-store mode `PJava0` preserves the semantics of the language and the standard packages. At the time of writing we have not tested this compatibility extensively. However, some significant applications such as the Java compiler and the Jeeves HTTP server [5] execute correctly under the `PJava0` interpreter.

Compliance in store mode is a more subtle issue. The extent to which the specification of Java or the virtual machine applies to store mode is unclear, since the specification is not attempting to define an orthogonally persistent system. Certainly, the PJava behavior differs in some cases and could be interpreted as being non-compliant. We will discuss these cases below. It must be noted, however, that the PJava design predates the existence of the detailed language specification [6], which was not published until August 1996.

3.3.1 Static Class Data

Currently, the specification is silent on the `static transient` combination of modifiers. Indeed the

-
1. This would require a separate mechanism to remove unused classes.
 2. This was called out as illegal in an earlier language specification. The current version does not, but the Java compiler in JDK 1.0.2. still reports it as an error.

meaning of transient is only defined in general terms. However, the natural interpretation would be to truncate the reachability analysis at `static transient` variables in the associated class instances.

3.3.2 System Properties

The properties table maintained by the `System` class is effectively global static data. The semantics of this data in the face of persistence is unclear and can cause compatibility problems. Certain properties are clearly transient in the sense that they relate to the execution of a *particular* virtual machine and to external values that might change between executions. Currently the rule that static data be persistent by default causes the properties table to be made persistent, which means that these transient properties are not reinitialized. However, although it is a dubious programming practice, some applications, for example the Java compiler (see 8.3), use the table for global application-specific data, which might well need to be persistent. Since there is no way in the language to indicate that a particular property should be transient or persistent, there can be no automatic solution to this problem. Although the `Properties` interface could be extended to indicate longevity for given keys, it might be preferable to explicitly define the table as transient and require that applications make alternative arrangements for communicating data.

3.3.3 Summary

Overall, these issues are of only minor concern, and we conclude that a very high degree of compatibility has been achieved in PJava.

4 PJava₀

PJava₀ is an initial prototype that is based on version 1.0.2 of the JDK from Sun Microsystems. PJava₀ currently runs only on the SPARCTM architecture under SolarisTM. In order to produce a prototype in a reasonably short time, as few changes as possible were made to the virtual machine implementation. PJava₀ will be made available by Sun to interested research groups in the near future.

PJava₀ provides a modified virtual machine. This is contentious because it violates the *run anywhere* guarantee. Applications that exploit PJava's persistence features will not run on standard virtual machines. Others [7] are attempting to provide persistence as an extension, ideally written entirely in Java. However, it is hard to see how to achieve the degree of orthogonality that PJava aspires to without modifying the virtual machine. Ideally we would like to see orthogonal persistence as part of the Java platform.

The implementation of PJava₀ is based around a cache of loaded objects and a separate buffer pool that manages pages transferred from the disk. The buffer pool itself is implemented on top of Recoverable Virtual Memory (RVM) [8]. The garbage collected heap in the Java Virtual Machine (JVM) has been left mostly unchanged. New persistent objects are copied from the heap to the object cache as the first part of a stabilization. Objects are copied between the object cache and the buffer pool as required. Object references are converted (swizzled) between machine addresses and persistent identifiers as part of this process. To reduce this overhead, not all references are immediately unswizzled on object load, but are translated on demand. Unswizzling is handled in a thread-safe manner.

A special area of the store called the *bootstrap region* is set aside for the `PJavaStore` class, those classes that it depends on and all the instances and associated classes reachable from the root table at the time the store is created. The bootstrap region is loaded in one step during startup, the idea being to speed up the loading of objects that will very likely be accessed.

The PJava interpreter runs in two modes: store mode and non-store mode. In non-store mode the interpreter behaves almost exactly like a standard Java interpreter, with the extra capability to create a store during execution and register objects as persistent roots. In store mode an extra argument “-store pathname” is passed to the interpreter. The pathname argument indicates the name of a persistent store against which to run. In PJava₀ a store is implemented as a file. The interpreter opens the store as part of its initialization process and makes the persistent roots available via an instance of the `PJavaStore` class.

The bi-modality implied by the “store” argument has proven to be an inconvenience. In a future release the interpreter will automatically switch to non-store mode if the file given as argument does not exist. Access will also be provided in the `PJavaStore` class to the pathname, which can then be used as the name of the store to create.

5 Performance and Reliability of PJava₀

The overall experience with PJava₀ has been very positive, both from a performance and a reliability perspective. This is particularly so given the relatively short time frame in which it was implemented and the fact that it is based on a relatively complicated and undocumented virtual machine implementation.

As a testament to this, the Forest benchmark, described in section 8.2, ran correctly on its first execution.¹ Fur-

thermore, the performance was quite a bit better than the version which runs over ObjectStore/C++ [9], a commercially supported object-oriented database.

6 Limitations of PJava₀

In this section we describe some of the limitations of the PJava₀ implementation that cause it to fall short of the goal of fully orthogonal persistence. The limitations either result from the constraints of the original JDK virtual machine design or from time and manpower considerations on PJava₀.

6.1 Class Loading

The Java Language Specification [6] defines the rules governing class loading in considerable detail. It defines specific phases of the loading process, taking pains to define exactly what, in source code, causes each phase to be completed. The specification explicitly permits different class files formats but is insistent that bindings between class files be symbolic. The specification permits some flexibility in the time at which references between classes are resolved. PJava₀ loads and resolves all required classes, transitively, on a store stabilization, so that a store is always self-contained.

Java permits multiple instances of a given named class to co-exist in the same virtual machine, provided that each is loaded by a different *classloader*. A fresh virtual machine contains only the system classloader, which cannot be replaced. The classloader used to load a class plays a role in the dynamic type-checking (*instanceof*) operator. Two instances of class C loaded by different classloaders are not be considered to be the same type. Note that they may have different definitions anyway, but this is irrelevant since Java uses name equivalence for type-checking purposes.

PJava₀ maintains a single class dictionary, in the *PJavaStore* class, which acts as the persistent table of classes loaded by the system class loader. The associated classloader for a class is currently *not* recorded in this table. Therefore, confusion and possible internal errors may occur if an application attempts to stabilize a store containing multiple class instances. This limitation will be removed in a future release.

6.2 Threads

Orthogonal persistence requires that instances of class Thread can be made persistent. This is not possible in PJava₀, in part owing to the complexity of the thread implementation in the JDK. Further, applications are

limited to using *cooperating* threads which are under the control of a manager that has sufficient knowledge to determine when it is safe to stabilize the store. This is due to the fact that, although *PJavaStore.stabilizeAll* is a synchronized method, thus serializing stabilize calls from multiple threads, there is no easy way to stop only the application threads. It is possible to enter single-threaded mode, but then the stabilization cannot succeed because more than one thread is needed for output. So, currently, threads are not stopped and therefore, while stabilization is occurring, other, possibly damaging, thread activity may be occurring. This could be repaired by appropriate locking in the virtual machine, but at some cost. It seems preferable to defer this issue to the design and implementation of the implicit locking required by the PJava extensible transactions model.

6.3 Extensible Transactions

The PJava extensible transaction model is not implemented in PJava₀. Applications are therefore limited to periodic checkpoints, with no ability to rollback. Nonetheless, a wide variety of useful applications can be written with this limitation.

6.4 Object Cache Replacement

In the current version of PJava₀, it is not possible to replace objects in the object cache. The cache is used for all objects brought into memory from the store, and for all newly allocated objects that are to be made persistent on a stabilize. The cache is also of a fixed size determined on interpreter startup. Therefore, there is a fixed limit to the amount of data that can be processed in one execution, which is clearly a major limitation for long-running applications that are associated with large stores. Work is underway to remedy this problem in the PJava₀ prototype.

In the applications that have been written to date, this problem typically affects the store loading phase, where a large amount of data is read from the file system and placed in the store. Since the cache size can be set on interpreter startup it is usually possible to workaround the fixed limit. Applications that process the resulting stored data are much less likely to access the entire database and can therefore usually execute with the default cache size. PJava's object cache architecture is a benefit here, since only those objects that are actually needed are transferred from the page buffer pool to the cache.

6.5 Native Code

In the PJava design [1] it is argued that native code

1. Having first been debugged in a non-persistent version.

should not be permitted in PJava applications because of the risks to data consistency that arise from the unchecked access to memory that is possible in native code. However, pragmatically, application writers must decide the appropriate trade-off between safety, functionality and performance. As more functionality migrates to the Java core, for example, Remote Method Invocation (RMI) [10], and just-in-time (JIT)¹ compilers become widespread, we expect the need for native code to diminish. In the meantime, native code implementors must follow some additional conventions to ensure that their code is compatible with the implementation of the PJava virtual machine. This is mostly concerned with ensuring object residency before accessing fields and methods. The situation is more complex when the native code contains state that must be made persistent in order to save a persistent version of the associated Java class. PJava₀ provides no support for native state which means that such classes cannot be made persistent. One workaround is to ensure that the state is instead declared as fields of the Java classes, but this idiom causes problems when the state is a C pointer, since Java has no way to declare such a type. It also causes portability problems between machines with 32 and 64 bit addressing. Hiding Java object references by declaring them as integers is more dangerous since it subverts the normal PJava handling of non-resident objects.

6.6 External State

Objects that contain references to external state pose a general problem for orthogonally persistent systems. Typically, such external state is associated with native code or core features of the language platform, for example operating system file descriptors (e.g. declared as integers). Such data is not marked `transient` in JDK 1.0.2 and therefore interacts badly with the PJava design choices of persistence by reachability and of `static` fields being persistent by default, since this data is usually meaningless when the object is reloaded in a subsequent execution. PJava₀ has not modified the core Java classes to solve this problem, although it does honor the `transient` modifier. This problem also besets the Java Object Serialization system [11], that will become part of the Java core in JDK 1.1 and, fortunately, in that release, such data will be marked as `transient`.

Of course, this is only half of the solution. There needs to be a way for the class to recreate the transient state when the object is reloaded. In the original PJava design this was handled via class-specific callbacks that were

registered with the `PJavaStore` class. Subsequently this was altered to be similar to the mechanisms provided for serialization, by invoking specially named methods that are provided as part of the class definition. However, these mechanisms are not yet implemented in PJava₀.

A particularly significant piece of external state is the code that actually implements the native methods. Some of this code is bundled in the interpreter. The remainder exists in dynamically loaded libraries that are loaded by calls to `System.loadLibrary` and the typical idiom is to make the call in a static initializer of the class. In PJava static initializers are only run once, when the class is first loaded, that is, when the `Class` class instance is created. So, on subsequent executions, the external library is not loaded, leading to an exception. The correct idiom, for a PJava program, would be something like the following:

```
static transient boolean loaded =
    loadLib();

static boolean loadLib() {
    System.loadLibrary("mylib");
    return true;
}
```

This idiom assumes that transient static variables are automatically reinitialized when a class is first faulted in from the persistent store. Alternatively `loadLib` could be called explicitly in a callback.

A related problem is that the library search path that is used by `System.loadLibrary` is made persistent by the default rules for static variables. This has the unfortunate property that stores cannot be moved into an environment where the library search path is different. The temporary workaround in PJava₀ is to force the search path to be reinitialized, which is straightforward, since the relevant code is in the core virtual machine. A more robust solution, that would be more consistent with the PJava model of persistent class consistency, would be to load the native code into the store and make it a persistent object. Subsequent attempts to load the library would look first in the store, as is done for persistent Java classes.

6.7 The Abstract Window Toolkit (AWT)

AWT is an important part of the Java core but the implementation, although structured for portability, is heavily platform dependent and contains a considerable amount of native code. It suffers all of the problems referred to in the previous two sections. Since AWT is a key part of the Java environment it is obviously rather common to

1. N.B. JIT compilers must cooperate with the persistence mechanisms in PJava.

attempt to make an AWT class persistent by reachability from some other class. In PJava₀ this is explicitly checked for and the stabilization aborted, since there is no prospect of the class working correctly on a subsequent execution. Unfortunately this restriction leads to some unnatural programming idioms. It is very natural and convenient to encapsulate application-level persistent data in a subclass of an AWT class. In PJava₀, the programmer must place the application data in a separate class, make this reachable from some persistent root and then associate it with the AWT class by a level of indirection.

A mechanism is needed to separate the persistent and portable aspects of an AWT class from the transient and platform dependent pieces. It would then be possible to mark the latter as `transient` and make persistent enough information to enable the class to be reinitialized on a subsequent reload, using the callback methods of PJava. This mechanism might be somewhat similar to those in place for mobile agents in Visual Obliq [12].

6.8 Persistence Independence

It should be clear from the preceding sections that the PJava design of explicit root registration is a very pragmatic choice. It completely avoids the problems that would arise if all loaded classes were the implicit roots of persistence, since many of these contain transient data not marked as such, or other state that is problematic. In the PJava design the application has complete control over which classes are made persistent.

7 Common Pitfalls

Perhaps the most significant consequence of the principles of orthogonality is the fact that code and data are bound together in a persistent store. While this provides a high degree of consistency, contributing to application reliability, it runs counter to the normal experience of most programmers. It also requires additional tools to support the evolution of software. We consider these two issues separately below.

7.1 Binding Code and Data

The lack of orthogonal persistence in everyday programming languages has meant that a significant part of learning a new programming language concerns the facilities for input/output. While, it would be incorrect to say that orthogonal persistence completely removes the need for such facilities, one of its goals is to substantially remove the need to save and restore data structures in ad hoc formats between separate executions of an application. Today, however, this is the norm and is so

ingrained that programmers have some difficulty in believing that there might be alternative.

Early programming languages were consistent with this separation of code and data, since they provided separate facilities for data definition (records, arrays etc.) and behavior (procedures), with a more or less loose connection between the data and the code that operated on it. However, since the advent of abstract data types and, more recently object-oriented programming, the focus has shifted towards the integration of code and data (state and behavior). Nevertheless, persistence mechanisms generally continue to maintain the separation. For example, `ObjectStore` does not store C++ [13] code in the database. To underline how pervasive the separation mindset can be, we must admit to spending one frustrating PJava₀ debugging session wondering why the application behavior was not changing despite having changed and recompiled the code!¹

One characteristic of the separation of code and data is the ability to open multiple databases from within one application, and the inability of PJava to do this has been commented on. However, it should be clear that if these databases contain code bound to data there is a conflict with the language semantics. There may be multiple, possibly different, versions of the same-named class, a situation that cannot exist according to the language definition.²

A much better analogy for the Java programmer is the notion of a store as a, more or less, consistent persistent virtual machine. Multiple stores are then handled as distinct virtual machines which are able to communicate, for example, by RMI. This model is more faithful to the object-oriented paradigm by focussing on active objects with behavior rather than passive data.

7.2 Schema (Type) Evolution

Since the code is bound to the data in the store, it becomes more difficult to modify it, for example to fix a bug. In addition to the need to recompile, the new code has to be installed in the store and all affected objects rebound.³ This seems onerous and makes separation of code and data seem an attractive simplification. However, the separation has a cost. Typically a consistency check (schema validation) has to be carried out every time a store is accessed. The same is true for the input

1. But only in the file system, not the store.

2. Except in different classloaders. Certainly not for the basic classes such as `Object` or `Thread`.

3. A more complex solution is to support schema *versioning* in which objects and types can coexist in multiple versions in the same store.

phase of Java Object Serialization. In contrast, except during schema evolution, PJava needs to make no checks, thus maximizing performance when the application is actually in use.

The lack of schema evolution tools is particularly felt by application developers since their main task is to change code, and this lack has been noted by the users of PJava₀. Once an application is deployed, the need for evolution occurs less frequently, corresponding to well defined release points. Further, when it does eventually occur it is common for the changes to be significant enough to cause problems even for applications using ad hoc persistence. Ultimately we expect the reflection capabilities available in PJava to actually ease such transitions, and even increase the rate at which applications can evolve without compromising reliability.

Since PJava₀ provides no schema evolution mechanisms, this means that stores must be completely rebuilt in the face of change. Clearly this is unacceptable in the long term. As a short term solution we investigated the provision of a simple schema evolution tool at SunLabs this summer. We made some progress towards supporting simple changes, such as only modifying the code of methods. However, the current solution suffers problems of scale that cannot be fixed without redesigning part of the PJava₀ implementation. It is clear that it is essential to keep schema evolution (or schema versioning) in mind when designing a persistent object system. The constraints under which PJava₀ was built prevented this foresight.

One problem that must be addressed by any schema evolution mechanism for PJava is what type compatibility rules to enforce. PJava takes a strong, compile-time centric, view of type compatibility which is reflected in the approach to class loading described in section 6.1. The language specification, on the other hand, specifies a weaker set of consistency rules to be applied at class load time. In particular, sets of classes that would not compile together can pass these weaker rules. Ironically, this weakening is justified to support a limited form of class evolution that does not require recompilation. It is to be hoped that evolution can be put on a firmer footing in the context of PJava.

In summary, it is clear that schema evolution is a very high priority feature to provide in future releases of PJava.

8 Example Applications

Since PJava has only been available for a short period and to a limited group of users, we do not have a extensive set of substantive applications to report on. How-

ever, the following set of applications cover a fairly wide spectrum and provide useful insights. For each application, we provide a brief description and then discuss it from four perspectives, suitability for PJava, ease of programming, performance and extra functionality.

8.1 Oscar

Oscar is an application from the domain of Geographical Information Systems (GIS), authored at Glasgow. Originally written to test the capabilities of Java in this domain, it was subsequently modified to run under PJava.

Oscar consists of three main phases:

1. Open a file of NTF data (National Transfer Format), as available from Ordnance Survey (GB). This data is a representation of carriageways (roads) in the UK.
2. Read the file a line at a time (where a 'line' of data can actually span several physical lines in the file). Each line represents one NTF structure. Each line is parsed and a Java object corresponding to the NTF structure is built. This continues until an end-of-data indicator is encountered.
3. AWT is used to display the data in a meaningful manner. Rudimentary GIS features are available, such as clicking a road to get information about it. It is possible to alter the display colors for different kinds of roads.

In normal usage Oscar expects to load 4 *tiles* NW, NE, SW and SE, where a tile is one file of NTF data, representing a 5km square area. These tiles are displayed via an offscreen image and can be shown in a number of scales (in pixels) defaulting to 800x800. The viewing area can display one complete tile and the user scrolls around to view the rest of the quadrant.

The four tiles comprise quite a lot of data, with each tile containing between 8,000 and 20,000 objects. For the tiles used in the experiment this translates to reading data for approximately 60,000 objects and building the corresponding Java objects.

8.1.1 Suitability for PJava

Oscar is representative of the classic set of applications for which persistent object systems were invented.

8.1.2 Ease of Programming

Converting Oscar to PJava was easily accomplished, requiring the addition or amendment of 23 lines of code in only 4 compilation units and one new class to load the tiles into the store. The limitation of no persistent AWT classes in PJava₀ made the conversion slightly more onerous.

8.1.3 Performance

Loading the tiles from the NTF format files was taking between 3 and 8 minutes per tile, depending on the amount of data in the tile and the load on the system. Loading all four would normally take about 20 minutes. While the interpreted implementation of Java in the Sun JDK undoubtedly inflates this time over a compiled implementation, the need to read the entire file clearly provides a limit on scalability.

The performance benefits of converting to PJava were substantial. The time to display a tile now measures in seconds - on average about 20 seconds to process the tile from the store, draw its data into an image buffer and display that buffer. Subsequent redispays are much faster since the objects have been loaded into the cache.

Since all the NTF data from one file has to be read into memory in one go, the Java version could run out of memory. This is an example of the *big-inhale* effect. In PJava, once the data has been converted to persistent objects, only the subset that is needed for a particular display need be loaded from the persistent store. This permits large sets of tiles to be loaded and saved incrementally in a single store. We must stress that no work is required on the part of the programmer to effect the storing of the data, beyond registering the root objects.

8.1.4 Extra Functionality

Although it would have been possible to save the map from road kinds to display colors in the Java version, using some ad hoc persistence mechanism, it would have involved extra programming and a mechanism to connect the color data and the NTF data. Since this structure had already been constructed in the program domain, making it persistent required no extra work under PJava.

8.2 Forest

Forest is an application from the domain of Computer-Aided Software Engineering (CASE). The first prototype was written in C++ using the ObjectStore object-oriented database.

Forest integrates the three essential activities of software development: authoring, versioning and configuration management and system building. It adopts the Vesta [14] approach to configuration management, which is characterized by a repository of immutable, versioned, *source* objects, that are combined with modular system build descriptions to generate *derived* software artifacts.

8.2.1 Suitability for PJava

Our experiences in implementing the Forest prototype

with ObjectStore/C++ were positive [15]. However, dissatisfaction with C++, the lack of orthogonality in ObjectStore/C++, and general enthusiasm for Java, lead us to the SunLabs collaboration with Glasgow to develop Persistent Java. We believe that CASE tools are an ideal application for persistent object systems and that, in such a framework, the Vesta approach is an optimal choice. In the long term, we believe that the extensible transaction model of PJava will greatly assist in the development of collaborative software development tools.

The complete Forest environment has not yet been translated into PJava. However, we have converted a benchmark that we developed as part of the evaluation of the prototype based on ObjectStore. The benchmark simulates a number of users exercising the versioning and authoring system by checking out components, editing them and checking them back in. Following Vesta, the logical unit of checkout is an entire tree of objects. Each step in the process is logically a separate transaction. Each run of the benchmark performs a set number of checkouts, passed in as a parameter. The particular set of objects that is accessed is chosen pseudo-randomly. Since ObjectStore is a commercial product a comparison between it and PJava₀ is a useful datapoint.

8.2.2 Ease of Programming

The Forest environment contains a fairly large set of object types; indeed an open-ended set, since developers can define new types using the environment itself. Therefore, the orthogonality of the persistence mechanism has a direct impact on the ease with which the system can be programmed. The ObjectStore/C++ version fell quite a bit short in this regard and in the end, in order to achieve adequate performance, our code had become quite ObjectStore specific. In particular, with ObjectStore, persistent objects can only be accessed inside transaction boundaries, and new persistent objects must be allocated with syntactically different forms of the C++ new operator. Paying attention to clustering, which must be managed at allocation time, turns out to be very important owing to false lock conflicts that can arise because of ObjectStore's choice of page-level locking and data transfer.

In contrast, the PJava version, which, like Oscar (8.1), was originally developed and tested as a non-persistent application, required only two changes. The first, isolated to one module, was the registration of a single object as the persistent root. The second change was the insertion of the calls to the `stabilizeAll` method to checkpoint the store at the transaction boundaries.

The addition of the calls to the `stabilizeAll`

method to achieve the checkpoints, if done in the obvious and simple way, does cause the code to become persistence-specific. Had the PJava transaction model been implemented, we would have used a sequence of flat transactions as we did in the ObjectStore version. Since the PJava transaction model supports the composition of user transaction code¹, with hindsight we could have made the body of the benchmark persistence-independent by defining a simple subclass of `TransactionShell` that provided only the durability (D) aspect of transactions (by virtue of global stabilization), and composing this with the classes implementing the benchmark proper.

8.2.3 Performance

An important aspect of the ObjectStore version of the benchmark was measuring how the system behaved as concurrent users were added. Unfortunately, PJava₀ does not yet support concurrent transactions, therefore performance comparisons are limited to the single user case. Even then, architectural differences between the two systems make like comparisons difficult. However, they do provide some insights into the costs of architectural decisions. ObjectStore employs a client-server architecture, with the server acting as the concurrency control point between the clients, which are regular operating system processes. In general, objects (pages) are transferred from the server to the client machine for processing locally, and then back to the server on a transaction commit. Typically, the clients reside on separate machines in a network. However, it is possible to execute the clients on the same machine as the server, and configure the server to use shared memory rather than sockets for client-server communication, and we used this mode for the comparison. One final difference is that code is not stored in an ObjectStore database; instead a schema (type) check is carried out every time a client starts a transaction. Since the type check does not extend to code behavior, this is a weak consistency check, but substantially reduces the need for schema evolution.

PJava₀, on the other hand, is not client-server in the ObjectStore sense and, as noted earlier, has an architecture that is essentially a (consistent) persistent virtual machine. The clients share the same address space (and protection domain) with the PJava₀ system. This is acceptable because Java is a safe language. Contrast this with Thor [16], which wishes to support clients written in unsafe languages and is devising ways of dealing with

the performance problems that result from crossing protection boundaries [17]. Although it would be possible for multiple PJava interpreters to execute in read-only mode from the same persistent store, concurrent updates can only be achieved using a single interpreter and the basic Java concurrency mechanism, namely threads. The limitations on PJava₀ prevent us from benchmarking the multi-user simulation.

We used the same data for the benchmark that we reported on in our previous paper [15]. The initial store is about 13Mb in size and consists of about 2500 text files in 216 (versioned) directories. The benchmark accesses all the versioned directories and creates about 0.5Mb of new leaf objects. It must be stressed that the PJava version is not a direct port of the C++ version, but the general structure of the code is the same. Also, the C++ code is compiled into machine code, whereas the PJava code is interpreted by bytecodes.

The total elapsed time for the single-user version of the benchmark running on ObjectStore/C++ version 4.0 was 207 seconds, and for PJava was 16 seconds, both times averaged over four runs and rounded to the nearest second. The tests were run on a SPARCcenterTM 2000, with 196Mb of memory, and the stores on a locally attached disk. The ObjectStore figure is consistent with the value that we reported previously, although we are now using the production version rather than the beta release.

We were not expecting to find an order of magnitude difference in favor of PJava, and began to search for an explanation. One surprising fact is that there appears to be only about a 10% improvement running the ObjectStore/C++ version on the same machine as opposed to a client-server combination. This suggests that the shared memory communication is providing little benefit.

As an experiment, we altered the benchmark to execute as a single transaction. In this case, the ObjectStore time came down to 53 seconds and the PJava time reduced to 12 seconds. Evidently transactions are considerably more expensive in ObjectStore than are stabilizations in PJava. Given that ObjectStore provides concurrency control and rollback, this is not surprising, although the cost seems rather high.

In the ObjectStore/C++ variant, the initial scan of the store to locate all the versioned directories took 31 seconds, as compared to 3 seconds in PJava, a substantial difference that demands an explanation. ObjectStore performs schema validation on every transaction.² We tested the cost of this by repeating the initial scan trans-

1. The lack of first-class functions (methods) in Java limits the composition to classes that implement the `Runnable` interface.

2. If the database has not been changed by another client since the last transaction, this test is very fast.

action and found that it took only 14 seconds. Therefore 17 seconds can be attributed to schema validation, a cost that PJava need not pay because of the consistent binding between code and data.

In conclusion the performance difference between ObjectStore/C++ and PJava₀ clearly merits more study. However, the results are very encouraging given the prototype nature of PJava₀, and demonstrate that the strong consistency guarantees of PJava have a clear performance benefit.

8.2.4 Extra Functionality

Forest was designed from the outset with a persistent object system as the implementation infrastructure, and many of its capabilities simply could not be realized (efficiently) without it. Indeed, a recent port of a limited set of its functionality using file-system storage served to remind us of this fact.

8.3 Java Compiler

The Java compiler in the Sun JDK was one of the first medium-sized applications to be written entirely in Java. It consists of over 300 classes. The compiler is ordinarily executed once per compilation unit, although it can compile multiple compilation units in one run.

8.3.1 Suitability for PJava

Superficially, a compiler does not appear to be an obvious application for PJava. If the source code itself is kept in the persistent store, as is the case in Forest (8.2), then the compiler necessarily becomes a suitable application. However, in its normal mode of reading and writing a file system, its requirements for orthogonal persistence appear limited. But delving a little deeper reveals several benefits from the PJava environment.

In general, every time the Java compiler runs, all of its classes are loaded from the file system, although, if there are errors, then the classes that handle code generation are not loaded. Small variations in the set of loaded classes may also occur depending on the nature of the source code being compiled. However, on average most of the classes are loaded. In principle, every time a class is loaded, it is verified. In practice, classes loaded from directories in an application's classpath are not verified, even though this is a trade-off between performance and security. In addition, the standard optimizations are performed on the bytecodes every time they are loaded. Evidently it should be possible to exploit the persistence of classes in PJava to reduce these costs, since the verification and optimization occurs once when the class is first loaded.

When a compilation unit contains errors, the compiler loads the error messages dynamically from a *properties* file.¹ This file is expected to be found at a special pathname relative to another, standard system property called `java.home`. While this approach provides some flexibility for tailoring the content of the error messages, for example, for internationalization purposes, it introduces the potential for inconsistency and adds a surprising amount to the compilation time (see 8.3.3). By binding the properties table with the compiler classes in the persistent store, we remove the potential for inconsistency and avoid the per-compilation overhead of loading the table.

8.3.2 Ease of Programming

Modifying the compiler to load the properties table and save it as a persistent object was simple. The code that already existed to read the table was moved from the main part of the compiler into a new class, `Install`, that is invoked to perform initialization. This permits the error message table to be reloaded at a later date, if required. It should be noted that the compiler could be modified to encapsulate the error messages as code, to completely remove the dependency on an external file. However, modification of the messages would then require a custom interface to display and edit the table.

Loading the classes into the persistent store is also carried out at initialization time. To achieve this it suffices to create a dummy persistent instance of the `Main` class and register it as a persistent root. The PJava rules for persistence by reachability of classes then force all necessary classes to be made persistent.

8.3.3 Performance

The time to load the error messages table from a file was 4 seconds, which effectively reduces to zero in the PJava version.

If the `Install` class operates on an existing, minimal, store, then the classes specific to the compiler will be faulted in on demand during a compilation. If the `Install` class creates a new store, then all the classes will be stored in the bootstrap region, which will be loaded in one step on startup. The rationale for the existence of the bootstrap region is to minimize the faulting overhead for classes that will be loaded early in an application's lifetime. Since the compiler contains a fairly large number of classes, it serves as a testbed for the effect of this optimization in the PJava₀ implementa-

1. The standard class `Properties` is a map from `String` to `String` values and has built-in persistence support through a defined file format.

tion.

We measured the time the compiler took to compile the `java.lang` classes, one compilation unit at a time, in four configurations:

1. using the standard JDK 1.0.2. interpreter (java).¹
2. using the pjava interpreter in non-store mode (pjava).
3. using the pjava interpreter in store mode without the compiler classes in the bootstrap region (pjava-s).
4. using the pjava interpreter in store mode with all classes in the bootstrap region (pjava-sb).

The compilation time in seconds, running on a SparcStation™ 10, averaged over four runs are as follows:

Table 1: Compilation time for java.lang

java	pjava	pjava-s	pjava-sb
380	392	332	346

We see that the PJava interpreter adds a small overhead to the compilation time. As expected, in store-mode, the time is reduced because it is faster to fault in the resolved classes that reload them from the class files. However, storing all the classes in the bootstrap region performs worse than this, although still better than non-store mode. We believe that a possible explanation for this is that there are more classes in the bootstrap region than are strictly needed by the compiler. Owing to a bug in the current version of PJava₀, we are unable to store just the transitively reachable classes. Instead, if any class is used from a package, we store all the classes in that package. However, it seems unlikely that fixing this bug will cause pjava-sb to perform significantly better than pjava-s.

The time to compile with class verification enabled was 620 seconds for the standard JDK compiler, thus verification accounts for an additional 240 seconds. This cost is incurred once in the PJava version during the initialization phase and so compiling with verification enabled should be identical to the figures in Table 1. Unfortunately, owing to another bug in PJava₀ related to class verification, we have been unable to prove this.

8.3.4 Extra Functionality

We did not attempt to add extra functionality to the compiler, although a number of possibilities suggest them-

1. We used a non-optimized version of the interpreter for compatibility with the non-optimized pjava interpreter.

selves. For example, the ability to encapsulate arbitrary configuration data, for example compiler options, could be arranged through a customization interface.

8.4 Jeeves

Jeeves [5] is an HTTP server under development at Sun. Jeeves is entirely written in Java and provides for extensibility through *servlets*, which may be loaded from anywhere on the Internet. Servlets, like applets, execute in a controlled environment (sandbox). Unlike applets, servlets have no user interface component.

8.4.1 Suitability for PJava

Jeeves is a highly suitable application for PJava. Jeeves maintains a considerable amount of state that controls its behavior in the external file system. Although, currently, the webmaster is required to edit some configuration files manually, the intent is to replace this with an applet interface that will allow the server to be configured from any web browser capable of running Java. If running under PJava, the server configuration data could be completely encapsulated and made durable in the face of crashes.

More interesting is the potential for storing user data in the persistent store associated with the server rather than the external file system. It is already common for OODB vendors to offer web front-ends to their databases, it being relatively straightforward to transform simple objects into an HTML representation.

The PJava design includes a proposal for a new URL protocol that provides type-checked access to objects in a persistent store [18]. This protocol could easily be recast into an HTTP-based servlet URL. However, it cannot easily be implemented without additional reflection facilities such as those proposed for JDK 1.1 [19].

Alternatively an applet could be transferred to the client browser and set up a custom communication mechanism using RMI, or some similar mechanism, to access the objects in the associated persistent store.

8.4.2 Ease of Programming

At the time of writing we have not modified Jeeves to store its configuration data in the persistent store. However, we expect the modifications to be very similar to those carried out for the Java compiler properties table, since Jeeves also uses property tables.

We have experimented with a simple persistent counter servlet, which requires only that the counter be registered as a root and that the store be stabilized on each request from a client browser, resulting in two extra lines of code.

Jeeves exploits multi-threading to service many requests concurrently. Each servlet executes in its own thread, and servlets are, in general, unaware of each other. It is therefore quite likely that several threads might attempt to stabilize the store at the same time. In PJava₀, this must be prevented by serializing the stabilize calls. Since the interface between Jeeves and a servlet is already handled through a method in a special class, the cleanest way to achieve this is to wrap this method in a `TransactionShell` that enforces serial stabilization.

8.4.3 Performance

Running an internal benchmark, we have observed a slowdown of 6% when running Jeeves under the `pjava` interpreter in non-store mode. It would be informative to compare the time taken to serve up HTML from the host file system against the time to serve it from the persistent store, and we hope to report on this in a later paper.

8.4.4 Extra Functionality

One piece of extra functionality that PJava₀ could provide, which would otherwise require special programming, is transparent caching of objects through the mechanisms of the object cache. Caching objects from the external file system would still require periodic checks for external modifications. One interesting possibility would be to provide a file-system emulation in the persistent store, thus eliminating this requirement.

9 Conclusions

The early experiences with PJava have been positive; reliability is good and performance rather better than might be expected from a first prototype. Many useful applications can be written with the simple global stabilization mechanisms available in PJava₀.

PJava is compatible with Java programs that do not use the persistence mechanisms. In PJava₀, several issues remain to be resolved regarding state that exists outside the control of PJava. Wider use of the `transient` modifier in the Java core packages would help in this task.

The transition from PJava₀ to a fully-fledged environment, supporting concurrent transactions and the extensible transaction model, and with the ability to evolve the code and data of applications that reside in PJava stores, remains a significant design and implementation challenge. We hope to be able to report our experiences with such a system in the future.

10 Acknowledgments

Information on Oscar was provided by Stewart Macneill of Glasgow. Cathy Waite, also from Glasgow, provided helpful input on her experiences with PJava. Finally, I would like to thank the Glasgow PJava team of Malcolm Atkinson, Susan Spence, Laurent Daynès and Tony Printezis for their help in clarifying my understanding of the PJava implementation, and their prompt response to bug reports.

11 Trademarks

Sun, Sun Microsystems, Java and Solaris are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. SPARC, SPARCstation and SPARCcenter are trademarks of SPARC International, Inc., licensed exclusively to Sun Microsystems Inc.

12 References

- [1] Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, M.P. Atkinson, M.J. Jordan, L. Daynès, S. Spence, Proceedings of the 7th International Conference on Persistent Object Systems, Cape May, New Jersey, May 1996.
- [2] PJava Design 1.2, Working Document, available via <http://www.dcs.gla.ac.uk/pjava>.
- [3] Orthogonally Persistent Object Systems, M.P. Atkinson and R. Morrison, VLDB Journal, 4(3), pp319-401, 1995.
- [4] The Java Development Kit version 1.0.2, <http://java.sun.com/JDK>.
- [5] Jeeves, <http://www.javasoft.com/jeeves/>
- [6] The Java Language Specification, James Gosling, Bill Joy, Guy Steele, Addison-Wesley, 1996, ISBN 0-201-63451-1.
- [7] Object PSE Pro for Java, Object Design Inc. <http://www.odi.com/products/pse/psepj.html>.
- [8] Recoverable Virtual Memory, H.H. Mashburn and Satyanarayanan, RVM Release 1.3, CMU, January 1994.
- [9] The ObjectStore Database System, Charles Lamb, Jack Orenstein and Dan Weinreb, *Communications of the ACM* 4,10 (October 1991) 50-63.
- [10] Java Remote Method Invocation, Revision 0.9, Sun Microsystems Inc., May 1996.

- [11] Java Object Serialization Specification, Revision 0.9, Sun Microsystems Inc., May 1996.
- [12] Migratory Applications in Visual Obliq, Krishna Bharat and Luca Cardelli, Proceedings of the ACM Symposium on User Interfaces, Software and Technology, Pittsburgh, PA, Nov. 1995.
- [13] *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, Reading, Massachusetts, 1990.
- [14] The Vesta Approach to Configuration Management, Roy Levin and Paul McJones, DEC Systems Research Center TR 105, June 1993.
- [15] Software Configuration Management in an Object-Oriented Database, Mick Jordan and Michael Van De Vanter, USENIX Conference on Object-oriented Technologies, Monterey, CA, June 1995.
- [16] B. Liskov et al. The language-independent interface of the Thor persistent object system. In *Object-Oriented Multi-Database Systems*, pp570-588, Prentice Hall, 1996.
- [17] Type-Safe Sharing can be Fast, B. Liskov, A. Adya, M. Castro, Q. Zondervan, Proceedings of the 7th International Conference on Persistent Object Systems, Cape May, New Jersey, May 1996.
- [18] Distribution Strategies for Persistent Java, Susan Spence, Proceedings of the 1st International Workshop on Persistence for Java, Drimen, Scotland, September 1996.
- [19] JDK Preview, Sun Microsystems Inc., <http://www.javasoft.com/products/JDK/1.1/designspecs/index.html>

Towards Accessible Human–Computer Interaction

Eric Bergman and Earl Johnson

Introduction by Earl Johnson

This paper identifies accessibility related factors to consider when designing the user interface of a software product. It gives user interaction designers and product developers an overview of the following topics:

- the software interaction issues faced by users with disabilities
- the types of tools used by people with disabilities to access information technology
- design considerations that improve a product's accessibility
- the market value of providing accessibility in a product

This paper was one deliverable of the Enabling Technologies project, which began in 1991 with two main goals: designing and delivering support for accessibility into the technologies that developers use to build products; and infusing accessibility design markers into the DNA of the overall product development process. In addition to this paper, the project's accomplishments included the first screen reader providing blind access to the UNIX® environment (Mercator), the delivery of keyboard enhancements into the Solaris™ Operating Environment to enable system access by people with a variety of physical disabilities (AccessX), and the design and partial implementation of an accessibility architecture in the X11 R6 Window server. This last achievement provided the philosophical design basis for key developer technologies delivered by Sun today.

In 1995, after four years of incubation during which accessibility to UNIX® was shown to be achievable, the Enabling Technologies project moved from the Labs to the Solaris OE product development organization, so that the effort could be more effective at influencing the design and development of Sun products. Some notable events that have occurred since the project graduated from the Labs include:

- the creation of the Accessibility Program Office, which is chartered with driving the development of a coherent cross-SMI accessibility strategy
- the definition and implementation of accessibility support directly into Java's JFC/Swing UI toolkit, making it easy for mainstream developers (without any expertise in accessibility) to build accessible applications almost by default
- recognition from the American Foundation for the Blind — the 2001 Access Award, citing Sun's Java Accessibility work and commitment to universal design
- collaboration with Sun's GNOME team and the open source community to design and implement an accessibility architecture that is a direct descendant of the vision born in the Labs and fully proven in the Java platform.

REFERENCE:

"Towards Accessible Human–Computer Interaction", Eric Bergman and Earl Johnson, Advances in Human–Computer Interaction, Volume 5, Jakob Nielsen, Editor, Copyright 1995.

Towards Accessible Human-Computer Interaction¹

Eric Bergman

SunSoft

Earl Johnson

Sun Microsystems Laboratories

INTRODUCTION

In spite of the growing focus on user-centered interface design promoted by human-computer interaction researchers and practitioners, there remains a large and diverse user population that is generally overlooked: users with disabilities. There are compelling legal, economic, social, and moral arguments for providing users with disabilities access to information technologies. Although we will touch on some of those arguments here, the primary purpose of this chapter is to provide a broad overview of the software human-computer interaction (HCI) issues surrounding access to computer systems.

The needs of users with disabilities are typically not considered during software design and evaluation. Although there are many plausible explanations for this omission, we are inclined to believe that much of the problem simply stems from lack of awareness. Until recently, there was little contact between HCI organizations and the sizable disability community of people with disabilities (McMillan, 1992). As a result, many software designers are not aware of the needs of users with disabilities. We hope that this chapter will serve to increase awareness of the scope and nature of these needs, and to stimulate interest in research and implementation of systems that enable access to information technologies by users with disabilities.

Designing software that takes the needs of users with disabilities into account makes software more usable for all users: people with disabilities who use assistive technologies, those who use systems "off the shelf", as well as users without any significant disabilities. Considerable literature already exists that discusses how people with disabilities can use assistive hardware and software to interact with computers. For this reason, we provide relatively brief coverage of assistive technologies here.²

In this chapter, we define and discuss accessibility, discuss accessibility design issues, provide a broad outline of the capabilities and needs of users with various disabilities, present guidelines for increasing application accessibility, and discuss future

-
1. The authors would like to thank Ellen Isaacs, Elizabeth Mynatt, Mark Novak, and Will Walker for their valuable suggestions and comments on this chapter
 2. For more detailed information on assistive technologies see Lazzaro (1993), Church and Glennen (1992), Brown (1992), and Casali and Williges (1990).

directions for improving accessibility.

THE RELEVANCE OF ACCESSIBILITY

What is Accessibility?

Providing accessibility means removing barriers that prevent people with disabilities from participating in substantial life activities, including the use of services, products, and information. We see and use a multitude of access-related technologies in everyday life, many of which we may not recognize as disability related when we encounter them. The bell that chimes when an elevator is about to arrive, for example, was designed with blind people in mind (Edwards, Edwards, and Mynatt, 1992). The curb cut ramps common on street corners in the United States were introduced for wheelchair users (Vanderheiden, 1983). Accessibility is by definition a category of usability: software that is not accessible to a particular user is not usable by that person. As with any usability measure, accessibility is necessarily defined relative to user task requirements and needs. For example, a telephone booth is accessible (e.g., usable) to a blind person, but may not be accessible to a person using a wheelchair. Graphical user interfaces are not very accessible to blind users, but relatively accessible to deaf users.

Vanderheiden (1991) makes a distinction between “direct” access and access through add-on assistive technologies. He describes direct access as “adaptations to product designs that can significantly increase their accessibility...” (p. 2). A major advantage of this approach is that large numbers of users with mild to moderate disabilities can use systems without any modification. Examples of direct access include software keyboard enhancements included with X Windows, OS/2, and the Macintosh (see Table 2).

Assistive access means that system infrastructure allows add-on assistive software to transparently provide specialized input and output capabilities¹. For example, screen readers (see Table 3) allow blind users to navigate through applications, determine the state of controls, and read text via text to speech conversion. On-screen keyboards replace physical keyboards, and head-mounted pointers replace mice. These are only a few of the assistive technologies users may add on to their systems.

We claim that in order to truly serve users with disabilities, accessibility must mean more than simply providing “direct” access through assistive technologies bundled with system software, and more than providing the capability to add such assistive technologies. It also must mean designing application user interfaces that are easier to use for users with disabilities as well as users “with out” disabilities by taking their needs into account when system and application software is designed.

1. By *infrastructure* we mean an environment’s standard set of application program interfaces (APIs).

These are low and high-level software routines used to build applications (e.g., Macintosh Toolbox, MS Windows API, Motif, and Xlib to name a few).

Broad Benefits of Accessibility

Accessibility provides benefits for a wide range of people -- not only for those with disabilities. Before curb cut ramps were placed on sidewalks, for example, it was difficult or impossible for people in wheelchairs to cross a street. In addition, curb cuts have benefited people pushing baby carriages and shopping carts as well as those on bicycles and roller blades. Vanderheiden (1983) suggested that our society is laying down electronic equivalents to sidewalks. These "electronic pathways", he argued, must include electronic "curb cuts".

Like physical curb cuts, electronic curb cuts provide benefits for the larger population as well as users with disabilities. Systems that allow use of keyboard shortcuts instead of the mouse increase efficiency of sighted users as well as providing access for blind users or users who have disabilities that affect mouse control. Users of portable computers or people in open offices may not be able to use or hear sounds, but they can use visual cues, as can hearing impaired users. Users who must keep their eyes on their task (e.g., aircraft mechanics) can benefit from systems that interact through voice rather than vision, as can users with visual disabilities.

In the near future, when the power of computing is available on noisy factory floors, in cars hurtling down expressways, and from devices stuffed in our pockets, the relevance of accessibility looms larger. As users and designers, we will soon deal with environment and task demands in which systems must deliver computing power to people who will be unable to use their hands, eyes, or ears to interact with computers. People working on accessibility have been tackling such design issues for years, typically with little or no input from HCI specialists. Clearly there needs to be a better communication between the disability access and HCI communities.

There are ample incentives for fostering a connection between design for access and design for "general audiences". Research on communication devices for the deaf led to development of the telephone, whereas development of an easy to use "talking book" for the blind led to the cassette tape (Edwards, Edwards, et al., 1993). HCI theory and practice can benefit from better understanding of the difficult issues that are already being addressed in the disability domain. Newell and Cairns (1993) cite designers who thought they had created novel interfaces, which were actually reinventions of disability access technologies such as foot-operated mice and predictive dictionaries¹. As McMillan (1992) suggested, cooperation among these different communities will require better communication among professionals in the fields of rehabilitation education and HCI.

Economic and Social Impact

A common argument is that computer accessibility is too costly, but in reality inaccessible systems may cost much more. Government statistics show that there is a growing market for accessible computer products. Approximately 43 million Americans have a dis-

1. Predictive dictionaries speed typing by predicting words as the user types them, and offering those words in a list for the user to choose. Originally intended for users with movement related disabilities, predictive dictionaries are now becoming popular for users with RSI and as a way to boost typing speed.

ability of some type (Perritt, 1991). According to Elkind (1990), about 15% of the population has a disability “severe enough to interfere with work or otherwise constitute a serious handicap”, whereas Vanderheiden (1990) points out that over 30% of the people with disabilities who want to work are not employed.

There are human costs that are harder to measure than numbers from tax tables or percentages of people employed. As a user with low-vision told us, "I don't believe in telling peers or management [that] I can't read it on the computer so I can't do my work". We have spoken to blind users who, because they are unable to read text that they have typed into word processing applications, have secretaries or coworkers type for them. We have spoken to users with mobility impairments who could not use job-critical applications because they were not accessible from the keyboard. Our experience suggests that a significant proportion of users with disabilities who are employed are not working at full productivity due to poor software accessibility.

Legal Requirements

In recent years, new laws have created incentives for the development of access to computer technology (see Casali & Williges, 1990; Lazzaro, 1993; McCormick, 1994). The most important of these laws, for the purposes of our discussion, are Section 508 of the 1986 Federal Rehabilitation Act and the 1992 Americans with Disabilities Act (ADA).

Key aspects of Section 508:

- Applies to office computer purchases made by the Federal Government
- Winning contract proposals must include solutions for employees with disabilities

Key aspects of the ADA:

- Applies to corporate entities with 15 or more employees
- Requires "reasonable accommodation" for employees with disabilities

One of the driving forces behind these laws has been the government's economic and social interest in retaining and recruiting people with disabilities. In fact, the government employs over 151,000 people with disabilities (McCarthy, 1994). The ADA was seen as a way to make the private sector a partner in hiring and retaining employees with disabilities, reducing demand for government assistance. The growing market for accessible technologies means that accessibility can be an important selling point for system and application software in both commercial and federal markets.

DESIGNING FOR DISABILITIES

Arguments typically leveled against designing for users with disabilities include the claims that costs are too high, and the benefits serve too small a market (Glinert & York, 1992). These arguments should sound familiar to HCI practitioners, who have historically faced initial resistance or even opposition to the introduction of HCI processes into product

development. Just as organizational understanding of the design process must be changed for HCI to be accepted, so too must the standard HCI conceptualization of "the user" change to recognize the needs of people with disabilities.

Ranges of User Capabilities

The traditional view of people "having a disability" or "not having a disability" is overly simplistic. All users have a range of capabilities that varies across many dimensions depending on the user and his or her life stage, task, and environment. As more of the population approaches their middle 40's, there are increasing numbers of users peering through bifocals at their screens. A nontrivial portion of the population experiences some degree of hearing loss, and may not always notice software alert sounds. As we age, more of us will develop age related disabilities -- 25% by age 55, jumping to 50% at age 65 (Vanderheiden, 1990).

In addition to normal consequences of aging, people may experience sudden temporary or permanent changes in capabilities at any time in their lives. If a computer user falls and breaks her wrist, she will spend several weeks or more with much the same keyboard capabilities as many people with spinal cord injuries or missing limbs. Every user's ability to interact with systems varies over short as well as long periods of time. A sprained wrist can restrict use of a mouse. A late night working or a lost contact lens can transform small fonts into a suddenly challenging experience. Any user who does not currently have a disability could someday have a stroke, car accident, or other event resulting in a temporary or permanent disability.

In fact, a significant number of user requirements for people with disabilities apply to almost any user, given the right circumstance or task context (Edwards, Edwards, et al, 1993; Newell and Cairns, 1993; Edwards, et al, 1993). Whether a user's hand is broken, painful due to repetitive strain injury, or permanently paralyzed, there are similar needs. Whether someone is unable to look at a screen because he is driving, or cannot see a screen because he is blind, the user requirements have much in common. Whether a user does not hear because she is talking to somebody on the phone, paying attention to her task, working in a noisy environment, or happens to be deaf is less important than the fact that users in these contexts need alternate sources of information. As McMillan (1992) observed, "From the point of view of a computer, all human users are handicapped" (p. 144).

Who is "the User?"

Who is "the user" -- that familiar catchphrase we encounter in papers, conference sessions, and design sessions? Does it include users with disabilities?

Does "the user" include a programmer we visited? She was diagnosed with muscular dystrophy in her early 20's. This condition, which results in progressive loss of muscular strength, means that she works from her motorized wheelchair, and is unable to sit upright for more than a brief time. As a result, she works in a reclined position, leaning back almost horizontally. Her vision problems limit the amount of time she can focus on the screen, and her muscular weakness prevents her from handling paper manuals.

Does "the user" include a secretary we interviewed? She has no vision in one eye and

"tunnel vision" in the other and prepares documents using a standard PC and screen magnification software. Sometimes she is unable to tell the difference between old and new email messages, because her mail application uses color to distinguish old from new. Like many users with low vision, she has problems working with columns, because it is difficult for her to see if text is aligned.

Does "the user" include a writer we know who took several months off from work when she developed tendonitis? She was not able to type or use a mouse for more than a few minutes, nor was she able to lift heavy objects, including manuals. Although her condition improved significantly over time, for several months she had a disability which affected her work significantly.

Does "the user" include a computer support technician we met? He has cerebral palsy, and is able to use only his left hand. There are keyboard assistive applications that would help speed his typing, but his work requires moving from computer to computer installing and troubleshooting Microsoft Windows software, which, at this writing, has no *built-in* access features¹. It is not practical for this user to install an assistive keyboard application for each consultation, so he instead stretches his single usable hand wide to reach and press multiple keys when he needs control keys and capital letters. Fortunately for him, and his employer, he has a large hand.

These users and many others have told us that their needs are not being met by current computer systems. Users with physical disabilities complain about applications that cannot be controlled from the keyboard. Users with low vision describe software that does not allow them to adjust the color to make text legible. Blind users complain about documentation that is not accessible because it is not available on paper in braille or on the computer as plain text (which is required for screen reading applications).

Access problems are not confined to users who have a "classic" disability. As they age, users who would claim they have no disability find that screens become more difficult to read and sounds become more difficult to hear. Users who break an arm, sprain a wrist, lose a contact lens, require bifocals, or develop repetitive stress injuries suddenly find that computer systems do not take their needs into account.

Limited View of the User

A common argument goes something to the effect that software engineers typically design for themselves, while HCI professionals follow a process based on understanding user characteristics, needs, tasks, and environments. In spite of this claim to the domain of user needs, we argue that most HCI research, literature, and practice holds forth a relatively limited view of who constitutes "the user." Most of the people we have discussed as examples of users with disabilities would not fit in this limited view.

Nielsen's (1993) Usability Engineering, for example, discussed the user in a section titled "Categories of Users and Individual Differences." Nielsen focuses on user experi-

1. Note that this is scheduled to change in "Windows 95". As demonstrated in this case, however, such add-on software alone does not meet the needs of a mobile user.

ence with computers in general, the system in particular, and the task at hand. He noted that "users also differ in other ways than experience" and went on to list such attributes as age, gender, spatial memory, and learning style. Users with disabilities are not mentioned, in spite of the explicit focus on "Categories of Users." In fact, disabilities are only mentioned in a few brief sentences in the entire book.

We use Nielsen's book only as an example of the extent to which disability issues are ignored by mainstream literature. Although there is a growing body of work on accessibility within the HCI community, it has not been generally recognized in standard texts or in work that is not explicitly focused on disability issues. Users with disabilities are simply not "on the radar screen" of mainstream HCI.

Universal Design

In recent years, as the notion of accessibility has been broadened to encompass much more than design for people with disabilities, the concept of "universal design" has gained visibility. Traditional design has focused on filling the needs of the "average" person, with the assumption that design for the average provides for the needs of most. The universal design argument is that designing for the "average" is by definition exclusionary, because the "average" user is a fictitious construct.

Attempts to design for this fictitious "average" user may not account for effects of multiple individual differences. Tognazzini (1992, p. 74) related an anecdote from Blake (1985) illustrating the pitfalls of ignoring such overlapping differences:

Several years ago, the Air Force carried out a little test to find out how many cadets could fit into what were statistically the average-size clothes. They assembled 680 cadets in a courtyard and slowly called off the average sizes--plus or minus one standard deviation--of various items, such as shoes, pants, and shirts. Any cadet that was not in the average range for a given item was asked to leave the courtyard. By the time they finished with the fifth item, there were only two cadets left; by the sixth, all but one had been eliminated.

The Universal Design philosophy emerges from a recognition of the idea central to this story -- that there is no average user. Universal designs target the broadest possible range of user capabilities. Examples of products that embody this theme include automatic doors, remote control thermostats, and velcro. Using no assistive technology, people who were previously unable to open a door, operate a thermostat, or tie their shoes are able to perform these tasks, whereas "the rest of us" find these tasks easier as well. Proponents of universal design do not assume that all users will be able to use all designs, but instead argue that by redefining our definition of the user, a much wider range of users can be accommodated without significant extra effort (Vanderheiden, 1992a).

Watch out for that Ramp

We claim that Universal Design is a worthy goal, but it is important to acknowledge that there are complex customization-related HCI issues that must be resolved before it can be achieved with computers. In discussing user interface design, Lewis and Rieman (1994) wrote, "Experience shows that many ideas that are supposed to be good for everybody aren't good for anybody." We agree that in human-computer interaction, as in much of life, what is "good for you" is not always "good for me."

An example of this principle in action was illustrated to us by a colleague who caught her foot at the base of a wheelchair ramp and tripped. The resulting fall produced injuries that included a sprained wrist and numbness in one hand. The injuries could easily have been more severe. The irony of a person acquiring a temporary or perhaps permanent disability because of an artifact designed to help people with disabilities strikes us as an appropriate alert that there may be stumbling blocks on the path to Universal Design.

One computer-related stumbling block is apparent in considering a simplified scenario of a public information kiosk. If we assume blind users must have access, then it becomes important to provide voice and sound output. There may be a tone when a control has been activated, and voice output to provide information about graphics and video displayed on screen. If a deaf user steps up to the kiosk, she will need visual notifications such as closed captions and visual alerts as alternatives to voice and sound alerts. If a user with no significant disability steps up to the kiosk, how will it interact with her? Surely she will not wish to deal with a cacophony of voice descriptions, closed captions, beeping dialogs and flashing screens?¹

Environments are needed that allow users to tailor system input and output modalities to their capabilities and preferences. Recent research has suggested that information can be represented in a form abstracted from the particulars of its presentation (Blattner, Glinert, Jorge, and Ormsby, 1992; Fels, Shein, Chignell, and Milner, 1992). The technical solution of providing multiple redundant interface input and output mechanisms is not, in itself, sufficient to resolve conflicting user needs. In the absence of any means for intelligently designing and customizing their use, highly multimodal interfaces could lead to as many usability problems as they resolve, causing some users to trip over features designed to help other users. Determining how users will interact with such systems is a challenging HCI issue.

Include People with Disabilities in the Design Process

It is in the design and evaluation of operating systems and desktop environments that designing for people with disabilities is most critical. Without the appropriate system software infrastructure, no amount of effort on the part of application developers can improve the accessibility of applications. As we discuss later, large gains in the accessibility of computer systems ultimately depend on improvements in software infrastructure.

On the other hand, there are many ways to improve the accessibility of applications within the constraints of current systems. Perhaps the most obvious way to enhance accessibility is to consider the needs of people with disabilities in all stages of the design process, including requirements gathering, task analyses, usability tests, and design guidelines. Other strategies include evaluating the usability of software in conjunction with popular assistive technologies, and testing under simulated disability conditions (e.g., unplug the mouse, turn off the sound, and use a screen reader with the monitor turned off).

1. This scenario is of course an oversimplification. Future systems are likely to adapt themselves to work intelligently with each user -- perhaps based on a stored profile. How such systems become configured and interact with users remains an interesting HCI question.

Note that none of these approaches are substitutes for testing with users. Simulation does not realistically represent the rich contexts and needs of users with disabilities. On the other hand, it is better than not testing accessibility at all.

Usability testing with even one user from each of the general disability categories we discuss in this chapter can have significant benefits for all users, not only those with disabilities. Depending on their disability, users can be especially affected by usability defects. Low vision users are sensitive to font and color conflicts, as well as problems with layout and context. Blind users are affected by poor interface flow, tab order, layout, and terminology. Users with physical disabilities affecting movement can be sensitive to tasks that require an excessive number of steps or wide range of movement. Usability testing with these users can uncover usability defects that are important in the larger population.

THE ROLE OF GUIDELINES AND APPLICATION PROGRAM INTERFACES

Design Guidelines

Although large leaps in accessibility await improvements in the design of operating systems and desktop environments, there is much that can be done to improve access within the constraints of current systems. The extent to which applications follow design guidelines, for example, can have a disproportionate affect on people with disabilities. User confusion about how to perform tasks is always a problem, but such problems become magnified for users who use alternative input and output devices or who require extra steps or time to navigate, activate controls, or read and enter text.

Keyboard mapping guidelines, for example, are especially important for those users who have movement impairments or are blind. Many of these users employ assistive software that assumes applications will use standard keyboard mnemonics and accelerators. Keyboard inconsistencies that are annoying to users without disabilities can become major roadblocks to users with disabilities.

Blind users of graphical user interfaces are especially affected by arbitrary violations of design guidelines with respect to application layout, behavior, and key mappings. These users interact with their systems through keyboard navigation, and use one-line braille displays and/or voice synthesis with screen reader software to read screen contents. Unlike a sighted user who can selectively scan and attend to screen elements in any order, blind users with screen readers move through a relatively linear presentation of screen layout. If that layout deviates significantly from guidelines, it can be especially difficult for a blind user to understand what is happening.

Because there are always exceptions, new situations, and missing guidelines, interface designers often have to violate guidelines or invent their own guidelines. In these cases, it is important that designers consider how any given violation might affect usability for users with disabilities. Unfortunately, most interface style guides were not written taking users with disabilities into account. For this reason, we have provided a set of general design

guidelines in this chapter that can be applied across a variety of environments.

Application Program Interface (API)

Just as guidelines specify standard methods for interacting with an interface, various user interface application program interfaces (APIs) specify standard methods for applications to interact with each other and the system. API's are the low and high-level software routines used to build applications. Every software environment provides standard API functions to support activities such as reading characters from the keyboard, tracking position of the pointer, and displaying information on the screen¹.

Assistive software and hardware mediates the communication between users and applications, making it particularly sensitive to cases where standard API functions are not used. This sensitivity occurs because assistive software such as screen readers and speech recognition monitor the state and behavior of applications partly by tracking their use of API functions. For this reason, applications that do not use standard API calls have the potential to create serious usability problems for people with disabilities. If an application performs a common function (e.g., reading a character from the screen) without using standard API calls, assistive software may not know that an event has occurred, and consequently the user may not be able to use the application. For the same reason, a non-standard interface component (e.g., a custom control) can cause access problems because chances are good that assistive technologies will not be able to recognize it or interact with it.

The API problem is a complicated issue, and beyond the scope of this chapter to address in detail, but it is important that project teams discuss and weigh the trade-off inherent in departures from APIs. Reasons for not using standard API functions include:

- It is not possible to implement the desired functionality using a high level API alone, because it does not support required user interface features.
- The high-level API calls do not yield desired performance.
- Developers are more familiar with lower-level APIs that existed prior to the development of higher-level APIs. In this case, developers may be more efficient or comfortable implementing existing features using the features they already know.

There are no simple solutions to the API problem. Where possible, engineers need to use APIs above the level in which assistive software connects to the system. If using a standard API constrains application functionality or performance, the reality of product or task requirements often means that it cannot be used. In cases where developers are unfamiliar with high-level APIs, training may help. When an interface feature is not supported by an API, it is rarely practical to drop that feature. In cases where an interface has features that clearly circumvent basic API functions, every attempt should be made to insure that the

1. Common APIs include the Macintosh Toolbox, the MS-Windows API, and Motif.

tasks these features support can also be accomplished using other features. In the future, access problems related to API support can be reduced environments whose infrastructure provides built-in support for access.

ABOUT DISABILITIES: BACKGROUND AND DESIGN GUIDELINES

In this section, we discuss some of the needs, capabilities, and assistive technologies used by people with disabilities, and we provide guidelines for improving application accessibility. The brief descriptions in this section do not constitute complete coverage of the wide range of disabilities, capabilities, needs, and individual differences across the population of people with disabilities--we have focused on providing a broad introduction to visual, hearing, and physical disabilities. Users with cognitive, language, and other disabilities may have needs in addition to those discussed in this chapter¹.

Use of assistive technologies varies across users and tasks. Our discussion of assistive technologies is not comprehensive, but it does cover many commonly used software and hardware solutions. In reading this section it is important to remember that as with all users, the needs of users with disabilities vary significantly from person to person. Many users with disabilities do not use assistive technologies, but can benefit from small design changes. Other users have significant investments in assistive technologies, but they too can benefit from software that better responds to their interaction needs.

About Physical Disabilities

Physical disabilities can be the result of congenital conditions, accidents, or excessive muscular strain. By the term "physical disability" we are referring to disabilities that affect the ability to move, manipulate objects, and interact with the physical world. Examples include spinal cord injuries, degenerative nerve diseases, stroke, and missing limbs. Repetitive stress injuries can result in physical disabilities, but because these injuries have a common root cause, we address that topic below under its own heading.

Many users with physical disabilities use computer systems without add-on assistive technologies. These users can especially benefit from small changes in interface accessibility. As a case in point, we recently met a manager with Cerebral Palsy who uses a standard PC and off-the-shelf Windows business productivity applications, but navigates almost exclusively via the keyboard because his poor fine motor coordination makes a mouse or trackball difficult to use. When a pointing device becomes necessary, he uses a trackball. As he explained it to us, "I hate the mouse".

Some users with physical disabilities use assistive technologies to aid their interactions (see Tables 1 and 2). Common hardware add-ons include alternative pointing devices such as head tracking systems and joysticks. The MouseKeys keyboard enhancement available for MS Windows, Macintosh, and X Windows-based workstations allows users to move

1. We believe that in many cases, consideration of the issues discussed here will address many of the needs of these users. See Vanderheiden (1992) and Brown (1989) for more information on other disabilities.

the mouse pointer by pressing keys on the numeric keypad, using other keys to substitute for mouse button presses. Because system-level alternatives are available, it is not necessary for *applications* to provide mouse substitutes of their own. The problem of the mouse is a good example of the kind of generic issue that must be addressed at the system rather than application level.

Unfortunately, the MouseKeys feature is often time-consuming in comparison to keyboard accelerators, because it provides relatively crude directional control. For tasks requiring drag and drop or continuous movement (e.g., drawing), MouseKeys is also inefficient. On the other hand, because current systems are designed with the implicit assumption that the user has a mouse or equivalent pointing device, many tasks require selecting an object or pressing a control for which there is no keyboard alternative. In these cases, MouseKeys provides an option. It is clear that future operating environments need to offer effective alternatives for users who may not use a pointing device.

It is important that applications provide *keyboard access* to controls, features, and information in environments that have keyboard navigation¹. Comprehensive keyboard access helps users who cannot use a mouse. Many environments allow users to use tab and arrow keys to navigate among controls in a window, space bar and enter to activate controls, and key combinations to move focus across windows. In some cases, extra engineering may be required to ensure that these features work in all areas of an interface.

In addition to keyboard navigation, keyboard accelerators and mnemonics are also helpful for users with physical disabilities (as well as blind and low vision users). Whenever practical, commonly used actions and application dialogs should be accessible through buttons or keyboard accelerators. Unfortunately few of the standard accelerator sequences were designed with disabilities in mind. Many key combinations are difficult for users with limited dexterity (e.g., in Motif, holding down Alt-Shift-Tab to change to the previous window in Motif). Nonetheless, use of key mapping consistent with guidelines for the local application environment not only speeds use of applications for users with movement difficulties, but it also increases the effectiveness of alternate input technologies such as speech recognition. Assistive technologies often allow users to define macro sequences to accelerate common tasks. The more keyboard access an application provides, the greater the user's ability to customize assistive technology to work with that application.

About Repetitive Strain Injuries (RSI)

Perhaps the fastest increasing disability in today's computerized workplace is repetitive strain injury (RSI). The Occupational and Health Safety Administration reported that 56 percent of all work place injuries reported during 1992 were due to RSI, up from 18 percent in 1981 (Furger, 1993). RSI is a cumulative trauma disorder that is caused by frequent and regular intervals of repetitive actions. Common repetitive stress injuries are tendonitis and carpal tunnel syndrome, although other types of injuries also occur. Symptoms of com-

1. The lack of keyboard control navigation in some user interface environments is an accessibility problem that needs to be addressed by the designers of these environments.

puter based RSI include headaches, radiating pain, numbness, tingling, and a reduction of hand function. For computer users, mouse movements and typing may be causes or contributors to RSI.¹

Sauter, Schliefer, and Knutson (1991) found that repetitive use of the right hand among VDT data entry operators was a factor in causing RSI. They suggested that a change to “more dynamic tasks” could help reduce the likelihood of RSI. In general, users should be given the choice of performing a task using a variety of both mouse and keyboard options. For custom applications involving highly repetitive tasks, consider providing automatic notification for users to take breaks at regular intervals if there is no such capability at the system level.

Frequently repeated keyboard tasks should not require excessive reach or be nested deep in a menu hierarchy. We once met a customer support representative who had an RSI-related thumb injury. One of her most common jobs tasks -- changing screens to register information from phone calls -- encouraged an extremely wide stretch of her left thumb and forefinger in order to press a control and function key simultaneously. This thumb eventually required surgery.

The needs of users already having symptoms of RSI overlap significantly with the needs of users with other types of physical disabilities. Assistive technologies such as alternate pointing devices, predictive dictionaries, and speech recognition can benefit these users by saving them keystrokes, reducing or eliminating use of the mouse, and allowing different methods of interacting with the system.

TABLE 1. Assistive Technologies for Physical Disabilities and RSI

Assistive Technology	Function Provided
Alternate Pointing Device	Gives users with limited or no arm and hand fine motor control the ability to control mouse movements and functions. Examples include foot operated mice, head-mounted pointing devices and eye-tracking systems.
Screen Keyboard	On-screen keyboard which provides the keys and functions of a physical keyboard. On-screen keyboards are typically used in conjunction with alternate pointing devices.
Predictive Dictionary	Predictive dictionaries speed typing by predicting words as the user types them, and offering those words in a list for the user to choose.
Speech Recognition	Allows the user with limited or no arm and hand fine motor control to input text and/or control the user interface via speech.

1. We discuss software strategies for reducing RSI in this section. Note that there are a wide variety of commercial ergonomic keyboards and alternative input devices aimed at reducing RSI.

TABLE 2. Keyboard Enhancements

Feature	Function Provided
StickyKeys	Provides locking or latching of modifier keys (e.g., Shift, Control) so that they can be used without simultaneously pressing the keys. This allows single finger operation of multiple key combinations.
MouseKeys	An alternative to the mouse which provides keyboard control of cursor movement and mouse button functions.
RepeatKeys	Delays the onset of key repeat, allowing users with limited coordination time to release keys.
SlowKeys	Requires a key to be held down for a set period before keypress acceptance. This prevents users with limited coordination from accidentally pressing keys.
BounceKeys	Requires a delay between keystrokes before accepting the next keypress so users with tremors can prevent the system from accepting inadvertent keypresses.
ToggleKeys	Indicates locking key state with a tone when pressed, e.g., Caps Lock.

About Low Vision

Users with low vision have a wide variety of visual capabilities. According to Vanderheiden (1992), there are approximately 9-10 million people with low vision. For the purposes of this chapter, consider a person with low vision to be someone who can only read print that is very large, magnified, or held very close.

We recently met a user who can read from a standard 21" monitor by using magnifying glasses and the largest available system fonts. His colleague down the hall must magnify text to a height of several inches high using hardware screen magnification equipment. A secretary we met has "tunnel vision", and can see only a very small portion of the world, as though she were "looking through a straw". In the region where she can see, her low acuity requires her to magnify text so that only a portion of her word processing interface is visible on-screen at any one time, reducing her view of the interface to only a single menu or control at a time. These are only a few of the wide variety of low vision conditions.

The common theme for low vision users is that it is challenging to read what is on the screen. All fonts, including those in text panes, menus, labels, and information messages, should be easily configurable by users. There is no way to anticipate how large is large enough. The larger fonts can be scaled, the more likely it is that users with low vision will be able to use software without additional magnification software or hardware. Although many users employ screen magnification hardware or software to enlarge their view, performance and image quality are improved if larger font sizes are available prior to magnification.

A related problem for users with low vision is their limited field of view. Because they use large fonts or magnify the screen through hardware or software, a smaller amount of information is visible at one time. Some users have tunnel vision that restricts their view to a small portion of the screen, while others require magnification at levels that pushes much

of an interface off-screen.

A limited field of view means that these users easily lose context. Events in an interface outside of their field of view may go unnoticed. These limitations in field of view imply that physical proximity of actions and consequences is especially important to users with low vision. In addition, providing redundant audio cues (or the option of audio) can notify users about new information or state changes. In the future, operating environments should allow users to quickly navigate to regions where new information is posted.

Interpreting information that depends on color (e.g, red=stop, green=go) can be difficult for people with visual impairments. A significant number of people with low vision are also unable to distinguish among some or any colors. As one legally blind user who had full vision as a child told us, his vision is like "watching black and white TV". In any case, a significant portion of any population will be "color blind". For these reasons, color should never be used as the only source of information -- color should provide information that is redundant to text, textures, symbols and other information.

Some combinations of background and text colors can result in text that is difficult to read for users with visual impairments. Again, the key is to provide both redundancy and choice. Users should also be given the ability to override default colors, so they can choose the colors that work best for them.

About Blindness

There is no clear demarcation between low vision and true blindness, but for our purposes, a blind person can be considered to be anybody who does not use a visual display at all. These are users who read braille displays or listen to speech output to get information from their systems.

Screen reader software provides access to graphical user interfaces by providing navigation as well as a braille display or speech synthesized reading of controls, text, and icons. The blind user typically uses tab and arrow controls to move through menus, buttons, icons, text areas, and other parts of the graphic interface. As the input focus moves, the screen reader provides braille, speech, or non-speech audio feedback to indicate the user's position (see Mynatt 1994). For example, when focus moves to a button, the user might hear the words "button -- Search", or when focus moves to a text input region, the user might hear a typewriter sound. Some screen readers provide this kind of information only in audio form, while others provide a braille display (a series of pins that raise and lower dynamically to form a row of braille characters).

Blind users rarely use a pointing device, and as discussed above, typically depend on keyboard navigation. A problem of concern to blind users is the growing use of graphics and windowing systems (Edwards, et al, 1992). The transition to window-based systems is an emotional issue, evoking complaints from blind users who feel they are being forced to use an environment that is not well-suited to their style of interaction. As one blind user put it, "This graphics stuff gives sighted people advantages...its user friendly...all this makes it user unfriendly for us [blind people]".

Although blind users have screen reading software that can read the text contents of buttons, menus, and other control areas, screen readers cannot read the contents of an icon or image. In the future, systems should be designed that provide descriptive information for all non-text objects. Until the appropriate infrastructure for providing this information becomes available, there are some measures that may help blind users access this information. Software engineers should give meaningful names for user interface objects in their code. Meaningful names can allow some screen reading software to provide useful information to users with visual impairments. Rather than naming an eraser graphic "widget5", for example, the code should call it "eraser" or some other descriptive name, that users will understand if spoken by a screen reader.

Without such descriptive information, blind or low vision users may find it difficult or impossible to interpret unlabeled, graphically labeled, or custom interface objects. Providing descriptive information may provide the only means for access in these cases. As an added selling point to developers, meaningful widget names make for code that is easier to document and debug.

In addition to problems reading icons, blind users may have trouble reading areas of text that are not navigable via standard keyboard features. In OpenWindows and MS Windows, for example, it is not possible to move the focus to footer messages. If this capability were built into the design, then blind users could easily navigate to footer messages in any application and have their screen reading software read the content.

TABLE 3. Assistive Technologies for Low Vision and Blind Users

Assistive Technology	Function Provided
Screen Reader Software	Allows user to navigate through windows, menus, and controls while receiving text and limited graphic information through speech output or braille display.
Braille Display	Provides line by line braille display of on-screen text using a series of pins to form braille symbols that are constantly updated as the user navigates through the interface.
Text to Speech	Translates electronic text into speech via a speech synthesizer.
Screen Magnification	Provides magnification of a portion or all of a screen, including graphics and windows as well as text. Allows user to track position of the input focus.

About Hearing Disabilities

People with hearing disabilities either cannot detect sound or may have difficulty distinguishing audio output from typical background noise. Because current user interfaces rely heavily on visual presentation, users with hearing related disabilities rarely have serious problems interacting with software. In fact, most users with hearing disabilities can use off-the-shelf computers and software. This situation may change as computers, telephones, and video become more integrated. As more systems are developed for multimedia, desktop videoconferencing, and telephone functions designers will have to give greater consideration to the needs of users with hearing impairments.

Interfaces should not depend on the assumption that users can hear an auditory notice. In addition to users who are deaf, users sitting in airplanes, in noisy offices, or in public places where sound must be turned off also need the visual notification. Additionally, some users can only hear audible cues at certain frequencies or volumes. Volume and frequency of audio feedback should be easily configurable by the user.

Sounds unaccompanied by visual notification, such as a beep indicating that a print job is complete, are of no value to users with hearing impairments or others who are not using sound. While such sounds can be valuable, designs should not assume that sounds will be heard. Sound should be redundant to other sources of information. On the other hand, for the print example above, it would be intrusive for most users to see a highly visible notification sign whenever a printout is ready. Visual notices can include changing an icon, posting a message in an information area, or providing a message window as appropriate.

Again, the key point here is to provide users with options and redundant information. Everybody using a system in a public area benefits from the option of choosing whether to see or hear notices. When appropriate, redundant visual and audio notification gives the information that is necessary to those who need it. If visual notification does not make sense as a redundant or default behavior, then it can be provided as an option.

Other issues to consider include the fact that many people who are born deaf learn American Sign Language as their first language, and English as their second language. For this reason, these users will have many of the same problems with text information as any other user for whom English is a second language, making simple and clear labeling especially important.

Currently, voice input is an option rather than an effective general means of interaction. As voice input becomes a more common method of interacting with systems, designers should remember that many deaf people have trouble speaking distinctly, and may not be able to use voice input reliably. Like the other methods of input already discussed, speech should not be the only way of interacting with a system.

TABLE 4. Assistive Technologies for Hearing Disabilities

Assistive Technology	Function Provided
Telecommunications Device for the Deaf (TDD)	Provides a means for users to communicate over telephone lines using text terminals.
Closed Captioning	Provides text translation of spoken material on video media. Important computer applications include distance learning, CD-ROM, video teleconferencing, and other forms of interactive video.
ShowSounds	Proposed standard would provide visual translation of sound information. Non-speech audio such as system beeps would be presented via screen flashing or similar methods. Video and still images would be described through closed captions or related technologies. This capability would be provided by the system infrastructure.

Design Guidelines

We have taken the design issues discussed in this chapter and condensed them into a list of guidelines contained in Table 5. This table also indicates which users are most likely to benefit from designs that follow these guidelines.

TABLE 5. Design Guidelines

Design Guideline	Physical	RSI	Low Vision	Blind	Hearing
Provide keyboard access to all application features	X	X	X	X	
Use a logical tab order (left to right, top to bottom or as appropriate for locale)	X			X	
Follow key mapping guidelines for the local environment	X	X	X	X	
Avoid conflicts with keyboard accessibility features (see Table 6)	X			X	
Where possible, provide more than one method to perform keyboard tasks	X	X			
Where possible, provide both keyboard and mouse access to functions	X	X	X	X	
Avoid requiring long reaches on frequently performed keyboard operations for people using one hand.	X	X			
Avoid requiring repetitive use of chorded keypresses	X	X			
Avoid placing frequently used functions deep in a menu structure	X	X	X	X	
Do not hard code application colors			X		
Do not hard code graphic attributes such as line, border, and shadow thickness			X		
Do not hard code font sizes and styles			X		
Provide descriptive names for all interface components and any object using graphics instead of text (e.g, palette item or icon)				X	
Do not design interactions to depend upon the assumption that a user will hear audio information					X
Provide visual information that is redundant with audible information					X
Allow users to configure frequency and volume of audible cues			X	X	X

Existing Keyboard Access Features

Designers of Microsoft Windows, Macintosh, and X Windows applications should be aware of existing key mappings used by access features built into the Macintosh and X Windows (and optionally available for MS Windows). These features provide basic keyboard accessibility typically used by people with physical disabilities (see table 2).

In order to avoid conflicts with current and future access products, applications should avoid using the key mappings indicated in Table 6.

TABLE 6. Reserved Key Mappings

Keyboard Mapping	Reserved For
5 consecutive clicks of shift key	On/ Off for StickyKeys
Shift key held down 8 seconds	On/Off for SlowKeys and RepeatKeys

TABLE 6. Reserved Key Mappings

Keyboard Mapping	Reserved For
6 consecutive clicks of control key	On/Off for screen reader numeric keypad
6 consecutive clicks of alt key	Future Access use

INFRASTRUCTURE FOR ACCESS

We have identified a number of approaches to improve the accessibility of applications, but these approaches leave many open issues. Even if all of these strategies were adopted for every current and future interface design, accessibility is ultimately limited by the capabilities of system infrastructure to support communication between assistive technologies and applications.

Recently, some progress has been made towards providing assistive access infrastructure. Parts of the OS/2 operating system, for example, were designed with blind access in mind (Emerson, Jameson, Pike, Schwerdtfeger, and Thatcher, 1992). The operating system infrastructure allows the IBM Screen Reader for Presentation Manager to obtain information about which window is receiving input, the contents of that window, and the layout of that window. For blind users, this infrastructure support means that they can obtain screen information that might be inaccessible on other systems (including font sizes and styles as well as icon positions and labels).

In its most recent release, the X Window system has incorporated some similar access design features for users who are blind or have a physical disability (Edwards, Mynatt, and Rodriguez, 1993; Walker, Novak, Tumblin, and Vanderheiden, 1993). As a consequence, any developer of screen reading software can now develop X Window screen readers that can determine the state and content of windows.

Vanderheiden (1992b) proposed a standard cross-platform method for providing visual presentation of auditory information on computers. This strategy is based on a “ShowSounds” flag which would be set by the user. The ShowSounds capability would allow users who are deaf, hearing impaired, or in quiet environments to request all information presented auditorially to be presented visually. Multimedia applications, for example, would provide captioning for voice while alternative visual presentations would be available for non-speech audio. Such capabilities would benefit users learning a language, working in multilingual environments, as well as deaf users.

Future Directions

ShowSounds is an example of the system-level approach to providing access that is the key to improving accessibility. While current direct and assistive access capabilities are positive steps, much more can be done to improve accessibility. In the remainder of this section, we discuss some of the areas where system infrastructures must be developed to provide support for access.

Users with low vision, for example, typically view a relatively small portion of their display. Often they do not see color, and because of high magnification are unable to see new information or interface changes that are occurring off-screen. In the future, users should have the capability to quickly navigate to regions where new information is posted. Future design efforts also need to explore how interfaces can gracefully handle a wider variety of viewing environments.

Note that some desktop environments (notably the Macintosh) do not provide comprehensive keyboard navigation capabilities allowing tab and arrow navigation through menus and controls. Assistive technologies requiring control navigation operate in these environments by providing their own keyboard navigation. This approach increases the potential for incompatibilities with application key functions, and may not work effectively across all applications.

MS Windows, OpenWindows, and Motif all provide keyboard navigation, but none of these environments provides a standard method for navigating to read-only interface regions such as labels, message bars, and footer messages, nor do they provide a method for obtaining information about graphics. As a near-term solution, systems need to allow users to navigate to read-only text areas, and provide descriptive information for all non-text objects. Over the longer term, systems should be designed so they present and accept information in multiple modalities.

Blind users, for example, would benefit from both speech and non-speech audio information based on video and graphics content. As in the case of closed captioning, such alternative forms of information presentation provide much more than access for people with certain disabilities. Audio descriptions of visual information would allow many users to access and manipulate graphical and video information. over conventional telephones or while working on eyes-busy tasks.

Current interfaces are primarily visual, but as telephones become integrated with computers and multimedia becomes ubiquitous, access barriers for hearing impaired users will increase. Future systems should include support of closed captioning of video. Closed captioning is useful not only for users with hearing disabilities, but also for users learning a second language and for a variety of contexts in which multiple languages are used. In addition, the descriptive text accompanying closed captioned video can be indexed to allow full-text searching of video and audio sources.

Looking farther in the future, adaptive interfaces along with intelligent agents can benefit users with disabilities by allowing them to interact with systems in the way best suited for each user (see Kuhme, 1993; Puerta, 1993). Unlike today's computers, future systems will allow most users to step up to any system and use it in a way best suited to their individual needs. We claim that an essential requirement for making this kind of adaptive system possible is to ensure that it accounts for the needs of users with disabilities.

CONCLUSIONS

As with any other issue in human-computer interaction, the key to understanding the problem of accessibility is to understand the needs of users. We have suggested that users

with disabilities are generally not considered in interface design. A shift from the narrow view of who constitutes “the user” to a broad view is the first step towards improving the accessibility of human-computer interaction for all users. Towards this end, we have presented guidelines to increase the accessibility of applications designed for current environments. While much can be done by simply paying attention to the needs of users with disabilities, significant progress towards computer accessibility awaits improvements in the overall accessibility of the user environments.

Much is understood about the needs of users with disabilities that is not addressed by current systems. It is well understood, for example, that blind users need auditory and braille access to graphic information, that deaf users need visual access to audio information, and that users with physical disabilities often need alternative means of interacting with their systems. To address these needs, future operating system infrastructures must be designed with accessibility in mind. To the extent possible, users should be able to “step up” to systems and use them without any adjunct assistive software or hardware. Where assistive software or hardware is required, system infrastructure should provide support for it to interoperate transparently with existing applications.

Multimedia and related future technologies hold both danger and promise for future prospects of accessible human-computer interaction. The danger is that without realizing the potential roadblocks presented by these emerging audio and video technologies, interface barriers far more daunting than those today will be designed into future applications and environments. Many users could be locked out by interfaces that are usable only by people with a wide range of sensory and motor capabilities.

The promise is that these same mix of audio and video capabilities can significantly increase the accessibility of human-computer interaction. By providing the appropriate infrastructure to support multiple redundant input and presentation of information, systems of the future can allow users with disabilities to interact on their own terms while providing information in the form most useful to them. Not only will these users benefit, but all users will benefit from interacting with systems based on their needs and context.

REFERENCES

Arons, Barry. *Tools for Building Asynchronous Servers to Support Speech and Audio Applications*, 5th Annual Symposium on User Interface Software and Technology, 71-78. ACM Press, 1992.

Blattner, M. M., Glinert, E.P., Jorge, J.A., and Ormsby, G.R., *Metawidgets: Towards a theory of multimodal interface design*. Proceedings: COMPASAC 92, pp 115-120, IEEE Press, 1992.

Brown, C. *Computer Access in Higher Education for Students with Disabilities*, 2nd Edition. George Lithograph Company, San Francisco. 1989.

Brown, C. *Assistive Technology Computers and Persons with Disabilities*, *Communications of the ACM*, pp 36-45, 35(5), 1992.

Casali, S.P., and Williges, R.C., Data Bases of Accommodative Aids for Computer Users with Disabilities, *Human Factors*, pp 407-422, 32(4), 1990.

Church, G., and Glennen, S. *The Handbook of Assistive Technology*, Singular Publishing Group, Inc, San Diego, 1992.

Edwards, W. K., Mynatt, E. D., Rodriguez, T. The Mercator Project: A Nonvisual Interface to the X Window System. *The X Resource*. O'Reilly and Associates, Inc. April, 1993.

Edwards, A., Edwards, E., and Mynatt, E. *Enabling Technology for Users with Special Needs*, InterCHI '93 Tutorial, 1993.

Elkind, J. The Incidence of Disabilities in the United States, *Human Factors*, pp 397-405, 32(4), 1990.

Emerson, M., Jameson, D., Pike, G., Schwerdtfeger, R., and Thatcher, J. *Screen Reader/PM*. IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1992.

Fels, D., Shein, G.F., Chignell, M.H., and Milner, M. Feedback Control: Whose Job is it Anyway?, pp 81-84, 25th Annual Conference of the Human Factors Association of Canada, 1992.

Furger, Roberta. Danger at Your Fingertips. *PC World*, pg. 118, 11(5), 1993.

Glinert, E.P., and York, B.W. Computers and People with Disabilities, *Communications of the ACM*, pp 32-35, 35(5), 1992.

Griffith, D. Computer Access for Persons who are Blind or Visually Impaired: Human Factors Issues. *Human Factors*, pp 467-475, 32(4), 1990.

Kuhme, T. A User-Centered Approach to Addaptive Interfaces. *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*. pp 243-246. Orlando, FL. New York: ACM Press 1993.

Lazzaro, Joseph J. *Adaptive Technologies for Learning and Work Environments*. American Library Association, Chicago and London, 1993

Lewis, C., and Rieman, J. *Task-Centered User Interface Design*. Electronic Shareware Publication, 1993.

Managing Information Resources for Accessibility, U.S. General Services Administration Information Resources Management Service, Clearinghouse on Computer Accommodation, 1991.

McCormick, John A. *Computers and the American's with Disabilities Act: A Manager's Guide*. Windcrest, 1994.

McMillan, W.W. *Computing for Users with Special Needs and Models of Computer-Human Interaction*. Conference on Human Factors in Computing Systems, CHI '92, pp. 143-148. Addison Wesley, 1992.

Mynatt, E. *Auditory Presentation of Graphical User Interfaces* in Kramer, G. (ed), *Auditory Display: Sonification, Audification and Auditory Interfaces*, Santa Fe. Addison-Wesley: Reading MA., 1994.

Newell, A.F., and Cairns, A. Designing for Extraordinary Users. *Ergonomics in Design*, October, 1993.

Nielsen, J. *Usability Engineering*. Academic Press, Inc., San Diego. 1993.

Perritt Jr., H.H. *Americans with Disabilities Act Handbook*, 2nd Edition. John Wiley and Sons, Inc, New York, 1991.

Puerta, A.R. The Study of Models of Intelligent Interfaces. *Proceedings of the 1993 International Workshop on Intelligent User Interfaces..* pp. 71-80. Orlando, FL. New York: ACM Press

Quindlen, T.H., Technology Outfits Disabled Feds for New Opportunities on the Job, *Government Computer News*, 1993.

Sauter, S.L., Schleifer, L.M., and Knutson, S.J. Work Posture, Workstation Design, and Musculoskeletal Discomfort in a VDT Data Entry Task. *Human Factors*, pp 407-422, 33(2), 1991.

Schmandt, C. Voice Communications with Computers, *Conversational Systems*. Van Nostrand Reinhold, New York, 1993.

Tognazzini, Bruce. *Tog on Interface*. Addison-Wesley Publishing Company, Inc., 1992.

Vanderheiden, G.C., Curbcuts and Computers: Providing Access to Computers and Information Systems for Disabled Individuals. *Keynote Speech at the Indiana Governor's Conference on the Handicapped*, 1983.

Vanderheiden, G.C., Thirty-Something Million: Should They be Exceptions? *Human Factors*, 32(4), 383-396. 1990.

Vanderheiden, G.C. *Accessible Design of Consumer Products: Working Draft 1.6*. Trace Research and Development Center, Madison, Wisconsin, 1991.

Vanderheiden, G.C. *Making Software more Accessible for People with Disabilities: Release 1.2*. Trace Research and Development Center, Madison, Wisconsin, 1992a.

Vanderheiden, G.C. *A Standard Approach for Full Visual Annotation of Auditorially Presented Information for Users, Including Those Who are Deaf: ShowSounds*. Trace Research & Development Center, 1992b.

Walker, W.D., Novak, M.E., Tumblin, H.R., Vanderheiden, G.C. Making the X Window System Accessible to People with Disabilities. *Proceedings: 7th Annual X Technical Conference*. O'Reilly & Associates, 1993.

Sources for more information on Accessibility

Clearinghouse on Computer Accommodation (COCA) 18th & F Streets, NW Room 1213 Washington, DC 20405 (202) 501-4906

A central clearinghouse of information on technology and accessibility. COCA documentation covers products, government resources, user requirements, legal requirements, and much more.

Sensory Access Foundation 385 Sherman Avenue, Suite 2 Palo Alto, CA 94306 (415) 329-0430

Non-profit organization consults on application of technology "to increase options for visually and hearing impaired persons". Publishes newsletters on assistive technology.

Trace Research and Development Center S-151 Waisman Center 1500 Highland Avenue Madison, WI 53528 (608) 262-6966

A central source for the current information on assistive technologies as well as a major research and evaluation center. Trace distributes databases and papers on assistive technology and resources.

Conferences

CSUN Conference on Technology and Persons with Disabilities. Every Spring in Los Angeles, California in the USA. Phone: (818) 885-2578

Closing the Gap Conference on Microcomputer Technology in Special Education and Rehabilitation. Every Fall in Minneapolis, Minnesota in the USA. Phone: (612) 248-3294

RESNA. Conference on rehabilitation and assistive technologies. Every Summer. Location varies. Phone: (703) 524-6686

ASSETS. Annual meeting of ACM's Special Interest Group on Computers and the Physically Handicapped. Every Fall. Location varies. Phone: (212) 626-0500

The Java Reliable Multicast Service™: A Reliable Multicast Library

Steve Hanna, Miriam Kadansky and Phil Rosenzweig

Introduction by Phil Rosenzweig and Miriam Kadansky

This technical report, written in 1998, is an early description of our four-year project developing a reliable multicast toolkit. Our goal was to enable developers to be able to easily create applications that use reliable multicast, such as multi-receiver file transfers, stock and news tickers, and collaborative applications. Unreliable multicast, in which data can be sent to many receivers at once, but with no guarantee of delivery, had been deployed in some applications, but the lack of reliability prevented multicast from being used for any commercial applications. Adding reliability would make multicast viable in the business world, marrying the bandwidth savings of multicast with the guaranteed delivery of unicast transport protocols such as TCP.

We were aware that many reliable multicast transport protocols were being developed, and soon decided that there would likely be no one-size-fits-all protocol suitable for use by all types of multicast applications. Therefore, unlike the unicast world in which most applications use the TCP transport protocol, JRMS™ allows developers to choose from a variety of protocols depending upon the application's needs. In addition to transport protocols, the toolkit also provides other services for reliable multicast applications, such as address allocation, security, group management, data filtering, and session advertisement.

Early on we decided to do all development in the Java™ programming language, primarily for its ease of portability, but also to experiment with the viability of writing a transport protocol in Java. Any reliable multicast application was bound to be used on a variety of platforms, and Java is an obvious way to ensure rapid deployment. We also had a set of new ideas for a scalable reliable multicast protocol. After much study of the three dozen or so already proposed, we designed, implemented, and published Tree-Based Reliable Multicast, or TRAM for short. Developing our own protocol gave us a context in which we explored many of the issues central to making reliable multicast work: security, congestion and flow control, acknowledgement implosion, data filtering, and group management. From this work, several key innovations were invented. Currently, TRAM has been combined with aspects of other tree-based reliable multicast protocols into the TRACK (Tree-Based Acknowledgement Protocol) protocol, and is under consideration for standardization within the Reliable Multicast Transport Working Group of the Internet Engineering Task Force (IETF). See <http://www.ietf.cnri.reston.va.us/html.charters/rmt-charter.html>

The JRMS project resulted in many prototype applications, as well as patents and publications. It is available at <http://www.experimentalstuff.com>

REFERENCES:

A Congestion Control Algorithm for Tree-based Reliable Multicast Protocols, Dah Ming Chiu, Miriam Kadansky, Joe Provino, Joseph Wesley, Hans-Peter Bischof and Haifeng Zhu, Sun Microsystems Laboratories Technical Report TR-2001-97

A Reliability Window for Flexible and Scalable Multicast Service, Dah Ming Chiu and Jaiwant Mulik, Sun Microsystems Laboratories Technical Report TR-2000-91

Pruning Algorithms for Multicast Flow Control, Dah Ming Chiu, Miriam Kadansky, Joe Provino, Joseph Wesley and Haifeng Zhu, Sun Microsystems Laboratories Technical Report TR-2000-85, also published in Networked Group Communication 2000, November 8 – 10, 2000, Stanford, CA

Some Observations on Fairness of Bandwidth Sharing, Dah Ming Chiu, Sun Microsystems Laboratories Technical Report TR-99-80

Experiences in Programming a Traffic Shaper, Dah Ming Chiu, Miriam Kadansky, Joe Provino and Joseph Wesley, Sun Microsystems Laboratories Technical Report TR-99-77; also published in the Fifth IEEE Symposium on Computers and Communications (ISCC 2000), 4 – 6 July 2000, Antibes, France

The Java Reliable Multicast Service: A Reliable Multicast Library, Steve Hanna, Miriam Kadansky and Phil Rosenzweig, Sun Microsystems Laboratories Technical Report TR-98-68

TRAM: A Tree-based Reliable Multicast Protocol, Dah Ming Chiu, Stephen Hurst, Miriam Kadansky and Joseph Wesley, Sun Microsystems Laboratories Technical Report TR-98-66

The Java™ Reliable Multicast™ Service: A Reliable Multicast Library

Phil Rosenzweig, Miriam Kadansky, and Steve Hanna

SMLI TR-98-68

September 1998

Abstract:

The Java™ Reliable Multicast Service™ (JRMS) is a set of libraries and services for building multicast-aware applications. It enables building applications that distribute data from “senders” to “receivers” over “channels” with distributed control over content mix. It includes a dynamic filtering mechanism that uses Java software that is pushed into the network for interpreting the data of the receiver. JRMS supports multiple reliable multicast transport protocols via a common programming interface, which provides isolation to applications. Supported transport protocols are selectable by applications based on reliability and type of service needs.

Emerging multimedia or “push” applications can use JRMS as a better platform for reliable delivery of content to very large constituencies. As compared to unicast (point-to-point) protocols), reliable multicast enables broadcasting to groups of receivers, ensuring bandwidth conservation and timely delivery. The JRMS reliable multicast transport protocol (TRAM) is designed for high scalability. JRMS also includes related services for multicast address allocation, channel advertisement and subscription, authentication and encryption, and receiver group management. JRMS is designed to be a flexible toolkit to the application developer for authoring new types of network-centric multimedia applications.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:

phil.rosenzweig@east.sun.com
miriam.kadansky@east.sun.com
steve.hanna@east.sun.com

The JavaTM Reliable MulticastTM Service: A Reliable Multicast Library

Phil Rosenzweig, Miriam Kadansky, and Steve Hanna

Sun Microsystems Laboratories
Boston Center For Networking
Chelmsford, Massachusetts

1 Introduction

1.1 What is IP Multicast?

IP Multicast is an internet protocol that enables transmission of data packets to a group of receivers. It is described in RFC 1112 [MULTICAST]. Datagrams are sent to a shared class D IP address to which all group members (or hosts) are listening. Hosts may attach to the group by joining or detach from the group by leaving. These actions associate or disassociate the host from the class D multicast IP address. In order for multicast to work, IP routers need to recognize hosts that have joined the multicast group and forward datagrams to each of their downstream links that have hosts on the multicast group. The IGMP protocol [IGMP] provides a mechanism for hosts to communicate with their associated routers to join and leave multicast groups. See [RTNG] for a complete description of IP Multicast routing.

1.2 Why IP Multicast?

Basic TCP/IP network protocols address the need for communicating between one sender and one receiver. This type of communication is known as “unicast” or point-to-point. When a Web browser is pointed at a Web site, TCP/IP provides a unicast connection to enable communications.

There is an emerging class of applications designed for collaborative computing, streaming multimedia, and real-time data delivery. These require multi-point communications between one or more senders and several receivers in order to work well. In this mode of operation, instead of a

receiver explicitly requesting data from a sender, data is sent from a sender to the receivers asynchronously as needed. IP Multicast is designed to solve this problem.

For example, when groups of people share data that changes often, it may be necessary to notify everyone when the data changes. Unless it is possible to broadcast changes to all group members at the same time, notifying each member in turn is inefficient. This would require individual point-to-point connections in order for members to receive updates from other members. Obviously, this won't scale beyond a handful of members.

1.3 IP Multicast Issues

Several challenging issues related to IP Multicast have emerged. The following subsections discuss these issues.

1.3.1 Multicast Routing Protocols

DVMRP [DVMRP], Sparse-mode PIM [PIM-SM], Dense-mode PIM [PIM-DM], MOSPF [MOSPF], and CBT [CBT] are the most common multicast routing protocols in use today. On the Mbone, the Internet's experimental multicast backbone [MBONE], DVMRP is most widely deployed. Each of these protocols is designed to deliver packets from senders to receivers on a class D IP Multicast group address.

Shortest path protocols are designed to find the shortest path through the routing infrastructure between each sender and each of its receivers. Shared tree protocols attempt to determine a common path which is shared by all senders on a multicast group. Each approach requires some amount of network traffic and elapsed time in order to build the initial routes (or tree) between

the sender(s) and receivers, and optimize for best path depending on the approach. If group membership is fluid because of frequent or continuous joins and leaves, or additional new senders join the group, or the group membership is very large, routing instability may result.

For example, because DVMRP is a distance vector protocol, it has shown scalability problems in environments such as the MBone which contain large numbers (8000+) of subnets. This limitation is one of the key driving forces behind the transition from DVMRP to a more scalable multicast routing protocol [BGMP] on the MBone.

1.3.2 Product Issues

Although multicast routing protocols are reasonably well-defined, product issues remain in various categories of network equipment. This is because there are very few applications exercising the current infrastructure. These issues will get resolved as more multicast-aware applications become deployed. Some example product issues are:

NICs: Network interface cards (NICs) are the hardware and software devices which connect network nodes to multi-access media networks such as ethernet. NICs usually provide a level of support in hardware for receiving packets on the node's IP address, broadcast address, and multicast address(es). Multicast address support can vary from none to some upward limit. For example, SBUS cards for Sun workstations and servers typically support up to 20 multicast addresses. Beyond this limit requires filtering incoming packets by address in software, which is expensive from a performance perspective.

Switches: LAN switches sometimes treat multicast packets in the same manner as broadcast packets, which results in flooding all ports. This is usually considered a bug.

Routers: Significant bugs exist in many of the multicast routing implementations. In particular, experience with DVMRP on the MBone indicates some unpredictability and instability in its operation, causing some network managers to be reluctant to enable multicast routing in their own networks.

TCP/IP Stacks: Most current TCP/IP stacks support multicast. The key capability is support for IGMP, which is the protocol used for joining

and leaving multicast groups. Older implementations often do not support IGMP.

1.3.3 Scalability

Router Tables: As multicast is increasingly deployed, more router table space will be consumed. Internal data structures that occupy router memory are created to indicate where to forward multicast packets. Routers on the tree for a particular multicast group must record state for that multicast group. In addition, depending on the multicast routing protocol used, a router may keep state for multicast groups even if it is *not* involved in transmitting data for the group. For backbone routers, if a large number of concurrent multicast groups are in use, there is the potential for exhaustion of table memory space or available processor bandwidth, which can limit scalability. The MBone routers on the Internet best illustrate this condition.

Congestion Control: The lack of congestion control mechanisms in multicast transports can cause inefficiencies in the bandwidth utilization of the network. This is especially dangerous since a single multicast group can affect large portions of a network. Other traffic, such as well-behaved TCP sessions, may be endangered by poorly behaved multicast flows.

1.3.4 Address Allocation

IP Multicast [IP] identifies a set of IP Multicast addresses (224.0.0.0 to 239.255.255.255) that are used to identify a group of recipients (a multicast group). An IP packet sent to one of these addresses is delivered to all members of the multicast group. These groups may span several IP networks.

Certain multicast addresses, 224.0.0.0 to 224.255.255.255, are reserved by the Internet Assigned Numbers Authority (IANA¹) for special purposes. Other addresses, known as administratively-scoped addresses [ADMIN] (239.0.0.0 to 239.255.255.255), are reserved for use within a predefined administrative scope, such as site-local or organization-local. The rest, known as globally-scoped addresses, operate with a global scope.

At the moment, there are several techniques for allocating multicast addresses, but they all have serious disadvantages and they are not always

¹ See <http://www.iana.org>

used. This can lead to collisions where two multicast groups are established independently with the same multicast address assigned to both, possibly causing data intended for one group to be received by members of another group. The ideal address allocation scheme should scale up to the size of the Internet and guarantee unique addresses across all networks for a particular session. Although IPv4 allows 2^{28} multicast addresses, reuse is required in order not to run out of addresses over time.

1.3.5 Session Directories

Currently, end-users find out about multicast sessions by consulting a session directory [SDR]. Directory entries are multicast over a well-known multicast address using the Session Announcement Protocol [SAP] and the Session Description Protocol [SDP]. Information associated with particular multicast sessions, such as group address, start time or public key, may be made available by the session directory. These directories that are responsible for advertisement of available multicast sessions must scale or otherwise become an inhibitor to multicast deployment. Unless this information is highly available, potential receivers for individual multicast sessions may not receive timely notification in which case they would not be able to participate. Handley [SCALE] describes the scalability considerations with session directories.

1.3.6 TTL Scoping

Another issue is scoping of multicast groups. It is often useful to contain transmission of multicast traffic to a subset of a network. The Time To Live (TTL) field in the header of IP packets is commonly used to limit the number of router hops a packet is to be forwarded. This field is decremented each time a packet is forwarded by a router. At zero, the packet is discarded. If the TTL is too small, packets may not reach every desired part of the network. If the TTL is too large, multicast packets may be forwarded farther than required. In general, the TTL should be large enough to reach all members of a multicast group.

Several reliable multicast transport protocols adjust the TTL field of repair packets to limit their scope. However, using TTL to contain retransmission of packets to a subset of a multicast group does not necessarily confine the data to the targeted receivers.

1.3.7 Security

In a multicast environment, any node can join a multicast group, then send or receive packets. This is a major concern for applications that require security, either to keep data from being seen by unauthorized observers, to ensure that data is sent only by authorized senders, or just to prevent denial-of-service attacks caused by senders flooding the group with data.

For unicast applications, data can be kept confidential and senders can be authenticated using various encryption and signature techniques. While these techniques can also be used for multicast data, it is more difficult to distribute keying information in a multicast environment. Key distribution and revocation algorithms are complex as they need to consider an arbitrary number of senders and receivers, not just two.

1.4 Reliable Multicast

1.4.1 Introduction

Basic IP Multicast is unreliable. This does not mean that it fails often! It means that if any data is lost during transmission, there is no mechanism for repairing this loss. For video, this just causes a momentary glitch. For some other data (like a spreadsheet), it renders the data useless.

Reliable multicast adds a way to repair lost or damaged multicast data. When a receiver misses one or more data packets, it needs these packets to be retransmitted. This sounds easy, but it's not. In fact, it is an area of active research, with new ideas and protocols coming out all the time. The challenges of supporting reliable multicast include:

- Performing repairs for those receivers that need them
- Not overloading the sender with messages from receivers
- Managing receivers with different abilities (loss rate, bandwidth, latency, etc.)
- Congestion detection, avoidance and control
- Insuring scalability to large receiver groups
- Maximizing throughput
- Handling different types of applications
- Providing data security

1.4.2 JRM Service Objectives

One of the basic assumptions of the JRM Service is that the problem of applying reliability to multicast is best addressed as an end-to-end problem. This means that it is best done from one end system to another, rather than within the routing infrastructure. The need to save state for the purpose of repairing packets on a per packet / per group basis puts an inordinate resource load on routing nodes which typically have limited memory and processor resources. Packet repair requires establishing packet caches at appropriate points throughout the network. Scalability requirements require these caches to be within reasonable proximity of the receivers. Since routers are best-effort packet forwarders and will drop packets if resources become exhausted, higher level protocols must deal with lost packet detection and retransmission when necessary. This would suggest that reliable multicast is best done in higher level protocols and in devices which have sufficient computing resources to provide adequate services.

As seen in the next section, there are a variety of types of applications for reliable multicast. Each of these imposes different requirements on the network. This implies a corresponding variety of reliable multicast protocols in order to address the needs of applications. For example, an application which transfers bulk data or large files from one sender to many receivers (such as software distribution), is very different from a collaborative application, which has multiple senders and receivers (such as sharing a virtual whiteboard). The JRM Service addresses the needs of different application types by providing a framework for including multiple reliable multicast protocols. These protocols are accessible by applications via a common API. Since these protocols are implemented end-to-end, it is straightforward to provide them as they become available and define interfaces for others to plug into. It would be much harder to implement and deploy this mix of protocols in router devices — not only because of the reasons previously discussed, but also because routers are homogeneous devices which are not extensible by others. Router code bases come from a single vendor and tend to be released and installed by customers on long time cycles in a conservative manner. This slow-moving release methodology is analogous to operating systems release, except that operating systems allow users to add drivers and other extensions to provide new functions, within reasonably well-

defined bounds. So, again, this argues that end systems are better than routers for timely deployment of reliable multicast protocol technology.

The objectives for the JRM Service are:

- Provide a platform for building multicast-aware applications organized as a set of libraries and services implemented on end stations
- Provide applications with access to multiple reliable multicast protocols through a common API which isolates applications from protocol changes, yet allows selecting appropriate protocols based on application need
- Design a highly scalable bulk data transfer protocol for large numbers of receivers which includes support for congestion detection, avoidance, and control
- Provide services for multicast address allocation, security (authentication and encryption), and administrative policy control
- Flexibility
 - Implement in Java software for multi-platform support
 - Handle simple or complex applications with scalable overhead for supporting services
 - Add new reliable multicast protocols with minimal impact to applications
 - Provide a mechanism for applying administrative policy to data at the receivers (dynamic filters)

2 Reliable Multicast Applications

There are several distinct types of multicast-aware applications. These are characterized by factors relating to such things as how the application behaves, characteristics of the data transmitted, demand placed on the network, connectivity characteristics such as stationary or mobile, and available bandwidth. This section describes multicast-aware application types and corresponding requirements that characterize them.

2.1 Application Characteristics

The requirements listed below are the key requirements of multicast-aware applications for

the reliable multicast protocols used to support them. These substantially affect the nature and design of the reliable multicast protocols used to address the applications. These requirements are:

Number of Senders: Protocols that allow more than one sender (known as “many senders”) are substantially more complicated than those that allow only one sender (known as “single sender”), as they have no centralized point of control.

Late Joins: Different applications have different requirements for receiver readiness. Receivers that join the group after some data has been sent are known as *late joins*. If all receivers must have a single start time, protocol design is simplified considerably; late joins are not allowed. If receivers may start at any time and must receive copies of all data sent, a good deal of effort must go into preserving all data. This is known as late join with full data recovery. If receivers may start at any time but do not receive copies of all data sent earlier, less effort is required. This is known as late join with limited data recovery.

Real-time Performance: Some applications require that data be delivered within a certain period of time. Others simply require that it be delivered eventually.

Consistency: Requirements in this area differ widely. Some applications require that all data be delivered to all receivers at exactly the same time (such as stock price delivery). Others require simply that all receivers eventually get the data. Some provide application-level checkpoints at which consistency must be guaranteed.

Ordering: Most applications require data to be delivered in order, but some (such as news feeds) do not.

Reliability: Most applications require full reliability, but some (such as streaming video) can trade off a certain amount of data loss in order to achieve high throughput and timely delivery.

2.2 Types of Reliable Multicast Applications

2.2.1 Bulk Data

Bulk data applications are concerned with transferring files or other large blocks of data. Examples include distribution of corporate data, updating software, loading Web caches, or updating data warehouses. Such applications typically require: one sender per channel, no late joins, no real-time requirement, file level consistency, ordering, and full reliability.

2.2.2 Live Data

Live data applications are concerned with distributing an ongoing flow of small pieces of data. Examples include stock price distribution, news feeds, and event logging. Such applications typically require: one sender per channel, no start time (late joins with limited data recovery), varied real-time requirements (strict for stock prices, loose for some others), varying levels of consistency (extremely strict for stock prices, loose for most others), varying levels of ordering (strict for stock prices, none for news feeds), and varying levels of reliability (strict for stock prices, none for event logging).

2.2.3 Resilient Streams

Resilient streams applications (also known as semi-reliable applications) are concerned with distributing a stream of real-time multimedia data that can handle some losses. Examples include video or audio feeds. While such applications may use unreliable multicast, reliable multicast can provide limited data repair and (more important) congestion control. Such applications typically require: one sender per channel, various start times, moderate real-time requirements, no consistency requirements, no ordering requirements, and loose reliability requirements.

2.2.4 Collaborative

Collaborative applications are concerned with sharing data within a group. Examples include a shared whiteboard, collaborative editing, or controlling a multimedia conference. Such applications typically require: many senders per channel, late join with full data recovery, moderate real-time requirements, various consistency requirements, various ordering requirements, and full reliability.

2.2.5 Hybrid

Hybrid applications are some combination of the above. Examples include Distributed Interactive Simulation (DIS) and other shared virtual reality environments. These applications often have many different requirements for different data streams or at different times. Some of them may be implemented as a combination of the above application types. Others may have special requirements that require special solutions (such as rapid joining and leaving of channels).

2.3 Suitability of the JRM Service for Various Applications

The JRM Service is designed to support many of the application types listed above. As can be seen, the architecture of the JRM Service accommodates multiple reliable multicast protocol implementations. This enables the JRM Service to address varying and perhaps conflicting requirements of multicast-aware applications. The JRM Service includes a transport protocol (Tree-based Reliable Multicast Protocol) [TRAM] that satisfies many of the needs of applications such as bulk data. Conversely, collaborative applications such as a shared whiteboard require a protocol that supports many senders. The JRM Service includes one such protocol, Lightweight Reliable Multicast Protocol [LRMP], and a simple *chat* application that supports multiple senders. In all cases, applications use a common API irrespective of the reliable multicast protocol in use.

The TRAM protocol is most useful with bulk data applications. These are typically single-sender applications with many receivers. TRAM supports ordering and reliability. Moderate real-time requirements are addressed in TRAM with

support for maintaining minimum (and maximum) data rates. This is a best effort mechanism to sustain throughput within a range of transmission speeds. More details of the TRAM protocol can be found in section 4.

3 The JRM Service Architecture

3.1 Abstract Model

The abstract model for the JRM Service is that one or more *senders* transmit data on a given *channel* and one or more *receivers* receive this data. In Figure 1, A is a sender on channel E; B, C, and D are receivers. It is possible to have many senders and for a single node to be both a sender and a receiver.

A channel is a multicast transport session that conveys data from one or more senders to multiple receivers. This is conceptually similar to an IP Multicast group, but reliable delivery, security, and other features may be added on.

Each channel has one or more *channel managers* that are responsible for creating and destroying the channel, configuring it, and managing channel membership (that is, controlling who can send and receive on it). In Figure 1, M is the channel manager.

These are the basic components in the JRM Service abstract model: channels, senders, receivers, and channel managers.

3.2 Systems

Figure 2 illustrates the major systems that make up the JRM Service .

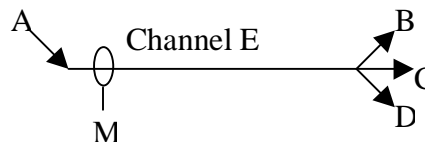


Figure 1: JRM Service Abstract Model

3.2.1 Transport

The transport system is responsible for providing a reliable multicast transport. It may be used directly by the application or indirectly through the channel management system. The transport system is composed of two parts, a set of protocol-independent transport APIs and one or more transport protocols.

3.2.1.1 Transport APIs

Many different reliable multicast transport protocols have been developed and many more are on the way. Currently, application vendors that want to use reliable multicast must develop code specifically to support one protocol. If another protocol comes out that meets their needs better, they must reintegrate their code. The JRM Service provides a interface that allows application vendors to write their code once and have it work with any reliable multicast protocol.

The JRM Service is provided with three protocols implemented: Tree-Based Reliable Multicast [TRAM], Lightweight Reliable Multicast [LRMP], and an unreliable multicast transport. Other protocols can easily be added using stream or packet interfaces.

3.2.1.2 The JRM Service Transport Protocol

The JRM Service Transport Protocol [TRAM] is a reliable multicast protocol designed to support bulk data transfer with a single sender and multiple receivers. TRAM uses dynamic trees to implement local error recovery that scales to a very large number of receivers without imposing a serious burden on the sender. It also includes congestion control and other techniques necessary to operate efficiently and fairly with other protocols across the wide variety of link and client characteristics that make up the Internet.

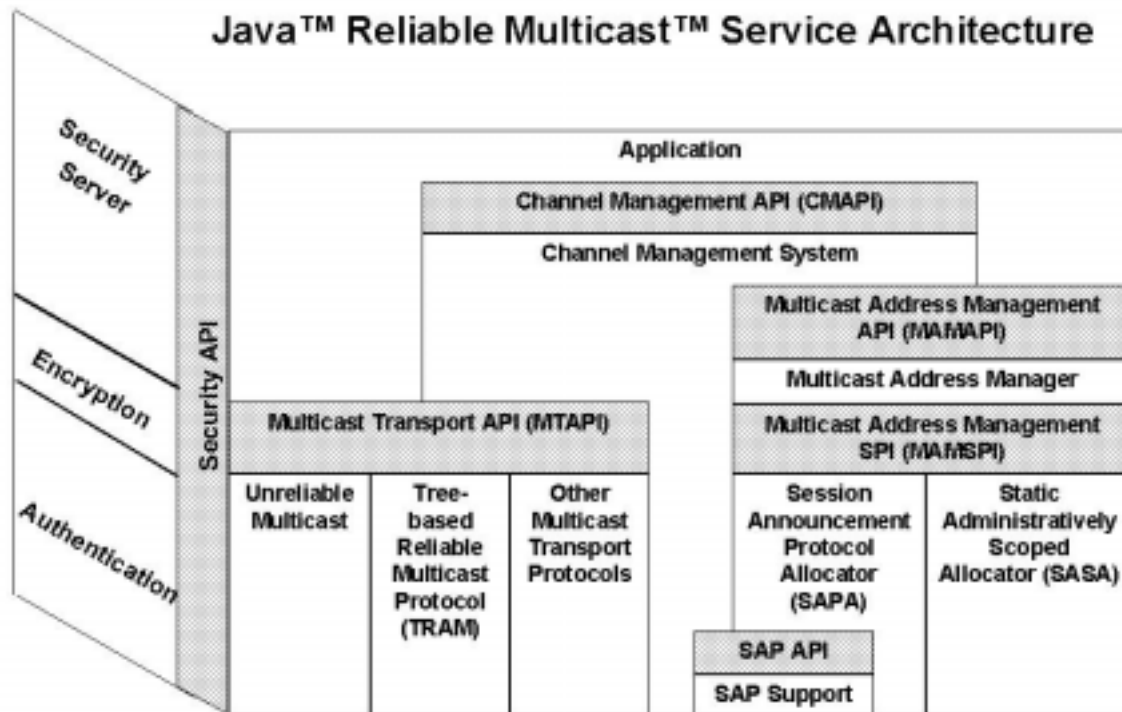


Figure 2: The JRM Service

3.2.2 Channel Management

The channel management system is responsible for creating, configuring, and destroying channels upon request. It handles end-to-end management of the multicast session, including session advertisement, discovery and joining, receiver authentication, event notification and distribution of dynamic filters. Operations at the channel layer are generic to the transport protocol in use.

A channel may be configured with one or more dynamic filters. A *dynamic filter* is a piece of Java software that runs on a receiver, which may be an end-station or intermediate node, and processes the data received on a channel, transforming it in some way. For instance, a dynamic filter can block access to entertainment channels during working hours. It can also highlight stories that match a user's interest profile.

Dynamic filters are delivered to receivers when they register for a channel. Different receivers may receive different filters, based on access control or preferences.

3.2.3 Address Allocation

The address allocation system is responsible for managing multicast address allocation. It may be used directly by the application or indirectly through the channel management system.

One of the first steps in establishing a new channel is choosing a multicast address. Because multicast addresses are shared across multiple networks, this is a tricky problem. Several techniques are in use today, but no one technique addresses every issue. The address allocation system provides a standard API that may be used to allocate and deallocate addresses. Different address allocators may be installed to implement different allocation techniques.

3.2.4 SAP Support

The SAP Support system provides an implementation of the Session Announcement Protocol [SAP] and Session Description Protocol [SDP]. These protocols are useful both for allocating multicast addresses and for advertising multicast sessions. The SAP Support classes are used by both the Address Allocation and Channel Management systems.

3.2.5 Security

Security solutions for reliable multicast are much more complex than those for unicast [SECURITY]. The JRM Service supports three aspects of security:

- data confidentiality
- data integrity and message authentication
- sender and receiver authentication and access control

Data confidentiality is achieved by encrypting data sent on the channel. Data integrity is achieved by including a message digest. In order to support a wide variety of implementations of data confidentiality and integrity, the JRM Service provides generic interfaces for confidentiality and authentication. Confidentiality is provided by a layer just above the transport; message and sender authentication is supported within the transport layer. The JRM Service provides implementations of several solutions using these open interfaces. New implementations can be plugged into the system via the supported open interfaces.

User authentication, access control, and key distribution are implemented as part of channel management. Senders and receivers identify themselves to the channel manager, and, after authentication, securely receive appropriate keying information. For instance, receivers are sent decryption and signature verification keys, while senders obtain encryption and signing keys.

4 TRAM

4.1 Overview

TRAM is designed to support bulk data transfer with a single sender and multiple receivers. It uses dynamic trees to implement local error recovery and to scale to a large number of receivers without impacting the sender. It also includes flow control, congestion control, and other adaptive techniques necessary to operate efficiently and fairly with other protocols across the wide variety of link and client characteristics that make up the Internet as well as intranets.

TRAM ensures reliability by using a selective acknowledgement mechanism, and scalability by

adopting a hierarchical tree-based repair mechanism. The hierarchical tree not only overcomes implosion-related problems but also enables localized multicast repairs and efficiently funnels feedback to the sender.

The receivers and the data source of a multicast session in TRAM interact with each other to dynamically form repair groups. These repair groups are linked together in a hierarchical manner to form a tree with the sender at the root of the tree. Figure 3 shows a typical TRAM repair tree. These types of trees have been shown to be the most scalable way of supporting reliable multicast transmissions [SURVEY]. Every repair group has a receiver that functions as a group head; the rest function as group members. These members are said to be affiliated with their head. Except for the sender, every repair group head in the system is a member of some other repair group. All members receive data multicast by the sender. The group members report lost and successfully received messages to the group head using a selective acknowledgement mechanism similar

to TCP's [SACK]. The repair heads cache every message sent by the sender and provide repair service for messages that are reported as lost by the members. The dynamic nature of the tree allows it to react to changes in the underlying network infrastructure without sacrificing reliability.

The TRAM tree formation algorithm works in both bi-directional multicast environments and uni-directional multicast environments (such as satellite links). Repair nodes are elected based on a wide spectrum of criteria, and the tree is continuously optimized based on the receiver population and network topology. The acknowledgement-reporting mechanism is window-based with optimizations to reduce burstiness and processing overhead. The flow control mechanism is rate-based and adapts to network congestion. The sender senses and adjusts to the rate at which the receivers can accept the data. Receivers that cannot keep up with the minimum data rate can be dropped from the repair tree.

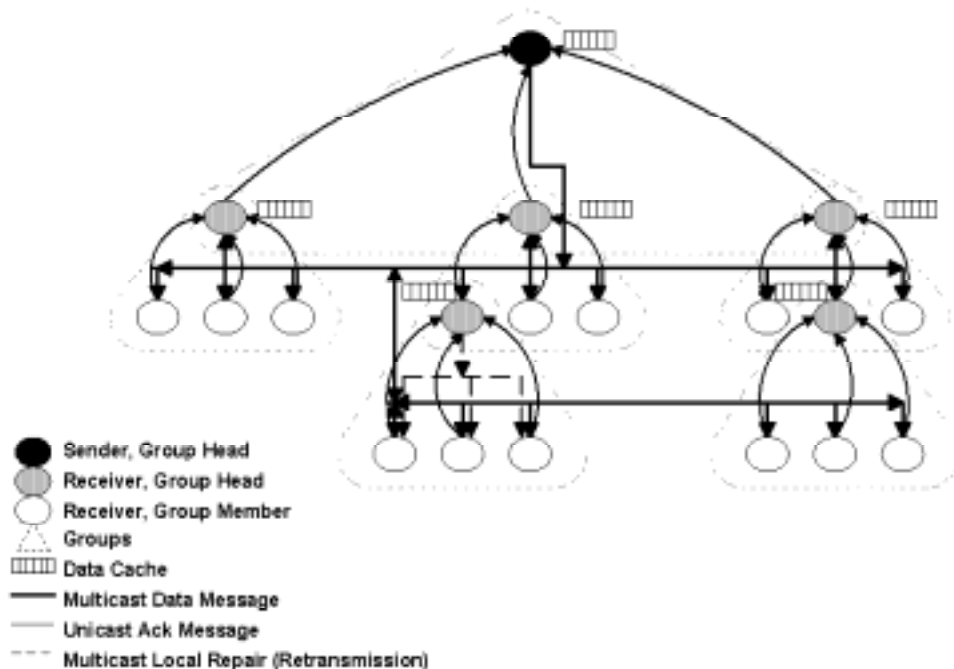


Figure 3: Typical Hierarchical Repair Tree in TRAM

Some of the major features of TRAM are:

Reliability: TRAM guarantees delivery of data to any receiver that joins the tree and is able to keep up with the minimum transmission speed specified by the sender. While this level of guarantee cannot protect applications from delivery failure, features of the JRM Service can be used to closely keep track of individual member's status.

Avoiding Acknowledgement Implosion: Point-to-point transports achieve reliability by acknowledging receipt of data directly to the sender. Unfortunately, this does not scale over multicast. A sender trying to field acknowledgements from many receivers will quickly become overloaded. TRAM avoids this implosion by dynamically designating a receiver to be a repair head for a group of members. The repair head fields acknowledgements from each of its group members and supplies them with repair packets. To avoid overload, each repair head is responsible for only a limited number of members.

Local Repair: TRAM builds the tree so that repair heads are close to their members. This enables repair heads to perform repairs using small time-to-live (TTL) values, which not only minimizes network bandwidth consumption but also avoids unnecessary processing at receivers not requiring repair.

Automatic Configuration: Different applications, networks, or user groups may perform better using different types of trees. TRAM provides for building different types of trees optimized for these different circumstances without adding complexity to the network infrastructure itself. As a receiver group changes, TRAM also provides for ongoing optimization of an existing tree, for instance, enabling a member to find a better repair head. A particular tree's formation method may also be changed at any time during the transmission at the discretion of the sender. TRAM's tree management works in networks providing only unidirectional multicast, as well as those supporting bi-directional multicast.

Rate-based Flow Control and Congestion Avoidance: TRAM schedules packet transmission according to a data rate. This data rate is dynamically adjusted based on congestion feedback from the receivers. Congestion feed-

back is aggregated by repair heads through the tree. The algorithm used to adjust the rate works in different network topologies. The rate is bounded by a maximum and minimum rate configured at the sender.

Feedback to the Sender: Each member of the tree periodically reports statistics to its repair head. This includes statistics that assist in building the tree (for instance, the number of available repair heads on the tree) as well as reports on congestion conditions, which allow the sender to adapt its data rate to network conditions. This information is aggregated at each level of the tree in order to reduce control traffic to the sender.

Controlling Memory Use: Each repair head is responsible for ensuring that the data is received by every member. This means that a repair head must cache data until it is sure that all of its members have received it. TRAM requires positive acknowledgements from members when data is received. This enables repair heads to reclaim cache buffers containing data that has been received by all members.

Repair Group Management: Both members and repair heads monitor each other to detect unreachability. Non-responsive members can be dropped from the repair group and corresponding cache buffers can be reclaimed. Non-responsive repair heads can be abandoned by their members in favor of an active repair head. Repair heads are also responsible for detecting receivers which cannot keep up with the minimum transmission rate specified by the sender. While such members cannot be dropped from the multicast group, they can be denied repair head support and receive no repairs.

Scalability: TRAM has been designed to be scalable in many situations, such as large numbers of receivers and sparsely or densely populated receiver groups. TRAM also accommodates wide ranges of receiver capabilities. Control message traffic is designed to be limited in all of these cases.

4.2 Data Transmission

The TRAM sender transmits data at a rate that adjusts automatically between a specified minimum and maximum. Sequence numbers in the data packets allow receivers to identify

missing packets. Each member is bound to a repair head that retransmits lost packets when requested. Acknowledgements sent by receivers contain a bitmap indicating received and missing packets. Missing packets are repaired by the repair head; packets received by all members are removed from the repair head's cache.

4.3 Limitations

Realistic TTL Values: Tree formation in TRAM relies on TTLs. Current multicast routing protocols may not provide accurate TTL values for this purpose.

Heterogeneous Receiver Population: Although TRAM adjusts itself to the capabilities of the receivers as much as possible, a particular slow receiver may end up without repair service if it cannot keep up with the minimum speed specified by the sender according to administrative policy.

5 Channel Management

The JRM Service provides an abstraction layer known as a “channel” for applications to deliver data to multiple receivers. In the protocol stack, it is layered above the transport. This allows channel to provide a common set of functions to applications irrespective of which transport protocols are in use. For example, one of channel's functions is authentication of receivers. This would be available to collaborative applications using a many-to-many transport protocol, such as LRMP, and to news distribution applications using a one-to-many transport protocol, such as TRAM. The choice of transport protocol can differ without affecting the functions provided by the channel layer.

Channels provide the following functions:

- Channel discovery
- Channel registration and authentication of channel users
- Event notification
- Application of policy through dynamic filtering
- Access to security services

A key motivation behind providing the channel layer comes from the requirements of applications moving from unicast to multicast. In unicast, there is a point-to-point connection

between a single pair of nodes in a network. If a banking application distributes updated currency information to each of its remote offices over a network at the end of each day, it can make a unicast connection to each office in turn, transfer the data and log the fact that it did so. In multicast, senders and receivers can be anonymous to each other, given the way nodes join and leave multicast groups. If one were to improve the banking application by moving it to multicast, how would it be able to log that it transferred data to each of the remote offices? One way would be for each of the remote offices to tell the sender by establishing a unicast connection with the sender when transmission is complete. This, of course, would somewhat defeat the purpose of using multicast. A better way would be to provide the ability for receivers to be identified and remembered as part of the initialization process of the session so that the sender would know each of the receivers and be able to log the information. This somewhat recreates, for reliable multicast, the implicit knowledge that a pair of nodes has of each other's identities in a unicast environment. Channel has discovery and registration functions to provide this capability for applications.

5.1 Channel Discovery, Registration and Authentication

A useful way to think about the concept of a channel is to visualize it as a shared medium to which all users of the channel are attached. Data sent on a channel is visible to all that have subscribed to it. Users (applications running on network nodes) can be senders, receivers or both. The channel itself is an entity that has associated characteristics. Applications discover the existence of channels, then find out the characteristics of the channels in order to use them. An application can create a channel, use it for the duration of the application session and then destroy it. Channels can also persist indefinitely and be reused by more than one application.

5.1.1 Discovery

Potential users of a channel can discover it in several ways: by listening for session announcements, by querying a channel manager or by static configuration. Using a well-known multicast address, sessions can be advertised via the Session Announcement Protocol [SAP].

Session announcements contain information necessary for an application to become a user of the channel. This information is packaged into a structure called a *channel profile*. Since applications in a multicast environment are distributed, this mechanism provides for synchronization of users. This is often useful for bulk data transfer applications that require all receivers to be ready and waiting prior to starting data transmission. Among other things, channel advertisements contain a channel name and a multicast address and port for the session. They may also contain session start time information, security information, and other optional fields. The JRM Service uses the Session Description Protocol [SDP] to format session advertisement messages. The use of SAP and SDP allows leveraging existing mechanisms used on the MBone for announcing multicast sessions.

Another way to discover channels is for a potential channel user to contact a channel manager. Channel managers are processes in the network that provide functions for maintaining channels. Session details are cached by channel managers and are subsequently available to requestors once properly authenticated. A potential user of a channel could query a channel manager for information about a channel whose channel name is known. The channel manager then provides the information necessary for the application to attach to and become a user of the channel.

Finally, channel information can be statically configured within applications themselves. This would alleviate the need for SAP announcements or channel manager queries but it is the least flexible. However, it may be appropriate for certain applications.

5.1.2 Registration and Authentication of Users

Once a potential user of a channel discovers the channel and its information, the next step is registration. To do this, it makes a registration call to a channel manager. The channel manager performs authentication to be sure the user has appropriate permissions to access the channel, records the event that this user has attached and, if configured, downloads any dynamic filters to the user application which may be necessary for administrative control over the channel data itself (see section 5.3). The channel manager may also send security information, such as encryption and authentication keys, to the

registered receiver (see section 7). Registration may also trigger events for which others can be waiting. For instance, the registration mechanism allows senders to wait for all receivers, or certain receivers, to register before transmitting data.

During the session, it may be required that receivers re-register. Re-registration causes users to contact their channel manager using the same registration calls performed the first time. This provides a way to apply new or changed configuration data during a session. For example, one way to change data encryption keys during a session is to require each receiver to periodically re-register and securely download new keys.

5.2 Event Notification

Channel provides interfaces that may be used to inform applications of various events. For instance, events are available to enable an application to monitor a particular user; the application can be informed when the user registers for a session, or when it completes reception of some data. Events can also be used to monitor the session as a whole, for instance, keeping track of how many users are registered at any particular time.

5.3 Dynamic Filters

Dynamic filters are designed to provide the flexibility to apply administrative control over the content of a channel. One or more dynamic filters may be installed on a receiver during the process of registering for a channel. Dynamic filters are pre-configured Java software classes which are associated with one or more receivers. Typically, they are downloaded to receivers from a service running on a server in the network. This service can be co-located with the channel manager or not but must be accessible by the channel manager for retrieving dynamic filters at registration time. The JRM Service stores dynamic filters in a database. Caching dynamic filters or otherwise making them persistent at the receiver is a possible enhancement.

Once dynamic filters are installed, they have access to data packets received from the network but not yet presented to the application. In the current implementation, dynamic filters are called on a per-packet basis which allow them to filter, transform or augment the data and return the result. Multiple dynamic filters may be

installed at a receiver in which case each is called in turn to process the received data packets. When the last dynamic filter completes, the data is presented to the application.

Some dynamic filter operations may require caching packets until a logical block of data is present in receiver memory. If, for example, a dynamic filter is performing language translation, it may require more than one packet at a time. Other examples may include encryption/decryption operations, forward error correction or distributed guaranteed delivery algorithms for time-sensitive data.

Application designers may choose to use dynamic filters as a way for end-users to customize preferences for their data. Additionally, network management personnel may apply centralized control for groups of end-users using dynamic filters to limit access to certain content based on local policies. The dynamic filtering feature is designed to be a flexible mechanism for accommodating many diverse requirements of multicast-aware applications.

6 Address Allocation

Allocating and managing multicast IP addresses is a tricky problem. Unlike a unicast IP address, data sent to a multicast address may be sent to receivers on many different networks. In fact, the set of receivers may change over time. If two senders use the same multicast address accidentally, the data they send may be mingled, causing application confusion. Therefore, it is important to manage multicast addresses carefully so that this doesn't happen.

Several multicast address allocation techniques are currently in use. Administratively scoped multicast addresses are often allocated with manual systems such as a list maintained by a network administrator. Non-administratively scoped addresses are allocated via multicast advertising with the Session Announcement Protocol (SAP). This allows for global dynamic allocation, but it will only scale to about 10,000 addresses [HANDLEY].

The Multicast Address Allocation Working Group of the IETF is working on solutions that will scale to millions of addresses. Their current architecture calls for a three-tiered model with interdomain allocation handled by MASC [MASC], intradomain allocation handled by AAP [AAP], and a host request protocol named MDHCP [MDHCP]. The JRM Service team is playing an active role in development of the new IETF protocols, and implementations of these protocols are being considered.

The JRM Service addresses the multicast address allocation problem in three ways. First, it provides a standard API that may be used to request and manage addresses. This insulates application vendors from any changes in address allocation techniques. Second, it provides support for allocating addresses from a predefined pool of administratively scoped addresses, set aside exclusively for the use of a single JRM Service channel manager. Third, it provides an implementation of SAP allocation.

The multicast address allocation APIs provided by the JRM Service are the Multicast Address Management API (MAMAPI), which applications use to allocate multicast addresses, and the Multicast Address Management Service Provider Interface (MAMSPI), which allows any multicast address allocator to be plugged in.

7 Security

The JRM Service security interfaces support both data confidentiality and data integrity. These interfaces expand on existing unicast security methods, enabling their use with multicast. Keys are managed by security servers (see section 7.5), which create, distribute, and update keys used by the channel. Senders and receivers use an encryption layer above the transport for confidentiality, and authentication services at the transport layer for message and sender authentication. Although the JRM Service provides a number of security methods, not all may be available for export. Figure 4 shows an overview of JRM Service security.

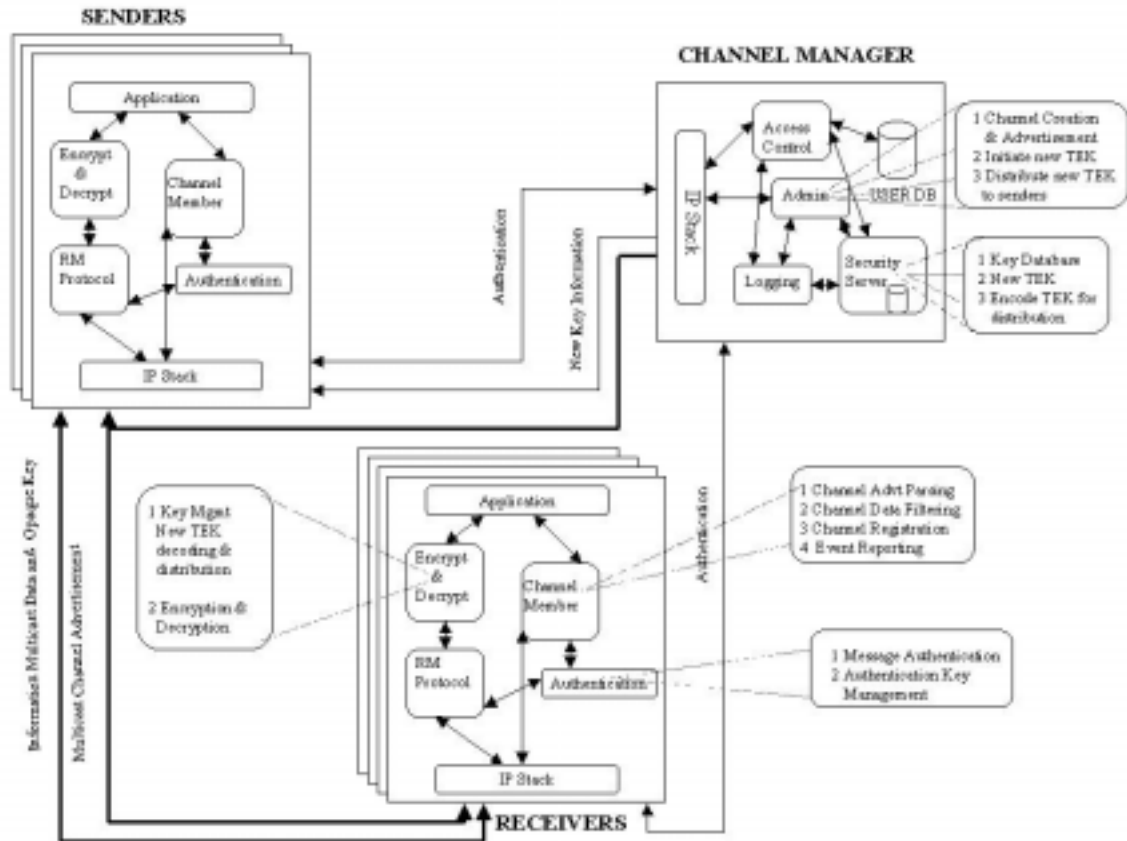


Figure 4: JRM Service Security Architecture

7.1 Data Confidentiality

Data confidentiality ensures that channel data can only be viewed by subscribers of the channel. At channel registration time, each registered sender or receiver is given the current text encryption keys. These are usually secret keys shared by the entire multicast group. Senders use these keys to encrypt data, and receivers use them to decrypt data.

7.2 Data Integrity

Data integrity guarantees the receiver that a message was not modified in transit. To do this, the sender computes a digest of the message, encrypts the computed digest and includes the encrypted value in the message that is to be sent. Receivers compute the digest of the message and compare the computed digest against the decrypted digest extracted from the received

message. Messages that fail the comparison test are considered altered in transit and are discarded. At channel registration time, each registered receiver is given keys to verify digests from each authorized sender. The channel may require a separate key for each sender, or it may allow all of the senders to use the same key.

These keys can be the public key of each sender, or a new public key generated for the channel's use. Senders use their private key to generate a signature of the data, and receivers use the corresponding public key to verify the signature.

Alternatively, shared secret keys may be used, either a separate one for each sender, or one key used by all senders. In this case, the same key is used to both encrypt and decrypt. However, public/private key pairs are often used, since, as described in the next section, they also provide sender authentication.

7.3 Sender Authentication

Sender authentication guarantees the receiver that a message came from the indicated sender. Usually a public/private key pair is used. At channel registration time, each registered receiver is given the public key of each authorized sender. The channel may require a separate key pair for each sender, or it may allow all of the senders to use the same key pair. These keys can be the public key of each sender, or a new public key generated for the channel's use. Senders use the private key to generate a signature of the data, and receivers use the corresponding public key to verify the signature. In practice, such signature generation can take a considerable amount of processing, so, instead of signing the entire message, the sender first generates a digest of the message (a much quicker operation), then encrypts the digest. Receivers decrypt the digest, then validate the digest by regenerating it from the data.

7.4 Rekeying

The authentication and encryption mechanisms described above are very similar to those in use for unicast connections. The main issue for these mechanisms in a multicast environment is rekeying. If a unicast session suspects that a key has been compromised, it is not difficult for the two parties to confidentially generate a new key. In a multicast environment, it is not so easy. In addition, a multicast application may have several additional reasons to redistribute keying information. For instance, the application may be paid for on a daily basis, requiring that any encryption keys be changed on a daily basis. Or, an application may require that a new member not be able to see any data sent before it joined. In this case, since the new member may request that old data be repaired, the encryption key must be changed any time a new member joins the group.

If the group has a small number of members, each one could be made to re-register and then receive new keying information, as if it had just joined the group. However, this may not scale well enough to larger groups, and could take some time. Another option is to use one of several innovative proposed algorithms [KEYING] that allow key managers to multicast information to the receivers that allows them to recalculate new keys. This information is cleverly encrypted to prevent excluded users

from decrypting it. JRM Service applications using these types of algorithms can use the channel to reliably send out the rekeying information.

Other algorithms may be used for data confidentiality and integrity; the JRM Service interfaces make the actual method used transparent to the JRM Service code itself. The implementor need only conform to general interfaces for encryption/decryption, signature generation/verification, and key management.

7.5 Security Servers

Security servers are used by channel managers to produce and distribute keys. Standard Java APIs accommodate implementations supporting most security strategies. Security servers within channel managers support distribution of keys to new users, as well as rekeying.

7.6 Authentication Services

The JRM Service provides authentication services solely within the transport layer. For reliable multicast, it is essential that authentication be done in the transport itself to avoid feeding invalid packets to the upper layers. By integrating authentication into the transport, the transport's reliability features can be used to recover the legitimate version of an altered packet. If an altered packet is accepted and passed to an upper layer for authentication, there is no possibility of replacing it with a legitimate one.

8 Conclusions

This paper describes the architecture and major features of a reliable multicast system designed to support the needs of new types of information delivery applications. We believe that a multicast-based system is inherently better than an HTTP-based system for pushing event-based data over a computer network. The results show better scalability through more efficient bandwidth utilization of the network and timely delivery of the data. When reliability is applied, this technology enables dissemination of mission-critical business data that expands the use of multicast beyond audio and video applications.

The JRM Service can be thought of as interesting “middleware”. Since the JRM Service approaches the problem of adding reliability to multicast as an end-to-end problem, it layers software between the IP stack of the end-system and the application. The design center of the JRM Service is to isolate the application as much as possible from the details of the network through a cleanly layered set of APIs. Alternative approaches require that applications be intertwined with the network software. This makes them more difficult to write, given all of the considerations about the network that must be taken into account. However, such intertwined solutions have the potential to be more intelligent about such things as repairing lost data in collaborative applications, for example.

The JRM Service uses IP multicast. This raises two major problems. The first concerns the multicast routing protocols themselves: how well they work and how well they scale to large networks. The second is the availability of multicast. Many corporate network administrators have yet to enable it in their network routers, and many ISPs are evaluating the business case to offer multicast service and the costs of doing so. These issues will be resolved over time and multicast will be ubiquitous on corporate networks in the next couple of years and in widespread use on the Internet someday. Ubiquity on the Internet is unclear — perhaps within the realms of ISPs is most realistic.

One of the key objectives of the JRM Service was to develop a reliable multicast protocol [TRAM] that enables large-scale distribution of data. Through design, implementation, analysis and experimentation, we have learned how hard this problem is. In real networks there is always loss occurring somewhere in the network. Throughput bandwidth is variable when there is some level of network load. This required consideration of complex algorithms for congestion detection, avoidance and control as well as protocol overhead to construct, maintain and optimize the repair tree. As the number of receivers grows, the protocol overhead must be minimized so as not to consume any significant bandwidth on the network. A mechanism to deal with slow receivers was needed to allow disconnection of receivers from the repair tree which were too slow to keep up, thereby causing the whole transmission to all receivers to lock-

step with the slowest receiver. This mechanism enables applications to decide how best to tradeoff completion of the overall transmission and error recovery. In addition, the TRAM protocol provides for transmission between a minimum and maximum speed. This is somewhat novel because a majority of IP network applications have been traditionally built to use as much bandwidth as they need and the network will provide without any regard for other applications or network traffic. Since TCP/IP networks are “best effort delivery,” applications such as ftp use as much bandwidth as the network will provide. The JRM Service goes one step further and tries to hold itself between two bounds, perhaps behaving as a better network citizen but also attempting to give streaming applications a minimum quality of service. Other applications may be built similarly in the future.

9 Future Work

9.1 TRAM

Support for Live and Resilient Data: With a few adjustments, TRAM could support live data transmissions. The primary difference between bulk data and live data is the requirement to view live data in real time. Bulk data can afford to delay repair of a single packet; live data cannot..

Full Data Recovery for Late Joins: Currently TRAM has no guaranteed support for late joiners requiring full data recovery. If a member comes late to a session, it may not find a repair head that has all of the existing session data in its cache.

Congestion Control: Once TRAM is more fully deployed and there are more results concerning the performance of TRAM's congestion control, we expect to make more improvements. In particular, interaction with other network traffic, for instance, TCP, needs further study.

Repair Tree Management: Some situations call for more controlled tree formation. Some possibilities include

- Static assignments of members to repair heads
- Static assignment of standby repair heads

9.2 Channel Management

Caching Dynamic Filters: The current design calls for dynamic filters to be automatically updated by requiring receivers to re-register from time to time. However, it is likely that dynamic filters will not change very often. Caching dynamic filters or otherwise making them persistent at the receiver is a possible enhancement.

9.3 Address Allocation

Emerging Protocols: Address Allocation Protocol [AAP] and Multicast Dynamic Host Configuration Protocol [MDHCP] are currently Internet Drafts. As they move along the standards process, they may replace or supplement The JRM Service's current address allocation techniques.

9.4 SAP Support

JRMS leverages SAP as a mechanism to advertise the existence of multicast sessions. Alternatives or enhancements to SAP must be examined due to the inherent limitations in its ability to scale well once the session count becomes extremely large.

10 Acknowledgements

The authors wish to thank Dah Ming Chiu, Steve Hurst, Radia Perlman, Seth Proctor and Joe Wesley for their contributions to the content of this paper.

11 References

- [AAP] MALLOC Working Group, M. Handley, *Multicast Address Allocation Protocol*, draft-ietf-malloc-aap-01.txt, work in progress, July 1998
- [ADMIN] D. Meyer, *Administratively Scoped IP Multicast*, RFC 2365, July 1998
- [AIM] B. Levine, J. Garcia-Luna-Aceves, *Improving Internet Multicast with Routing Labels*, University of California, Santa Cruz, 1997
- [BGMP] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoglu, D. Estrin, M. Handley, *The MASC/BGMP Architecture for Inter-domain Multicast Routing*, SIGCOMM '98
- [CBT] T. Ballardie, *Core Based Trees (CBT) Multicast Routing Architecture*, RFC 2201, September 1997
- [DVMRP] T. Pusateri, *Distance Vector Multicast Routing Protocol*, draft-ietf-idmr-dvmrp-v3-06, work in progress, March 1998
- [HANDLEY] M. Handley, *On Scalable Multimedia Conferencing Systems*, University of London, November, 1997
- [IGMP] W. Fenner, *Internet Group Management Protocol, Version 2*, RFC 2236, November 1997
- [IP] D. Comer, *Internetworking with TCP/IP, Volume I*, Prentice-Hall, ISBN 0-13-468505-9, 1991
- [KEYING] G. Caronni, M. Waldvogel, D. Sun, B. Plattner, *Efficient Security for Large and Dynamic Multicast Groups*, Proceedings of the Seventh Workshop on Enabling Technologies (WET ICE '98), IEEE Computer Society Press, 1998
- [LMS] C. Papadopoulos, G. Parulkar, G. Varghese, *An Error Control Scheme for Large-Scale Multicast Applications*, Washington University St. Louis, 1997
- [LRMP] T. Liao, *Light-weight Reliable Multicast Protocol as an Extension to RTP*, <http://monet.inria.fr/lrmp/lrmp-rtp.html>
- [MASC] D. Estrin, R. Govindan, M. Handley, S. Kumar, P. Radoslavov, D. Thaler, *The Multicast Address-Set Claim (MASC) Protocol*, draft-ietf-idmr-masc-01, work in progress, August, 1998
- [MBONE] *The Mbone Information Web*, <http://www.mbone.com>
- [MDHCP] S. Hanna, B. Patel, M. Shah, *Multicast Address Allocation based on the Dynamic Host Configuration Protocol*, draft-ietf-malloc-mdhcp-00.txt, work in progress, August, 1998
- [MOSPF] J. Moy, *Multicast Extensions to OSPF*, RFC 1584, March 1994
- [MULTICAST] S. Deering, *Host Extensions for IP Multicasting*, RFC 1112, August 1989
- [PIM-DM] D. Estrin, V. Jacobson, D. Farinacci, D. Meyer, L. Wei, S. Deering, A. Helmy, *Protocol Independent Multicast Version 2 Dense Mode*, draft-ietf-pim-v2-dm-00, work in progress, August 1998
- [PIM-SM] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, L. Wei, *Protocol Independent Multicast - Sparse Mode (PIM-SM); Protocol Specification*, RFC 2117, June 1997
- [RANDOM] A. Costello, *Search Party: An Approach to Reliable Multicast with Local Recovery*, University of California, Berkeley, 1997
- [RTNG] T. Maufer, C. Semeria, *Introduction to IP Multicast Routing*, draft-ietf-mboned-intro-multicast-03.txt, work in progress, July 1997
- [SACK] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, *TCP Selective Acknowledgement Options*, RFC 2018, October 1996
- [SAP] M. Handley, *SAP: Session Announcement Protocol*, draft-ietf-mmusic-sap-03.txt, work in progress, March 1997
- [SCALE] M. Handley, *Session Directories and Internet Multicast Address Allocation*, Computer Communication Review, volume 28, number 4, October 1998

[SDP] M. Handley, V. Jacobson, *SDP: Session Description Protocol*, RFC 2327, April 1998

[SDR] M. Handley, *The sdr Session Directory: An Mbone Conference Scheduling and Booking System*, <http://mice.ed.ac.uk/mice/archive/sdr.html>, April 1996

[SECURITY] C. Kaufman, R. Perlman, M. Speciner, *Network Security: Private Communication in a Public World*, Prentice Hall, 1995

[SURVEY] B. Levine, J. Garcia-Luna-Aceves, *A Comparison of Known Classes of Reliable Multicast Protocols*. University of California, Santa Cruz, 1996

[TRAM] D. Chiu, S. Hurst, M. Kadansky, J. Wesley, *TRAM: a Tree-based Reliable Multicast Protocol*, Sun Microsystems Laboratories, SMLI TR-98-66, July 1998

Growing a Language

Guy L. Steele Jr.

Introduction by Guy L. Steele Jr.

This is the (slightly edited) text of an invited talk at the 1998 ACM OOPSLA conference. Richard Gabriel reports that the fellow sitting next to him, on seeing Steele walk up onto the stage wearing a suit and tie, remarked that they must be in for some sort of marketing talk or sales pitch. Well, he was right, in a way—but the talk was not at all in the form he expected!

This talk argues that most programming languages have impoverished vocabularies that make them awkward to use. The talk also makes a case for designing facilities into programming languages to allow the programmer, not just the language designer, to add new facilities to the language, to grow the language for the benefit of other programmers. What makes the talk unusual is that it is reflexive, or metacircular, serving as an example of the phenomenon being described.

REFERENCES:

Steele, Guy L., Jr. "Growing a Language." Higher-Order and Symbolic Computation 12, 3 (October 1999), Kluwer Academic Publishers (Hingham, MA), 221–236.

Additional Publications by Guy L. Steele Jr.:

Common Lisp: The Language (Digital Press, 1984, 1990)

C: A Reference Manual (Prentice-Hall, 1984, 1986, 1991, 1995)

The High Performance Fortran Handbook (MIT Press, 1994)

The Java™ Language Specification (Addison-Wesley, 1996, 2000)

Growing a Language

Guy L. Steele Jr.

Sun Microsystems Laboratories
1 Network Drive
Burlington, Massachusetts 01803

`guy.steele@sun.com`

October 1998

[This is the text of a talk I once gave, but with a few bugs fixed here and there, and a phrase or two changed to make my thoughts more clear. The talk as I first gave it can be had on tape [12].]

I think you know what a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a *person* is a woman or a man (young or old).

To keep things short, when I say “he” I mean “he or she,” and when I say “his” I mean “his or her.”

A *machine* is a thing that can do a task with no help, or not much help, from a person.

(As a rule, we can speak of two or more of a thing if we add an “s” or “z” sound to the end of a word that names it.)

$$\begin{array}{l} \langle \text{noun} \rangle ::= \langle \text{noun that names one thing} \rangle \text{ “s”} \\ \quad \quad \quad | \langle \text{noun that names one thing} \rangle \text{ “es”} \end{array}$$

These are names of persons: *Alan Turing*, *Alonzo Church*, *Charles Kay Ogden*, *Christopher Alexander*, *Eric Raymond*, *Fred Brooks*, *John Horton Conway*, *James Gosling*, *Bill Joy*, and *Dick Gabriel*.

The word *other* means “not the same.” The phrase *other than* means “not the same as.”

A *number* may be nought, or may be one more than a number. In this way we have a set of numbers with no bound.

$$\begin{array}{l} \langle \text{number} \rangle ::= 0 \\ \quad \quad \quad | 1 + \langle \text{number} \rangle \end{array}$$

There are other numbers as well, but I shall not speak more of them yet.

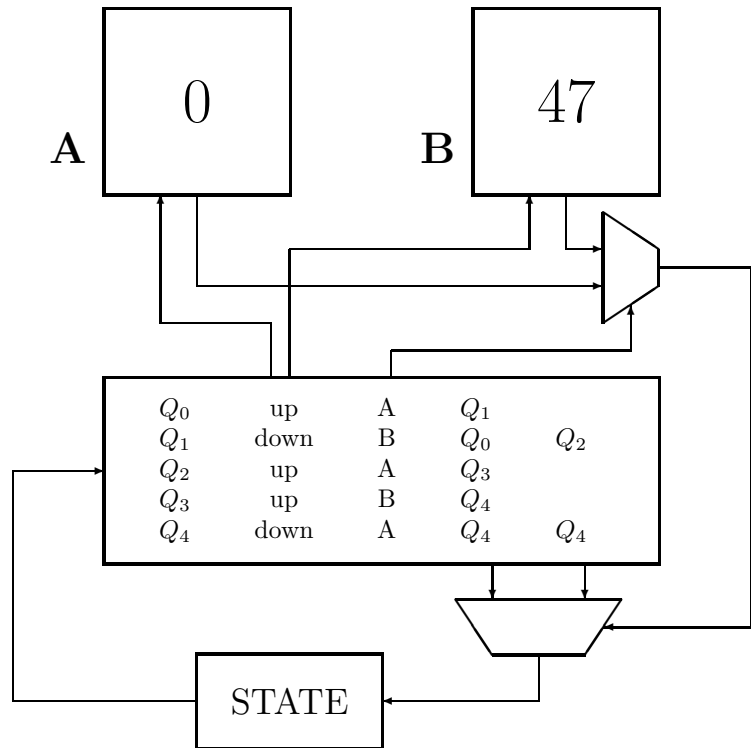
These numbers—nought or one more than a number—can be used to count things. We can add two numbers if we count up from the first number while we count down from the number that is not the first till it comes to nought; then the first count is the sum.

$$4 + 2 = 5 + 1 = 6 + 0 = 6$$

Four plus two is the same as five plus one, which is the same as six plus nought, which is six.

We shall take the word *many* to mean “more than two in number.”

Think of a machine that can keep track of two numbers, and count each one up or down, and test if a number be nought and by such a test choose to do this or that. The list of things that it can do and of choices that it can make must be of a known size that is some number.



Here you can see the two numbers and a list. The machine starts its work with the first row of the list. Each row in the list has a state name; the word “up” or “down”; and which number to count up or count down. For “up,” we have the name of the next state to go to (and the machine counts the number up by one); for “down,” the machine first tests the number, and so we have the name of the new state to go to if the number be nought and the name of the new state to go to if the number be other than nought (in which case the machine counts the number down by one). Note that no two rows of the list have the same state name.

A *computer* is a machine that can do at least what the two number machine can do—and we have good cause to think that if a computer task can be done at all, then the two number machine can do it, too, if you put numbers in and read them out in the right way. In some sense, all computers are the same; we know this thanks to the work of such persons as Alan Turing and Alonzo Church.

A *vocabulary* is a set of words.

A *language* is a vocabulary and rules for what a string of words might mean to a person or a machine that hears them.

To *define* a word is to tell what it means. When you define a word, you add it to your vocabulary and to the vocabulary of each person who hears you. Then you can use the word.

To *program* is to make up a list of things to do and choices to make, to be done by a computer. Such a list is called a *program*.

(A noun can be made from a verb in such a way that it means that which is done as meant by that verb; to make such a noun, we add “ing” to the verb stem. Thus we can speak of “programming.”)

$$\langle \text{noun} \rangle ::= \langle \text{verb stem} \rangle \text{ “ing”}$$

A programming language is a language that we can use to tell a computer a program to do. *Java* is a brand name for a computer programming language. I shall speak of the Java programming language a great deal more in this talk. (I have to say the full phrase “Java programming language,” for there is a guy who works where I do who deals with the laws of marks of trade, and he told me I have to say it that way.) Names of other programming languages are *Fortran*, *APL*, *Pascal*, and *PL/I*.

A part of a program that does define a new word for use in other parts of the program is called a *definition*. Not all programming languages have a way to code definitions, but most do. (Those that do not are for wimps.)

Should a programming language be small or large? A small programming language might take but a short time to learn. A large programming language may take a long, long time to learn, but then it is less hard to use, for we then have a lot of words at hand—or, I should say, at the tips of our tongues—to use at the drop of a hat. If we start with a small language, then in most cases we can not say much at the start. We must first define more words; then we can speak of the main thing that is on our mind.

(We can use a verb in the past tense if we add a “d” or “ed” sound at the end of the verb. In the same way we can form what we call a *past participle*, which is a form of the verb that says of a noun that what the verb means has been done to it.)

$$\begin{array}{l} \langle \text{verb} \rangle ::= \langle \text{verb stem} \rangle \text{ “ed”} \\ \quad \quad \quad | \quad \langle \text{verb stem} \rangle \text{ “d”} \end{array}$$
$$\begin{array}{l} \langle \text{past participle} \rangle ::= \langle \text{verb stem} \rangle \text{ “ed”} \\ \quad \quad \quad | \quad \langle \text{verb stem} \rangle \text{ “d”} \end{array}$$

For this talk, I wanted to show you what it is like to use a language that is much too small. You have now had a good taste of it. I must next define a few more words.

An *example* is some one thing, out of a set of things, that I put in front of you so that you can see how some part of that thing is in fact a part of each thing in the set.

A *syllable* is a bit of sound that a mouth and tongue can say all at one time, more or less, in a smooth way. Each word is made up of one or more syllables.

A *primitive* is a word for which we can take it for granted that we all know what it means.

For this talk, I chose to take as my primitives all the words of one syllable, and no more, from the language I use for most of my speech each day, which is called *English*. My firm rule for this talk is that if I need to use a word of two or more syllables, I must first define it. I can do this by defining one word at a time—and I have done so—or I can give a rule by which a new word can be made from some other known word, so that many new words can be defined at once—and I have done this as well.

This choice, to start with just words of one syllable, was why I had to define the word “woman” but could take the word “man” for granted. I wanted to define “machine” in

terms of the word “person”, and it seemed least hard to define “person” as “a man or a woman.” (I know, I know: this does not quite give the true core meaning of the word “person”; but, please, cut me some slack here.) By chance, the word “man” has one syllable and so is a primitive, but the word “woman” has two syllables and so I had to define it. In a language other than English, the words that mean “man” and “woman” might each have one syllable—or might each have two syllables, in which case one would have to take some other tack.

We have to do this a lot when we write real computer programs: a thought that seems like a primitive in our minds turns out not to be a primitive in a programming language, and in each new program we must define it once more. A good example of this is `max`, which yields the more large of two numbers. It is a primitive thought to me, but few programming languages have it as a primitive; I have to define it. This is the sort of thing that makes a computer look like a person who is but four years old. Next to English, all computer programming languages are small; as we write code, we must stop now and then to define some new term that we will need to use in more than one place. Some persons find that their programs have a few large chunks of code that do the “real work” plus a large pile of small bits of code that define new words, so to speak, to be used as if they were primitives.

I hope that this talk brings home to you—in a way you can feel in your heart, not just think in your head—what it is like to have to program in that way. This should show you, true to life, what it is like to use a small language. Each time I have tried this sort of thing, I have found that I can not say much at all till I take the time to define at least a few new terms. In other words, if you want to get far at all with a small language, you must first add to the small language to make a language that is more large.

(In some cases we will find it more smooth to add the syllable “er” to the end of a word than to use the word “more” in front of it; in this way we might say “smoother” in place of “more smooth” or “larger” in place of “more large.” Let me add that *better* means “more good.”)

⟨word that can change what a noun means⟩ ::=
 ⟨word that can change what a noun means⟩ “er”
 ⟨word that can change what a noun means⟩ “r”

In truth, the words of one syllable form quite a rich vocabulary, with which you can say many things. You may note that this vocabulary is much larger than that of the language called *Basic English*, defined by Charles Kay Ogden in the year one nine three nought [10]. I chose not to use Basic English for this talk, for the cause that Basic English has many words of two or more syllables, some of them quite long, handed down to us from the dim past of Rome. While Basic English has fewer words, it does not give the feel, to one who speaks full English, of being a small language.

By the way, from now on I shall use the word *because* to mean “for the cause that.”

If we look at English and then at programming languages, we see that all our programming languages seem small. And yet we can say, too, in some cases, that one programming language is smaller than some other programming language.

A *design* is a plan for how to build a thing. To *design* is to build a thing in one's mind but not yet in the real world—or, better yet, to plan how the real thing can be built.

The main thing that I want to ask in this talk is: If I want to help other persons to write all sorts of programs, should I design a small programming language or a large one?

I stand on this claim: I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow—but I need, too, to leave some choices so that other persons can make those choices at a later time.

This is not a new thought. We have known for some time that huge programs can not be coded from scratch all at once. There has to be a plan for growth. What we must stop to think on now is the fact that languages have now reached that large size where they can not be designed all at once, much less built all at once.

Let me pause to define some names for numbers. *Twenty* is twice ten. *Thirty* is thrice ten. *Forty* is twice twenty. A *hundred* is ten times ten. A *million* is a hundred times a hundred. *Eleven* is ten plus one. *Thirteen* is ten plus three. *Fourteen* is ten plus four. *Sixteen* is twice eight. *Seven* is one plus six. *Fifty* is one more than seven squared. One more thing: *ago* means “in the past, as one counts back from now.”

In the past, it made sense to design a whole language, once for all. Fortran was a small language forty years ago, designed for tasks with numbers, and it served well. PL/I was thought a big language thirty years ago, but now we would think of it as small. Pascal was designed as a small, whole language with no plan to add to it at a later time. That was five and twenty years ago.

What came to pass?

Fortran has grown and grown. Many new words and new rules have been added. The new design is not bad; the parts fit well, one with the other. But to many of those who have used Fortran for a long time, the Fortran of here and now is not at all the same as the language they first came to know and love. It looks strange.

PL/I has not grown much. It is, for the most part, just as it was when it first came out. It may be that this is just from lack of use. The flip side is that the lack of use may have two causes. Number one: PL/I was not designed to grow—it was designed to be all things to all who program right from the start. Number two: for its time, it started out large. No one knew all of PL/I; some said that no one *could* know all of PL/I.

Pascal grew just a tad and was used to build many large programs. One main fault of the first design was that strings were hard to use because they were all of fixed size. Pascal would have been of no use for the building of large programs for use in the real world if this had not been changed. But Wirth had not planned for the language to change in such ways, and in fact few changes were made.

At times we think of C as a small language designed from whole cloth. But it grew out of a smaller language called B, and has since grown to be a larger language called C plus plus. A language as large as C plus plus could not have spread so wide if it had been foisted on the world all at once. It would have been too hard to port.

(One more rule for making words: if we add the syllable “er” to a verb stem, we make a noun that names a person or thing that does what the verb says to do. For example, a buyer is one who buys. A user is one who does use.)

$\langle \text{noun} \rangle ::= \langle \text{verb stem} \rangle \text{ “er”}$

As you may by now have guessed, I am of like mind with my good friend Dick Gabriel, who wrote the well known screed “Worse Is Better” [5, 6]. (The real name was “Lisp: Good News, Bad News, How to Win Big,” which is all words of just one syllable—which might seem like good luck for me, but the truth is that Dick Gabriel knew how to choose words with punch. Yet what first comes to mind for most persons is the part headed “Worse Is Better” and so that is how they cite it.) The gist of it is that the best way to get a

language used by many persons is not to design and build “The Right Thing,” because that will take too long. In a race, a small language with warts will beat a well designed language because users will not wait for the right thing; they will use the language that is quick and cheap, and put up with the warts. Once a small language fills a niche, it is hard to take its place.

Well, then, could not “The Right Thing” be a small language, not hard to port but with no warts?

I guess it could be done, but I, for one, am not that smart (or have not had that much luck). But in fact I think it can not be done. Five and twenty years ago, when users did not want that much from a programming language, one could try. Scheme was my best shot at it.

But users will not now with glad cries glom on to a language that gives them no more than what Scheme or Pascal gave them. They need to paint bits, lines, and boxes on the screen in hues bright and wild; they need to talk to printers and servers through the net; they need to load code on the fly; they need their programs to work with other code they don’t trust; they need to run code in many threads and on many machines; they need to deal with text and sayings in all the world’s languages. A small programming language just won’t cut it.

So a small language can not do the job right and a large language takes too long to get off the ground. Are we doomed to use small languages with many warts because that is the sole kind of design that can make it in the world?

At one time this thought filled me with gloom. But then I saw a gap in my thinking. I said that users will not wait for “The Right Thing,” but will use what comes first and put up with the warts. *But*—users will not put up with the warts for all time. It is not long till they scream and moan and beg for changes. The small language will grow. The warts will be shaved off or patched.

If one person does all the work, then growth will be slow. But if one lets the users help do the work, growth can be quick. If many persons work side by side, and the best work is added with care and good taste, a great deal can be added in a short time.

APL was designed by one man, a smart man—and I love APL—but it had a flaw that I think has all but killed it: there was no way for a user to grow the language in a smooth way. In most languages, a user can define at least some new words to stand for other pieces of code that can then be called, in such a way that the new words look like primitives. In this way the user can build a larger language to meet his needs. But in APL, new words defined by the user do not look like language primitives at all. The name of a piece of user code is a word, but things that are built in are named by strange glyphs. To add what look like new primitives, to keep the feel of the language, takes a real hacker and a ton of work. This has stopped users from helping to grow the language. APL has grown some, but the real work has been done by just the few programmers that have the source code. If a user adds to APL, and what he added seems good to the hackers in charge of the language, they might then make it be built in, but code to use it would not look the same; the user would have to change his code to a new form, because in APL a use of what is built in does not look at all like a call to user code.

Lisp was designed by one man, a smart man, and it works in a way that I think he did not plan for. In Lisp, new words defined by the user look like primitives and, what is more, all primitives look like words defined by the user! In other words, if a user has good taste in defining new words, what comes out is a larger language that has no seams. The designer in charge can grow the language with close to no work on his part, just by

choosing with care from the work of many users. And Lisp grew much faster than APL did, because many users could try things out and put their best code out there for other users to use and to add to the language. Lisp is not used quite as much as it used to be, but parts of it live on in other languages, the best of which is called *garbage collection* (and I will not try to tell you now what that means in words of one syllable—I leave it to you as a task to try in your spare time).

This leads me to claim that, from now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.

I now think that I, as a language designer who helps out with the design of the Java programming language, need to ask not “Should the Java programming language grow?” but “*How* should the Java programming language grow?”

There is more than one kind of growth and more than one way to do it. But, as we shall see, if the goal is to be quick and yet to do good work, one mode may be better by far than all other modes.

There are two kinds of growth in a language. One can change the vocabulary, or one can change the rules that say what a string of words means.

A *library* is a vocabulary designed to be added to a programming language to make the vocabulary of the programming language larger. *Libraries* means more than one library. A true library does not change the rules of meaning for the language; it just adds new words. (You can see from this that, in my view, the code that lets you do a “long jump” in C is not a true library.)

Of course, there must be a way for a user to make libraries. But the key point is that the new words defined by a library should look just like primitives of the language. Some languages are like this and some are not. Those that are not are harder to grow with the help of users.

It may be good as well to have a way to add to the rules of meaning for a language. Some ways to do this work better than other ways. But the language should let work done by a user look just like what was designed at the start. I would like to grow the Java programming language in such a way that users can do more of this.

In the same way, there are two ways to do the growing. One way is for one person (or a small group) to be in charge and to take in, test, judge, and add the work done by other persons. The other way is to just put all the source code out there, once things start to work, and let each person do as he wills. To have a person in charge can slow things down, but to have no one in charge makes it harder to add up the work of many persons.

The way that is faster and better than all other ways does both. Put the source code out there and let all persons play with it. Have a person in charge who is a quick judge of good work and who will take it in and shove it back out fast. You don’t have to use what he ships, and you don’t have to give back your work, but he gives all persons a fast way to spread new code to those who want it.

The best example of this way to do things is *Linux*, which is an *operating system*, which is a program that keeps track of other programs in a computer and gives each its due in space and time.

You ought to read what Eric Raymond had to say of how Linux came to be. I shall tell you a bit of it once I define two more words for you.

A *cathedral* is a huge church. It may be made of stone; it may fill you with awe; but the key thought, in the mind of Eric Raymond, is that there is but one design, one grand plan, which may take a long time—many years—to make real. As the years pass, few

changes are made to the plan. Many, many persons are needed to build it, but there is just one designer.

A *bazaar* is a place with many small shops or stalls, where you can buy things from many persons who are there to sell their wares. The key thought here is that each one sells what he wants to sell and each one buys what he wants to buy. There is no one plan. Each seller or buyer may change his mind at whim.

Eric Raymond wrote a short work called “The Cathedral and the Bazaar” in which he looks at how programs are built or have been built in the past. (You can find it on his web site [11].) He talks of how he built a mail fetching program with the help of more than two hundred users. He quotes Fred Brooks as saying, “More users find more bugs,” and backs him up with tales from this example of building a program in the bazaar style. As for the role of the programmer in charge, Eric Raymond says that it is fine to come up with good thoughts, but much better to know them when you see them in the work of other persons. You can get a lot more done that way. Linux rose to its heights of fame and wide use in much the same way, though on a much larger scale. (To take this thought to the far end of the line: it may be that one could write an operating system by putting a million apes to work at a million typing machines, then just spotting the bits of good work that come out by chance and pasting them up to make a whole. That might take a long time, I guess. Too bad!)

But the key point of the bazaar is not that you can get many persons to work with you at a task, for cathedral builders had a great deal of help, too. Nor is the key point that you get help with the designing as well as with the building, though that in fact is a big win. No, the key point is that in the bazaar style of building a program or designing a language or what you will, the plan can change in real time to meet the needs of those who work on it and use it. This tends to make users stay with it as time goes by; they will take joy in working hard and helping out if they know that their wants and needs have some weight and their hard work can change the plan for the better.

Which brings me to the high point of my talk. It seems, in the last few years, at least, that if one is asked to speak on design, one ought to quote Christopher Alexander. I know a bit of his work, though not a lot, and I must say thanks to Dick Gabriel for pointing out to me a quote that has a lot to do with the main point of this talk.

I am sad to say that I do not know what this quote means, because Christopher Alexander tends to use many words of more than one syllable and he does not define them first. But I have learned to say these words by rote and it may be that you out there can glean some thoughts of use to you.

Christopher Alexander says [1]:

Master plans have two additional unhealthy characteristics. To begin with, the existence of a master plan alienates the users ... After all, the very existence of a master plan means, by definition, that the members of the community can have little impact on the future shape of their community, because most of the important decisions have already been made. In a sense, under a master plan people are living with a frozen future, able to affect only relatively trivial details. When people lose the sense of responsibility for the environment they live in, and realize that they are merely cogs in someone else’s machine, how can they feel any sense of identification with the community, or any sense of purpose there?

I think this means, in part, that it is good to give your users a chance to buy in and

to pitch in. It is good for them and it is good for you. (In point of fact, a number of cathedrals were built in the bazaar mode.)

Does this mean, then, that it is of no use to design? Not at all. But instead of designing a thing, you need to design a way of doing. And this way of doing must make some choices now but leave other choices to a later time.

Which brings me to the word I know you all want to hear: a *pattern* is a plan that has some number of parts and shows you how each part turns a face to the other parts, how each joins with the other parts or stands off, how each part does what it does and how the other parts aid it or drag it down, and how all the parts may be grasped as a whole and made to serve as one thing, for some higher goal or as part of a larger pattern. A pattern should give hints or clues as to when and where it is best put to use. What is more, some of the parts of a pattern may be holes, or slots, in which other things may be placed at a later time. A good pattern will say how changes can be made in the course of time. Thus some choices of the plan are built in as part of the pattern, and other choices wait till the time when the pattern is to be used. In this way a pattern stands for a design space in which you can choose, on the fly, your own path for growth and change.

It is good to design a thing, but it can be far better (and far harder) to design a pattern. Best of all is to know when to use a pattern.

Now for some more computer words.

A *datum* is a set of bits that has a meaning; *data* is the mass noun for a set of datums.

An *object* is a datum the meanings of whose parts are laid down by a set of language rules. In the Java programming language, these rules use types to make clear which parts of an object may cite other objects.

Objects may be grouped to form classes. Knowing the class of an object tells you most of what you need to know of how that object acts.

Objects may have fields; each field of an object can hold a datum. Which datum it holds may change from time to time. Each field may have a type, which tells you what data can be in that field at run time (and, what is more, it tells you what data can not be in that field at run time).

A *method* is a named piece of code that is part of an object. If you can name or cite an object, then you can call a method of that object; the method then does its thing.

The Java programming language has objects and classes and fields and methods and types. Now I shall speak of some things that the Java programming language does not yet have.

A *generic type* is a map from one or more types to a type. Put another way, a generic type is a pattern for building types. A number of groups of persons have set forth ways to add generic types to the Java programming language [2, 3, 8, 9, 14]; each way has its own good points.

An *operator* is a glyph, such as a plus sign, that can be used in a language as if it were a word. In C or the Java programming language, as in English, the sign first known as “and per se and” is an operator, but a full stop or quote mark is not an operator.

A word is said to be *overloaded* if it is made to mean two or more things and the hearer has to choose the meaning based on the rest of what is said. For example, by the rules defined near the start of this talk, a verb form such as “painted” might be a past tense or a past participle, and it is up to you, the hearer, to make the call as to which I mean when I say it. An other example is “design,” which I defined both as a noun and as a verb.

At some times, by some persons, an overloaded word is called *polymorphic*, which means that the word has many forms; but the truth is that the word has but one form,

and many meanings. The definition of a word may be polymorphic, but the word as such is not.

An operator can be overloaded in C plus plus, but right now operators in the Java programming language can not be overloaded by the programmer, though names of methods may be overloaded. I would like to change that.

I have said in the past, and will say now, that I think it would be a good thing for the Java programming language to add generic types and to let the user define overloaded operators. Just as a user can code methods that can be used in just the same way as methods that are built in, the user ought to have a way to define operators for user defined classes that can be used in just the same way as operators that are built in. What is more, I would add a kind of class that is of light weight, one whose objects can be cloned at will with no harm and so could be kept on a stack for speed and not just in the heap. Classes of this kind would be well suited for use as user defined number types but would have other uses, too. You can find a plan for all this on a web page by James Gosling [7]. (There are a few other things we could add as well, such as tail calls and ways to read and write those machine flags for numbers whose points float. But these are small language tweaks next to generic types and overloaded operators.)

If we grow the language in these few ways, then we will not need to grow it in a hundred other ways; the users can take on the rest of the task. To see why, think on these examples.

A *complex number* is a pair of numbers. There are rules for how to find the sum of two complex numbers, or a complex number times a complex number:

$$(a, b) + (c, d) = (a + c, b + d)$$

which says that a paired with b , plus c paired with d , is the same as a plus c paired with b plus d , and

$$(a, b) \cdot (c, d) = (a \cdot c - b \cdot d, a \cdot d + b \cdot c)$$

which says that a paired with b , times c paired with d , is the same as a times c less b times d paired with a times d plus b times c . Some programmers like to use complex numbers a lot; other programmers do not use them at all. So should we make “complex number” a type in the Java programming language? Some say yes, of course; other persons say no.

A *rational number* is a pair of numbers. There are rules (not the same as the rules for complex numbers, of course) for how to find the sum of two rational numbers, or a rational number times a rational number:

$$(a, b) + (c, d) = (a \cdot d + b \cdot c, b \cdot d)$$

which says that a paired with b , plus c paired with d , is the same as a times d plus b times c paired with b times d , and

$$(a, b) \cdot (c, d) = (a \cdot c, b \cdot d)$$

which says that a paired with b , times c paired with d , is the same as a times c paired with b times d . A few programmers like to use rational numbers a lot; most do not use them at all. So should we make “rational number” a type in the Java programming language?

An *interval* is a pair of numbers. There are rules (not the same as the rules for complex numbers or rational numbers, of course) for how to find the sum of two intervals, or an interval times an interval:

$$(a, b) + (c, d) = (a + c, b + d)$$

$$(a, b) \cdot (c, d) = (\min(a \cdot c, a \cdot d, b \cdot c, b \cdot d), \max(a \cdot c, a \cdot d, b \cdot c, b \cdot d))$$

A few programmers like to use them a lot and wish all the other programmers who use numbers would use them, too; but most do not use them at all. So should we make “interval” a type in the Java programming language?

John Horton Conway once defined a game to be a pair of sets of games (see his book *On Numbers and Games* [4]), then pointed out that some games may be thought of as numbers that say how many moves it will take to win the game. There are rules for how to find the sum of two games, and so on:

$$\begin{aligned} (A, B) + (C, D) &= \left(\{ a + (C, D) \mid a \in A \} \cup \{ (A, B) + c \mid c \in C \}, \right. \\ &\quad \left. \{ b + (C, D) \mid b \in B \} \cup \{ (A, B) + d \mid d \in D \} \right) \\ -(A, B) &= \left(\{ -b \mid b \in B \}, \{ -a \mid a \in A \} \right) \\ A - B &= A + (-B) \\ (A, B) \cdot (C, D) &= \left(\{ a \cdot (C, D) + (A, B) \cdot c - a \cdot c \mid a \in A, c \in C \} \cup \right. \\ &\quad \left. \{ b \cdot (C, D) + (A, B) \cdot d - b \cdot d \mid b \in B, d \in D \}, \right. \\ &\quad \left. \{ a \cdot (C, D) + (A, B) \cdot d - a \cdot d \mid a \in A, d \in D \} \cup \right. \\ &\quad \left. \{ b \cdot (C, D) + (A, B) \cdot c - b \cdot c \mid b \in B, c \in C \} \right) \end{aligned}$$

(I will not try to state here in words what these rules mean!) From this he worked out for hundreds of kinds of real games how to know which player will win. I think, oh, three persons in the world want to use this kind of number. Should we make it a type in the Java programming language?

A *vector* is a row of numbers all of the same type, with each place in the row named by the kind of number we first spoke of in this talk. There are rules ... In fact, for vectors of length three there are two ways to do “a vector times a vector,” so you can have twice the fun!

$$(a, b, c) + (d, e, f) = (a + d, b + e, c + f)$$

$$(a, b, c) \cdot (d, e, f) = a \cdot d + b \cdot e + c \cdot f$$

$$(a, b, c) \times (d, e, f) = (b \cdot f - c \cdot e, c \cdot d - a \cdot f, a \cdot e - b \cdot d)$$

Vectors of length three or four are a great aid in making bits on the screen look like scenes in the real world. So should we make “vector” a type in the Java programming language?

A *matrix* is a set of numbers laid out in a square. And there are rules (not shown here!). So should we make “matrix” a type in the Java programming language?

And so on, and so on, and so on.

I might say “yes” to *each* one of these, but it is clear that I *must* say “no” to *all of them*! And so would James Gosling and Bill Joy. To add all these types to the Java programming language would be just too much. Some parts of the programming vocabulary are fit for all programmers to use, but other parts are just for their own niches. It would not be fair to weigh down all programmers with the need to have or to learn all the words for all niche uses. We should not make the Java programming language a cathedral, but a plain bazaar might be too loose. What we need is more like a shopping mall, where there are not quite as many choices but most of the goods are well designed and sellers stand up and back what they sell. (I could speak at length on the ways in which a shopping mall is like a cathedral—but not here, not now!)

Now, the user could define objects for such numbers to have methods that act in the right ways, but code to use such numbers would look strange. Programmers used to adding numbers with plus signs would kvetch. (In fact, the Java programming language does have a class of “big numbers” that have methods for adding them and telling which is larger and all the rest, and you can not use the plus sign to add such numbers, and programmers who have to use them do kvetch.)

Generic types and overloaded operators would let a user code up all of these, and in such a way that they would look in all ways just like types that are built in. They would let users grow the Java programming language in a smooth and clean way. And it would not be just for numbers; generic types would be good for coding hash sets, for example, and there are many other uses.

And each user would not have to code up such number classes, each for his own use. When a language gives you the right tools, such classes can be coded by a few and then put up as libraries for other users to use, or not, as they choose; but they don’t have to be built in as part of the base language.

If you give a person a fish, he can eat for a day.

If you teach a person to fish, he can eat his whole life long.

If you give a person tools, he can make a fishing pole—and lots of other tools! He can build a machine to crank out fishing poles. In this way he can help other persons to catch fish.

Meta means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of an other pattern. A meta foo is a foo in whose slots you can put foos.

In a way, a language design of the old school is a pattern for programs. But now we need to “go meta.” We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind.

This is the nub of what I want to say. A language design can no longer be a thing. It must be a pattern—a pattern for growth—a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal.

My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language.

In the course of giving this talk, because I started with a small language, I have had to define fifty or more new words or phrases and sixteen names of persons or things; and I laid out six rules for making new words from old ones. In this way I added to the base language. If I were to write a book by starting from just the words of one syllable, I am

sure that I would have to define hundreds of new words. It should give no one pause to note that the writing of a program a million lines of code in length might need many, many hundreds of new words—that is to say, a new language built up on the base language. I will be so bold as to say that it can be done in no other way. Well—there may be one other way, which is to use a large, rich programming language that has grown in the course of tens or hundreds of years, that has all we need to say what we want to say, that we are taught as we grow up and take for granted. It may be that a hundred years from now there will be a programming language that by then has stood the test of time, needs no more changes for most uses, and is used by all persons who write programs because each child learns it in school. But that is not where we are now.

So. Language design is not at all the same kind of work it was thirty years ago, or twenty years ago. Back then, you could set out to design a whole language and then build it by your own self, or with a small team, because it was small and because what you would then do with it was small.

Now programs are big messes with many needs. A small language won't do the job. If you design a big language all at once and then try to build it all at once, you will fail. You will end up late and some other language that is small will take your place.

It would be great if there were some small programming language that felt large, the way Basic English is small but feels large in some ways. But I don't know how to do it and I have good cause to doubt that it can be done at all. In its day, APL was a small language that felt large; but our needs have grown and APL did not have a good pattern for growth.

So I think the sole way to win is to plan for growth with help from users. This is a win for you because you have help. This is a win for the users because they get to have their say and get to bend the growth to their needs. But you need to have one or more persons, too, or one or more groups, to take on the task of judging and testing and sifting what the users do and say, and adding what they think best to the big pile of code, in the hope that other users will trust what they say and not have to go to all the work to test and judge and sift each new claim or each new piece of code, each for his own self.

Parts of the language must be designed to help the task of growth. A good set of types, ways for a user to define new types, to add new words and new rules to the language, to define and use all sorts of patterns—all these are needed. The designer should not, for example, define twenty kinds of number types in the language. But there will be users who, all told, beg for twenty kinds of numbers. The language should have a way for the user to define number types that work well with each other, and with plus signs and other such signs, and with the many ways of pushing bits in and out of the computer. One might define just one or two number types at the start, to show how it ought to be done. Then leave the rest to the users. Help them all as best you can to work side by side (and not nose to nose).

You may find that you need to add warts as part of the design, so that you can get it out the door fast, with the goal of taking out the warts at a later time. Now, there are warts and then there are warts! With care, one can design a wart so that it will not be too hard to take out or patch up later on. But if you do not take care at the start, you may be stuck for years to come with a wart you did not plan.

Some warts are not bad things you put in, but good things you leave out. Have a plan to add those good things at a later time, if you should choose to do so, and make sure that other parts of your design don't cut you off from adding those good things when the time is right.

I hope to bring these thoughts to bear on the Java programming language. The Java programming language has done as well as it has up to now because it started small. It was not hard to learn and it was not hard to port. It has grown quite a bit since then. If the design of the Java programming language as it is now had been put forth three years ago, it would have failed—of that I am sure. Programmers would have cried, “Too big! Too much hair! I can’t deal with all that!” But in real life it has worked out fine because the users have grown with the language and learned it piece by piece, and they buy in to it because they have had some say in how to change the language.

And the Java programming language needs to grow yet some more—but, I hope, not a lot more. At least, I think only a few more rules are needed—the rest can be done with libraries, most of them built by users and not by Sun.

If we add hundreds of new things to the Java programming language, we will have a huge language, but it will take a long time to get there. But if we add just a few things—generic types, overloaded operators, and user defined types of light weight, for use as numbers and small vectors and such—that are designed to let users make and add things for their own use, I think we can go a long way, and much faster. We need to put tools for language growth in the hands of the users.

I hope that we can, in this way or some other way, design a programming language where we don’t seem to spend most of our time talking and writing in words of just one syllable.

One of the good things I can say for short words is that they make for short talks. With long words, this talk would run an hour and a half; but I have used less than an hour.

I would like to tell you what I have learned from the task of designing this talk. In choosing to give up the many long words that I have come to know since I was a child, words that have many fine shades of meaning, I made this task much harder than it needed to be. I hope that you have not found it too hard on your ears. But I found that sticking to this rule made me think. I had to take time to think through how to phrase each thought. And there was this choice for each new word: is it worth the work to define it, or should I just stick with the words I have? Should I do the work of defining a new word such as *mirror*, or should I just say “looking glass” each time I want to speak of one? (As an example, I was tempted more than once to state the “ly” rule for making new words that change what verbs mean, but in the end I chose to cast all such words to one side and make do. And I came that close to defining the word *without*, but each time, for better or for worse, I found some other way to phrase my thought.)

I learned in my youth, from the books of such great teachers of writing as Strunk and White [13], that it is better to choose short words when I can. I should not choose long, hard words just to make other persons think that I know a lot. I should try to make my thoughts clear; if they are clear and right, then other persons can judge my work as it ought to be judged.

From the work of planning this talk, in which I have tried to go with this rule much more far than in the past, I found that for the most part they were right. Short words work well, if I choose them well.

Thus I think that programming languages need to be more like the languages we speak—but it might be good, too, if we were to use the languages we speak more in the way that we now use programming languages.

All in all, I think it might be a good thing if those who rule our lives—those in high places who do the work of state, those who judge what we do, and most of all those who

make the laws—were made to define their terms and to say all else that they say in words of one syllable. For I have found that this mode of speech makes it hard to hedge. It takes work, and great care, and some skill, to find just the right way to say what you want to say, but in the end you seem to have no choice but to talk straight. If you do not veer wide of the truth, you are forced to hit it dead on.

I urge you, too, to give it a try.

References

- [1] Christopher Alexander, et al. *The Oregon Experiment*. Oxford University Press, 1988.
- [2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 183–200, October 1998.
- [3] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 201–215, October 1998.
- [4] J. H. Conway. *On Numbers and Games*. Academic Press, 1976.
- [5] Richard P. Gabriel. Lisp: Good news, bad news, how to win big. *AI Expert* 6, 6, pages 30–39, June 1991.
- [6] Richard P. Gabriel. Lisp: Good News, Bad News, How to Win Big. Copy of [5]. URL (correct as of August 2001) <http://www.ai.mit.edu/docs/articles/good-news/good-news.html>
- [7] James A. Gosling. The Evolution of Numerical Computing in Java. Undated. URL (correct as of August 2001) <http://www.javasoft.com/people/jag/FP.html>
- [8] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.
- [9] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.
- [10] Charles Kay Ogden. *Basic English: A General Introduction with Rules and Grammar*. Paul Treber and Co., Ltd., London, 1930.
- [11] Eric S. Raymond. The Cathedral and the Bazaar. Dated November 22, 1998. URL (correct as of August 2001) <http://www.tuxedo.org/~esr/writings/cathedral-bazaar>
- [12] Guy L. Steele Jr. *Growing a Language*. Videotape (54 minutes) of a talk at the *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1998. University Video Communications, 1998.
- [13] William Strunk Jr. and E. B. White. *The Elements of Style*. Third edition. Allyn and Bacon, 1979.
- [14] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of European Conference on Object-Oriented Programming*, pages 444–471, LNCS 1241, Springer-Verlag, 1997.

make the laws were made to define their terms and to say all else that they say in words of one syllable. or I have found that this mode of speech makes it hard to hedge. It takes work, and great care, and some skill, to find just the right way to say what you want to say, but in the end you seem to have no choice but to talk straight. If you do not veer wide of the truth, you are forced to hit it dead on.

I urge you, too, to give it a try.

References

- [1] Christopher Alexander, et al. *The Oregon Experiment*. Oxford University Press, 1988.
- [2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 183–200, October 1998.
- [3] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 201–215, October 1998.
- [4] J. H. Conway. *On Numbers and Games*. Academic Press, 1976.
- [5] Richard P. Gabriel. Lisp: Good news, bad news, how to win big. *AI Expert* 6, 6, pages 30–39, June 1991.
- [6] Richard P. Gabriel. Lisp: Good News, Bad News, How to Win Big. Copy of [5]. URL (correct as of August 2001) <http://www.ai.mit.edu/docs/articles/good-news/good-news.html>
- [7] James A. Gosling. The Evolution of Numerical Computing in Java. Undated. URL (correct as of August 2001) <http://www.javasoft.com/people/jag/FP.html>
- [8] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.
- [9] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.
- [10] Charles Kay Ogden. *Basic English: A General Introduction with Rules and Grammar*. Paul Treber and Co., Ltd., London, 1930.
- [11] Eric S. Raymond. The Cathedral and the Bazaar. Dated November 22, 1998. URL (correct as of August 2001) <http://www.tuxedo.org/~esr/writings/cathedral-bazaar>
- [12] Guy L. Steele Jr. *Growing a Language*. Videotape (54 minutes) of a talk at the *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1998. University Video Communications, 1998.
- [13] William Strunk Jr. and E. B. White. *The Elements of Style*. Third edition. Allyn and Bacon, 1979.
- [14] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of European Conference on Object-Oriented Programming*, pages 444–471, LNCS 1241, Springer-Verlag, 1997.

The GC Interface in the EVM

Derek White and Alex Garthwaite

Introduction by Steve Heller

This version of "The GC Interface in the EVM" was shortened by Alex Garthwaite, and includes an updated "Addenda, Open Issues, and Future Directions" section. Visit www.sun.com/research/jtech/ for the full version.

The Java™ Technology Research Group (originally known as Java Topics) was formed late in 1995 to develop expertise in Java garbage collection (GC). The founding members of the group, Guy Steele, Derek White and Steve Heller, were joined by Ole Agesen during the summer of 1996, and we set the goal to explore soft real-time GC, probabilistically meeting real time constraints. We wanted to "solve the Java GC problem," but we soon realized that Java didn't *have* a GC problem. Java implementations at the time were so slow that GC was not the weakest link. We set out to build a high-performance Java™ Virtual Machine (JVM) that would have a GC problem so that we could solve it. And we carefully designed this JVM to facilitate GC experimentation.

Soon after Christine Flood and Dave Detlefs joined the team in the fall of 1996, Ole Agesen led an effort to overhaul the Sun JVM™ now known as the "Classic VM" to create a JVM known as "ExactVM" (also known as "EVM," and eventually known as "The Sun Microsystems Laboratories Virtual Machine for Research," or "ResearchVM"). At the center of the ExactVM was a GC interface, designed to facilitate GC experimentation. Alex Garthwaite, who joined the team as a summer intern in 1997 and 1998 and a full time member in the fall of 1998, teamed up with Derek White to produce the GC interface document (this TR).

We also joined forces with a product team whose goal was to build a high-performance JVM for the Solaris™ Operating Environment. The GC interface abstraction did not cost the VM performance, and ExactVM became Sun's Java™ 2 SDK Production Release for Solaris OE, versions 1.2.2_xx, managed by David Nelson-Gal, Borek Vokach-Brodsky, Dave Seberger (and many others), led technically by Ole Agesen and Ross Knippel (Mario Wolczko was the initial product team technical lead), and contributed to by scores of others. Three years later, the ExactVM is still packaged with Solaris and shipped as a product.

The ExactVM team was a close and highly successful collaboration between research and development; together we investigated various aspects of Java performance and scalability, notably synchronization, just-in-time compilation, and garbage collection. This project was a success disaster as the research team was drawn tightly into the product team. David Nelson-Gal, the director of the product side of the team, extracted a written commitment (dated 8/26/97) from the research side of the team to remain committed to the project for at least one year. We did that and more.

The GC interface supported various generational garbage collection configurations, the default system being a semi-spaces young generation and a mark-compact mature generation. We also developed many experimental GC configurations. As is often the case, we leveraged the momentum and energy of summer interns: during the summer of 1998 Alex Garthwaite and Tony Printezis, respectively, designed and implemented incremental (a bit at a time) and concurrent (simultaneous with the Java program) collectors. The next summer we also explored parallel (multi-threaded, led by Christine Flood, with help from summer intern Catherine Zhang) GC. None of these technologies were made widely available to customers in 1.2.2_xx.

Within Sun, at least one other product group and a host of other research teams leveraged the GC interface and the ExactVM to experiment or to build other JVMs. On the product side, Borek Vokach-Brodsky and his team built the J2ME™ Connected Device Configuration which contains the CVM, a small footprint JVM based on ExactVM and the GC interface. The following efforts were on the research side. Mario Wolczko derived the "Tracing JVM" for extracting and studying Java traces, and it was licensed to nine universities and several companies. Mick Jordan, Malcolm Atkinson (Glasgow University), and their teams derived a JVM that implemented orthogonal persistence for the Java platform by leveraging the GC interface; this JVM was also licensed to a number of universities. In a related project Laurent Daynes used the ExactVM extensively to experiment with transaction-based isolation between applications. Grzegorz Czajkowski and Laurent used the ExactVM to evaluate the performance of a multi-tasking JVM prototype based on bytecode editing and to test the native method isolation mechanism.

ExactVM source was licensed to a number of academic institutions to facilitate research, especially in garbage collection: UC Santa Barbara, Glasgow University, Purdue, Tokyo Institute of Technology, Pennsylvania State University, Rice University, University of Kent, University of Waterloo, University of Texas at Austin, Technion, Princeton, University of Cambridge, San Francisco State University, and the University of Toronto. The success of this program was due in large part to the GC interface document, which was the only external documentation delivered with the ExactVM source. The success can be measured by the many papers and theses enabled by the program, some of which are listed below and some of which are yet to emerge.

A second version of the GC Interface was designed by Dave Detlefs for use in the Java HotSpot™ JVM and is part of Sun's 1.4 release. In subsequent releases, we expect that many of the incremental, concurrent, and parallel technologies we developed will be deployed in products. This represents another close and successful collaboration between research and development, led technically by Dave Detlefs on the research side, and Peter Kessler on the product side.

The current research of the team continues to build on the incremental, concurrent, and parallel techniques developed in conjunction with the GC interface and ExactVM. We walk the hairy edge between research and development.

While I've attempted to retell the story of our GC interface and the ExactVM (they are inseparable), it was such a broad team effort that I undoubtedly have not given adequate credit in some cases. We worked with many researchers and product people here at Sun. We worked with interns, visiting professors, and academic collaborators. And, although I will make him uncomfortable by concluding thus, the undisputed champion and leader of our effort was Ole Agesen, to whom we owe a great debt. [note to self: send Ole a certificate of appreciation (use color printer) and a couple of pounds of "Peet's"]

RELATED GROUP PUBLICATIONS (see <http://www.sun.com/research/jtech/>):

Ole Agesen. 1998 "GC Points in a Threaded Environment." Sun Labs TR 98-70. Mountain View, CA

Ole Agesen. 1999 "Space and Time-Efficient Hashing of Garbage-Collected Objects." Theory and Practice of Object Systems, 5(2) (June): 119-124.

Ole Agesen and David Detlefs. 1997. "Finding References in Java(TM) Stacks." OOPSLA Workshop on Garbage Collection and Memory Management. Atlanta, GA

Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, Derek White. 1999. "An Efficient Meta-lock for Implementing Ubiquitous Synchronization." Sun Labs TR 99-76. Mountain View, CA. Presented at OOPSLA '99.

Ole Agesen, David Detlefs, and J. Eliot B. Moss. 1998. "Garbage Collection and Local Variable Type-Precision and Liveness in Java(TM) Virtual Machines." ACM SIGPLAN Conference on Programming Language Design and Implementation (June):269-279. Montreal, Canada

Ole Agesen and Alex Garthwaite. 2000. "Efficient Object Sampling Via Weak References." ISMM2000. Minneapolis, MN

Dave Detlefs and Tony Printezis. 2000. "A Generational Mostly-Concurrent Garbage Collector" Sun Labs TR-2000-88. Mountain View, CA. Short version appeared in ISMM2000.

Christine Flood, Dave Detlefs, Nir Shavit, Catherine Zhang. 2001. "Parallel Garbage Collection for Shared Memory Multiprocessors." Java Virtual Machine Research and Technology Symposium. Monterey, CA

Timothy L. Harris. 2000. "Dynamic Adaptive Pre-Tenuring." ISMM2000. Minneapolis, MN

Derek White and Alex Garthwaite. 1998. "The GC Interface in the EVM." Sun Labs TR 98-67. Mountain View, CA

RELATED OUTSIDE PUBLICATIONS (GENERALLY ENABLED BY THE SOURCE LICENSING PROGRAM)

M. P. Atkinson, M. Dmitriev, C. G. Hamilton, T. Printezis. 2000. "Scalable and Recoverable Implementation of Object Evolution for the PJama_1 Platform," Proceedings of The Ninth International Workshop on Persistent Object Systems (POS9). Lillehammer, Norway.

M. P. Atkinson and M. J. Jordan 2000. "A Review of the Rationale and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform," Sun Labs TR-2000-90.

Kumar Brahnmath. May 1998. "Optimizing Orthogonal Persistence for Java," Master's Thesis, Purdue University.

Kumar Brahnmath, Nathaniel Nystrom, Antony L. Hosking, Quintin Cutts. 1998, 1999. "Swizzle barrier optimizations for orthogonal persistence in Java," In Proceedings of the Third International Workshop on Persistence and Java (Tiburon, California, September 1998). Published as Advances in Persistent Object Systems, Morrison, Jordan and Atkinson (Eds.), pp 268-278. Morgan Kaufmann, 1999

Quintin Cutts and Antony L. Hosking. 1997. "Analysing, profiling and optimising orthogonal persistence for Java," In Proceedings of the Second International Workshop on Persistence and Java (Half Moon Bay, California, August 1997). Sun Microsystems Laboratories Technical Report 97-63, pp 107-115.

Quintin Cutts, Steve Lennon and Antony Hosking, 1998, 1999. "Reconciling buffer management with persistence optimisations," In Eighth International Workshop on Persistent Object Systems" (Tiburon, California, August 1998). Published as Advances in Persistent Object Systems, Morrison, Jordan and Atkinson (Eds.), pp 51-63. Morgan Kaufmann, 1999.

Sylvia Dieckmann and Urs Hoelzle. 1999. "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks." In Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, Springer Verlag.

Sylvia Dieckmann and Urs Hoelzle. 1999. A Case for Using Active Memory to Support Garbage Collection. Workshop on Hardware Support for Objects and Microarchitectures in Java. Held in Conjunction with ICCD'99, Austin, TX

Antony Hosking, Nathaniel Nystrom, Quintin Cutts and Kumar Brahnmath. 1998, 1999. "Optimizing the read and write barrier for orthogonal persistence," In Eighth International Workshop on Persistent Object Systems (Tiburon, California, August 1998). Published as Advances in Persistent Object Systems, Morrison, Jordan and Atkinson (Eds.), pp 149–159. Morgan Kaufmann, 1999.

Antony L. Hosking, Nathaniel Nystrom, David Whitlock, Quintin Cutts and Amer Diwan. 2001. "Partial redundancy elimination for access path expressions." Software – Practice and Experience 31(6):577–600, 2001.

E. Hunt, M. P. Atkinson, and R. W. Irving. 2001. "A Database Index To Large Biological Sequences," Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001). Morgan Kaufmann, Roma, Italy.

Nathaniel Nystrom. 1998. "Bytecode Level Analysis and Optimization of Java Classes," Master's Thesis, Purdue University.

T. Printezis. 2000 . "Management of Long-Running High-Performance Persistent Object Stores," PhD Thesis. Department of Computing Science, University of Glasgow, Scotland.

Tony Printezis. 2001. "Hot-Swapping Between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment." Java Virtual Machine Research and Technology Symposium. Monterey, CA. Also: TR-2001-78, Department of Computing Science, University of Glasgow, Scotland.

T. Printezis and M. P. Atkinson. 2001. "An Efficient Object Promotion Algorithm for Persistent Object Systems," SPE 31(10) 941--981.

Darko Stefanovic. 1999. "Properties of Age-Based Automatic Memory Reclamation Algorithms" Doctoral Dissertation, University of Massachusetts Amherst.

Darko Stefanovic, J. Eliot B. Moss, and Kathryn S. McKinley. 1999. "Age-Based Garbage Collection," Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Denver, Colorado.

David Whitlock and Antony L. Hosking. 2000. "A Framework for Persistence-Enabled Optimization of Java Object Stores," In Ninth International Workshop on Persistent Object Systems." Lillehammer, Norway.

David Whitlock. 2000. "Persistence-enabled optimization of Java programs," Master's Thesis, Purdue University.

The GC Interface in the EVM¹

Derek White and Alex Garthwaite

SML TR-98-67

December 1998

Abstract:

This document describes how to write a garbage collector (GC) for the EVM. It assumes that the reader has a good understanding of garbage collection issues and some familiarity with the Java[™] language.

The EVM is part of a research project at Sun Labs. The interfaces described in this document are under development and are guaranteed to change. In fact, the purpose of this document is to solicit feedback to improve the interfaces described herein. As a result, specific product plans should not be based on this document; everything is expected to change.

¹EVM, the Java virtual machine known previously as ExactVM, is embedded in Sun's Java 2 SDK *Production* Release for Solaris[™], available at <http://www.sun.com/solaris/java/>.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
derek.white@east.sun.com

The GC Interface in the EVM¹

Derek White and Alex Garthwaite

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, California 94303

1. Introduction

This document describes how to write a garbage collector (GC) for the EVM. It assumes that the reader has a good understanding of garbage collection issues and some familiarity with the JavaTM programming language.

The EVM is part of a research project at Sun Labs. The interfaces described in this document are under development and are guaranteed to change. In fact, the purpose of this document is to solicit feedback to improve the interfaces described herein. As a result, specific product plans should not be based on this document; everything is expected to change.

This section defines some terms used in the document, describes how the EVM is organized with respect to memory management, and describes how the Java programming language and the Java virtual machine (JVM) implementation affect the design of collectors.

A Java virtual machine executes abstract instructions in one or more threads of control. There may be other threads running in the system, but they are not relevant to this document. A JVM may execute bytecodes by interpreting them directly, or by first compiling the bytecodes into a native form and executing the native code (“compiled code”). Both forms of the code executed are known as “Java code.” Each thread has two stacks: a “Java stack” to run bytecodes and a “native stack” to run native code. The JVM contains explicit support for objects, which are dynamically allocated out of a garbage collected heap (“Java heap”). The JVM also supports manipulating variables associated with classes (“Java static variables”).

The EVM is a Java runtime environment (JRE) optimized for Solaris and server applications. Particular attention has been paid to scalability and reliability issues, including garbage collection. The EVM has an interface that makes it simple to implement and to use different garbage collectors. A collector for the EVM must be chosen at build time, to allow the collector to supply macros and inline functions that the rest of the system can use.

The GC interface has been used to implement mark and sweep, copying, compacting, generational, and concurrent garbage collectors. The interface will not handle simple reference counting collectors due to limited information about Java stacks, but will support deferred reference counting collectors. Work still needs to be done to support weak references with concurrent collectors in a GC independent way.

1. EVM, the Java virtual machine known previously as ExactVM, is embedded in Sun’s Java 2 SDK *Production* Release for SolarisTM, available at <http://www.sun.com/solaris/java/>

1.1 Organization of the EVM

A Java runtime environment consists of class files, shared libraries containing the native methods for the class files, and a JVM shared library proper. A JRE is the smallest set of executables and files that constitute the standard Java platform.

A JVM can load and execute class files, and native methods can call JVM code via the Java native interface (JNI) [2] or the private JVM interface. Native methods have no direct access to the Java heap.

For the purpose of describing the memory system and the collector interface, we can further divide the EVM into normal Java execution (interpreter, JIT, class loading, JNI/JVM interfaces, etc.), and the memory system and collector:

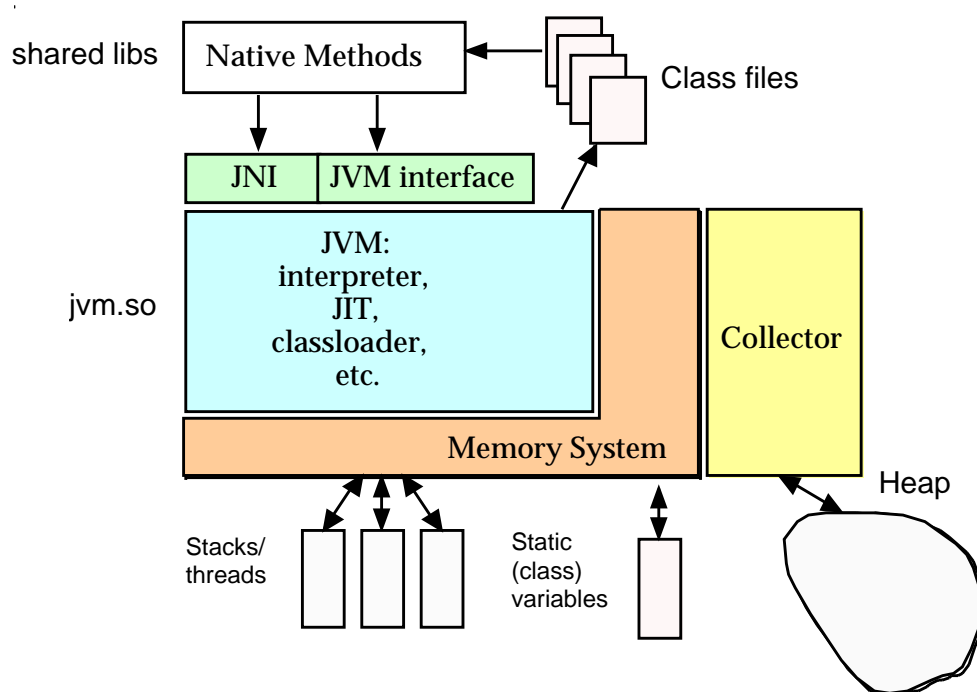


FIGURE 1. JVM Organization

The memory system sits between the collector and the rest of the JVM. It provides an internal interface to the JVM for object allocation and garbage collection (`gc.h`), as well as access to objects in the heap (`memsys.h`). The memory system also has a private interface (`gc/gcimpl.h`) that supports all collectors. The private interface provides routines to iterate over all “roots” in the JVM, suspend and resume Java threads, and handle most details related to locking, class unloading, and weak references.

A collector has to implement only the details of a specific GC algorithm. It does this by defining the `specificXXX` functions in the abstract collector interface, and optionally defining read/write barriers. The barriers ensure that no code can access or modify the heap without the collector knowing about it. The abstract collector interface is private to the memory system. For the most part, the collector has access to the rest of the JVM only through the memory system. Object layout and class information (defined in JVM headers) are notable exceptions to this.

1.2 Java Language Issues

The Java language specifies some language features that impact memory management.

1.2.1 Object Issues

The Java programming language is mostly object-oriented. Values in the language can be references to objects or they can be primitive values (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, or `double`). The language specifies the precision of these values explicitly (often 32-bit or 64-bit signed numbers), which precludes using tagging on most hardware. To deduce which values are references and which are primitives, the JVM uses the fact that the Java language is strongly typed and that objects carry runtime type information.

1.2.2 Finalization

A class may define a `finalize()` method. If an object has a `finalize()` method, the method will be called sometime after the collector has determined that the object is unreachable, but before the object is collected. See section 12.6 in the Java Language Specification (JLS) [1] and section 2.16.7 in the Java Virtual Machine Specification (JVM spec) [5] for more details. In JVMs based on the Java Platform 1.2 API [3], finalization is implemented using weak references.

1.2.3 Weak References

New to Java Platform 1.2 are public weak reference classes defined in the `java.lang.ref` package. This section gives an overview of the language feature. We will provide more information and examples later in section 2.2.3.3. The basic idea is that each weak reference object has a special field called the `referent`, which is a weak pointer to some object. Weak reference objects have another field that refers to the queue that the weak reference should be placed on for further processing. When the memory system notices that there are no stronger pointers to a weak reference object's `referent`, it places the weak reference object onto its queue. Various Java threads are waiting on the queues for objects to process. There are built-in threads to handle finalization, but users may define their own queue and thread to handle other types of weak reference objects.

There are several predefined references classes: `Reference` (the abstract superclass of the following classes), `SoftReference`, `WeakReference`, and `PhantomReference`. Note that this document uses the term “weak reference” to describe all subclasses of `java.lang.ref.Reference`. The term `WeakReference` refers to the specific class `java.lang.ref.WeakReference`. There is also a private subclass of `Reference` called `FinalReference` which is used to implement finalization and there are JNI weak references for use within native code.

Note that weak references in other languages usually require the collector to break the weak link to the dying referent. But weak references in the Java language often preserve the weak link. The `referent` of a `PhantomReference` or a `FinalReference` is not cleared until Java code calls the `clear()` method on it. This means that a referent may still be reachable after processing if the weak reference object stays reachable, or if processing the weak reference object stores the referent somewhere reachable. The built-in processing code for `FinalReferences` calls the `clear()` method and drops all references to the processed `FinalReference`. The `referent` fields of `SoftReferences` and `WeakReferences` are cleared by the memory system before enqueueing the weak reference object.

Some weak reference objects are stronger than others. This means that if there are both “stronger” weak reference objects and “weaker” weak reference objects that share the same dying referent, a given strength of weak reference objects will be processed in a particular collection only if after the stronger references have all been cleared. All weak reference objects of the same strength that share the same dying referent will be queued for processing in the same GC. From strongest to weakest, weak reference objects are ordered as follows:

1. `SoftReference` and subclasses. These weak reference objects are not processed as aggressively as others, and are used to create caches that are cleared as a last resort. The referent fields of these weak reference objects are cleared before the reference objects are enqueued for processing.
2. `WeakReference` and subclasses. The referent fields of these weak reference objects are cleared before the reference objects are enqueued for processing.
3. `FinalReference`. These weak reference objects are enqueued in a finalization queue. Another Java thread will call the `finalize()` method on the referent of each `FinalReference` object in the queue.
4. `PhantomReference` and subclasses. These weak reference objects do not have access to the referent, but since they are processed last, they can be used to trigger “postmortem finalization” actions. Because users don't have access to the `PhantomReference`'s referent field, users usually create subclasses of `PhantomReference` that contain interesting state.
5. JNI weak references. These weak reference objects do have access to the referent. Like `WeakReference` objects, the referent is cleared as soon as the reference object is found to have no stronger references. There are no queues associated with JNI weak references.

The class `Reference` is both abstract and has package-private constructors, so there can be no weak reference objects in the system other than those described above.

It may take a while for a dying referent to be collected: it will only be collected in the GC after the weakest reference object that refers to the dying referent has been enqueued and processed by its queue, and only if the Java code processing the queue has not stored the pointer to the dying referent somewhere. See Java Software's documentation [3] [6] and “Collecting Weak References,” page 11, in the Cookbook section for more information.

1.2.4 Class Unloading

A class or interface may be unloaded if and only if its class loader is unreachable (the definition of unreachable is given in section 12.6 in the JLS [1]). Classes loaded by the default system loader may not be unloaded. This rule implies that a JVM cannot unload a class if any instances (direct or indirect) of the class are reachable or if the object representing the class is reachable. In the JVM, class unloading may occur incrementally or as a result of a full GC. Generational collectors in the JVM need to treat classes as if they are part of the oldest generation. See section 12.8 in the JLS [1] and section 2.16.8 in the JVM spec [5] for more details.

1.3 Implementation Issues

A collector needs to be designed with knowledge of certain implementation details of the JVM and its memory system. The memory system takes care of many issues that are common to all garbage collectors, but it requires some cooperation.

1.3.1 Object Layout

All objects begin with a header. The header contains a pointer to class information and a word that holds the object's hash code, lock fields, and bits that can hold age information (used by collectors). The rest of an object contains zero or more 32-bit words to hold object fields, or a word holding the array length followed by array elements. `long` and `double` object fields and array elements may or may not double word aligned. Alignment is controlled by a compile-time flag.

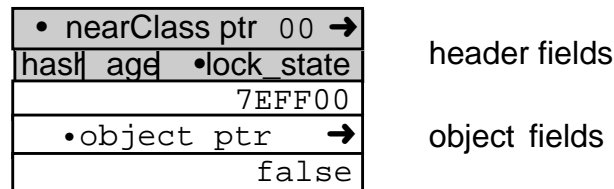


FIGURE 2. An Example of a 3-field Object's Layout

An object's class information is stored in a `NearClass` structure. A `NearClass` stores the basic information needed to call a virtual method or perform GC. This includes the object's method table, size, and “instance layout map.” The `NearClass` also contains a pointer to a more complete representation of a class called the `Class` structure. Neither the `NearClass` nor `Class` structures are objects. Note that an object does not have an ordinary reference to its `java.lang.Class` instance, so simply scanning an object's reference fields will not scan its class (an additional operation is required).

All objects are at least two words long, and are minimally aligned at a word boundary (4-byte alignment). This means that a collector can count on the low-order bits (at least two) of an object pointer being zero, and can set those bits during garbage collection. A collector can use the `OBJECT_ALIGNMENT_MASK` to get access to those bits. Similarly, the `NearClass` is aligned, so a collector can use low-order bits of the `nearClass` pointer field in an object's header to record things like mark bits or indirection bits during collection. A collector can use the `HEADER_ALIGNMENT_MASK` to get access to those bits.

1.3.2 Roots and Memory Consistency

The collector needs to know the locations of all the pointers to objects in the system: in Java stacks, Java static variables, JVM internal global and local roots, etc. But recording where all pointers are in the system at every instant would take too much time and space. So the JVM arranges execution such that each thread will know where its pointers are at frequent intervals. If a thread is at a point where all pointers in the thread's stack are known, it is called “consistent.” Otherwise, it is called “inconsistent.” A thread's stack is separated into a native stack provided by the OS where VM code, compiled methods, and native methods run, and a Java stack managed by the JVM where interpreted methods run.

When some part of the JVM decides to do a garbage collection, the memory system suspends all threads. Then it resumes all inconsistent threads after arranging it so that each thread will suspend itself when it becomes consistent. When all of the threads are stopped consistently, the memory system calls the collector to do actual collection. Inconsistent threads suspended themselves by either polling a variable and suspending explicitly when the thread becomes consistent, or by the memory system inserting trap instructions at the points where threads will become consistent.

The JVM code itself (such as class loading) normally is consistent when it runs. Most JVM code accesses objects using the low level native interface (LLNI). The LLNI is a layer over the memory system that allows the JVM to refer to objects using LLNI handles (pointers to “local roots” which point to an object), manages the consistency of threads, and provides many accessor operations. When an LLNI operation needs to dereference a handle, it becomes inconsistent temporarily. The JVM code has to register all local roots so the memory system will know where to find direct pointers to objects.

Java code (either running in the interpreter loop or as compiled code) normally is inconsistent when it is executed, so it can use pointers to objects without informing the memory system. But Java code must become consistent at frequent intervals or threads that need to allocate objects may be arbitrarily delayed. So the interpreter and JIT compiler generate “stack maps” for every method call and backward branch, which describe where the pointers are for a particular method and program counter (pc).

Native methods almost always run while consistent. The only JNI function that makes a thread inconsistent is `GetPrimitiveArrayCritical`. The only JNI functions that are callable while a thread is inconsistent are more calls to `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical`.

1.3.3 Locking

In order to support multiple threads, the memory system uses a “heap lock” around any allocation from the global heap and around GC itself. Local allocation buffers (LABs), described in “Allocation,” page 9, reduce contention on this lock. The memory system also depends on various JVM data structures that are protected by locks, such as the table of all classes and the thread queue. So the memory system takes care of acquiring and releasing these locks around garbage collection.

If a collector needs other locks around collection, the collector can create and register them with the memory system. The memory system takes care of acquiring and releasing these “GC locks” in the correct order.

2. Collector Cookbook

A collector must support five core routines, and several supporting routines. In addition, it may supply read and write barriers that the rest of the JVM must use. This section describes the basic header files and types that a collector deals with, and the routines and barriers that a collector should support. It also describes how the collector must use major routines that the memory system provides.

2.1 Preliminary Information

Before we look at what a collector should provide, it is useful to explain where the definitions and interfaces can be found and what the types are that are used in these interfaces.

Each macro or function definition described in the Cookbook and Reference sections falls into one of four categories.

- *[must define]*. The collector must define the macro or function.
- *[may define]*. The collector may define the macro or function to enable optional or non-

default behavior.

- *[must use]*. The collector must use the macro or call the function.
- *[may use]*. The collector may use the macro or call the function as needed.

All definitions are in the *[may use]* category unless otherwise noted.

2.1.1 Code Organization

Any headers files that are part of a collector should be placed in a subdirectory of `src/share/javavm/include/gc` named for the collector that uses them. This directory name should also be used for the source directory for the collector under `src/share/javavm/runtime/gc`. For example, for the ReallySimpleHeap collector described below, the collector's header files and source files would be placed in the following two directories:

```
src/share/javavm/include/gc/reallysimple
src/share/javavm/runtime/gc/reallysimple
```

The organization and build process for the EVM as a whole is beyond the scope of this document.

2.1.2 Definitions

The definitions used by a collector come from several header files. To simplify the development of collectors, these header files have been grouped together in a common `gc/specific_gc.h` header file. Collectors should include this header file along with any collector-specific header files.

Note that the names of all the routines that the collector must define start with “`specific`,” and the names of all the barrier macros end with “`_BARRIER`.”

2.1.3 Types

There are a large number of types defined by the whole JVM. Fortunately, there are only a handful that a collector needs to use.

2.1.3.1 Primitive Types

The following primitive (non-object) types are commonly used in the interfaces for the collector:

- `bool_t` - a simple boolean type with two values: `TRUE` or `FALSE` (1 or 0).
- `java_int` - the integral type for Java language integers (always 32-bit).
- `uintptr_t` - unsigned integers big enough to hold pointer values.

2.1.3.2 Object Types

The primary type used for representing objects is: `java_lang_Object`. It defines the basic structure of the start of any object in the heap:

```
struct {
    ObjHeader h;
    word32 fields[1]; /* actually zero or more fields */
}
```

This structure is described in “Object Layout,” page 5 in the Introduction section. There are three pointer types defined for each class that the JVM needs intimate knowledge of, but collectors typically pass references to objects either as `java_lang_Object*` or `Ijava_lang_Object`.

- `java_lang_Object*` is a direct pointer to an object, and should only be used by the memory system, the collector, or in inconsistent regions of the JVM.
- `DEREFERENCED_Ijava_lang_Object` is a volatile direct pointer to an object, and is usually only used to construct the following type:
- `Ijava_lang_Object` is a handle to an object (a pointer to a `DEREFERENCED_Ijava_lang_Object`). This type is used in the “mutator” portions of the JVM to protect the JVM from the GC moving objects (otherwise the C compiler might create a cached temporary pointer somewhere).
- `word32` - a union of one-word primitive types and `DEREFERENCED_Ijava_lang_Object`. In general, a collector uses this as an opaque type and should not examine the contents.

In addition, macros are provided for accessing the fields of an object and for querying information about its state. Collectors also need access to class information stored in a `NearClass`.

2.1.3.3 Function Types

In order to support the collection of information about objects in a heap, there are functions and macros that allow a collector to iterate over the objects in the heap applying a callback function to each object in turn. The type of these callback functions is: `ObjIterator`. Its definition is:

```
void (*ObjIterator)(java_lang_Object *obj, void *data)
```

where the `data` field is used to store the inputs, current state, and results of the process of applying the callback function to all the objects in question.

```
void (*RefIterator)(java_lang_Object **objp, void *data)
```

Also available as `RootCallback`, the `RefIterator` type is like `ObjIterator`, but it accepts pointers to a location that points to an object allowing the callback to update pointers to objects.

Another callback type can be used for any function that accepts a `void*` and returns `void`:

```
void (*SimpleFct)(void *data)
```

2.2 Core Collector Routines

There are 16 routines that a collector must implement, but only five that must interact with the memory system in complex ways. The collector must provide support for initialization, allocation, collection, expansion, and iteration. These functions are the core of the collector.

2.2.1 Initialization

Early in the initialization of the VM (before loading any classes), the memory system will call:

```
[must define]
uintptr_t specificInitHeap(uintptr_t minWords, uintptr_t maxWords)
```

The collector is responsible for doing whatever initialization it needs, and allocating a heap large enough to hold at least `minWords`. The collector must not expand the heap larger than `maxWords` of object space. `minWords` will be `maxWords`. The sizes refer to the number of words available for objects—it doesn't limit the overhead that the collector might also need (for semi-spaces, remembered sets, mark bits, etc.). `specificInitHeap()` must return the number of words available for objects or zero if initialization failed. A collector will often use the `MemoryArea` utilities to allocate address space for the heap.

If a collector needs its own locks, it should initialize the locks here, and register the locks with the memory system. This allows the memory system to obtain all locks in the proper order before a garbage collection.

2.2.2 Allocation

The memory system can allocate objects either by calling the collector for each allocation, or by using a per-thread cache if the collector supports it.

```
[must define]
word32 * specificAllocateWords(java_int numWords)
```

The collector should return a pointer to new memory. It does not need to clear the memory. If there is not enough room, the collector should initiate a GC by calling `getLocksThenGC()` (declared in `gcimpl.h`). If there is still not enough space, it should return `NULL`. The memory system is responsible for grabbing and releasing the heap lock, calculating the object size, initializing the object, dealing with heap expansion, and out of memory signaling.

2.2.2.1 “Fast” Allocator

If the collector supports linear allocation (allocation of various size objects in contiguous memory), it can define the macro `FAST_ALLOCATOR` to true. The memory system will then use a two-level allocation cache. The collector must provide a global cache of memory to the memory system bounded by the variables `fastAllocatorFreePtr` and `fastAllocatorLimit`. Essentially, the collector pre-allocates a block of space that the memory system may divide into objects on its own. A thread can create a local allocation buffer (LAB) by allocating out of the global cache (avoiding calls to the collector, but still requiring the heap lock). Each thread can then allocate objects out of its private LAB without synchronizing with other threads.

The collector needs to update `fastAllocatorFreePtr` and `fastAllocatorLimit` at the end of each call to `specificInitHeap()`, `specificGC()`, and `specificExpandHeap()`. `fastAllocatorFreePtr` should be set to point to the heap's “freeLimit” variable, so the memory system and the heap are kept in sync.

During normal execution, a thread's LAB is an unformatted array of words. When threads are suspended for GC or object iteration, the memory system formats the unused portion of the LAB to look like an array of `int`, and the thread clears its pointer to the LAB. The collector will then reclaim the unused LAB normally.

2.2.3 Collection

A collector cleans up the garbage in close cooperation with the rest of the memory system. The memory system does a lot of the work before calling the collector's GC routine, and provides essential services to the collector.

A collector can do a “full” collection (collect all objects in the heap), or a “partial” collection. Some collectors can only do full collections, but generational collectors (and others) may do partial collections.

Note: Concurrent collectors may do most collection work as part of allocation and/or read/write barriers or in a separate thread. The collection steps described below may only apply to the non-concurrent portion of collection (or not at all).

Collections may be initiated by Java code explicitly (using `java.lang.Runtime.gc()`), by the JVM for profiling or debugging, or more commonly by the collector when an allocation fails. All cases end up calling the memory system routine `getLocksThenGC()`.

2.2.3.1 Memory System's Responsibilities

`getLocksThenGC()` will:

1. Get the heap lock.
2. Start a GC thread to do the rest of collection:
3. Get the registered GC locks in order.
4. Suspend all threads.
5. Run the inconsistent threads until they are consistent and they suspend themselves.
6. Call the collector's `specificGC()`.
7. Unload unreferenced classes (if `specificGC()` did a full GC).
8. Call the collector's `specificExpandHeap()` if collection did not free enough memory.
9. Resume the threads.
10. Release the GC locks.
11. Enqueue discovered weak reference objects that have dying referents for processing.

2.2.3.2 Collector's Responsibilities

[must define]

```
bool_t specificGC(java_int wordsRequested, bool_t fullCollection)
```

The collector should attempt to remove at least `wordsRequested` words of garbage from the heap. The memory system may request a full collection or a partial collection, but the collector can ignore this argument. The collector returns `TRUE` if it did a full collection and `FALSE` otherwise.

Most collectors work by walking the object graph from roots, keeping all reachable objects alive, and recycling the space used by unreachable objects. The memory system provides functions to iterate over all roots—Java language local and global variables, VM-registered handles, and JNI handles. A macro to iterate over all reference fields of a given object is also provided. The collector is free to use some low-order bits in the header of each object, and in each pointer for marking, etc. In fact, it may modify the pointers in the object graph in any way it pleases as long as it restores the graph at the end of the collection and as long as it is able to recover the `NearClass` for a given object during the collection.

To iterate over the roots, the collector must call:

[must use]

```
void processStrongRoots(RootCallback globalRootCB, RootCallback localRootCB,  
                        RootCallback stackRootCB, RootCallback classRootCB,  
                        bool_t allClasses, void *data)
```

The collector must provide callback functions for processing each VM global root, VM local root, Java stack root, JNI handle, and root in classes that are active on the stack or otherwise not unloadable. Often the same callback function can be used for all root types. The `data` parameter is passed unchanged to the callback functions. If a collector is only doing a partial collection, then it must treat all classes as roots. The collector should use `allClasses TRUE` in this case.

RootCallbacks

The signature of `RootCallback` is the same as `RefIterator`:

```
void (*RootCallback)(java_lang_Object **objp, void *data)
```

A root callback is responsible for processing the references in the object referred to by the object pointer `objp`, and updating the object pointer if the collector moves the object. Two support routines are provided to help the callback. The first takes care of references in the object's class:

```
[must use]
static void scanNearClasses(java_lang_Object *obj, NearClass *ncls,
                             RootCallback classRootCB, void *data)
```

If the `NearClass` structure referenced by `ncls` is unmarked, this function will mark it and call the `classRootCB` callback on references found in the `Class` structure referred to by `ncls`, including:

- Various objects referred to directly by the class structure (class loader, name string, etc).
- All static object variables of the class.
- All of the class' constant pool references.

This function will also be called recursively on all of the unmarked classes referred to by the class, including:

- The superclass of the class.
- The class representing an “array of” the class if it exists.
- If the class is an array class, the class representing the array's declared element class.

To process the references in the object itself, the collector should use the following macro provided by the memory system:

```
[must use]
macro OBJ_REFS_DO(java_lang_Object *obj, NearClass *ncls, statements)
```

This macro creates an iteration variable named `refPtr` (of type `java_lang_Object**`) and executes *statements* once for each reference in the object, with `refPtr` set to the location of the reference. As a side-effect, `OBJ_REFS_DO()` may discover that `obj` is a weak reference object that needs to be treated specially (described in detail below).

2.2.3.3 Collecting Weak References

Weak reference objects were described in section 1.2.3. Weak reference objects have three fields and an implicit state that are of interest to the memory system:

- `referent`. The `referent` field is treated specially by the memory system only during the discovery phase and only if the state is `ACTIVE`.
- `next`. This field is used to link weak reference objects into user-level queues. The `next` field is always `null` until the weak reference object is enqueued.
- `discovered`. This field, added by the class loader if not present, is a field only manipulated by the garbage collector. Like the `next` field, this field is used to link discovered weak reference objects so that at the end of a collection these objects may be processed. It is separate from the `next` field because application code may manipulate that field (for example, by explicitly enqueueing a reference object on its queue) and this activity must be supported in the presence of concurrent and incremental garbage collectors.

- There are several implicit states, `ACTIVE`, `PENDING`, `ENQUEUED`, and `INACTIVE`, but the memory system is only concerned about reference objects in the `ACTIVE` state. This is encoded as `next = null` and `discovered = null`.

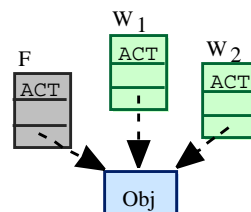
To implement weak references, the memory system and collector collaborate in several phases:

1. **Discovery** phase. Discover weak reference objects and link them together into a `discoveredRefs` queue—a list of potential weak reference objects to process. This is accomplished by invoking the `startDiscoveringWeakRefs()` and then calling `OBJ_REFS_DO()` to walk over the objects in the heap. At the end of the traversal, `stopDiscoveringWeakRefs()` is invoked. The collector must call these in order to do the “scanning” phase of the collection algorithm.
2. **Processing** phase. Once the collector has determined all of the strongly reachable objects, the memory system and collector process the `discoveredRefs` queue and create a `tempPendingQ` of weak reference objects that have referents that are only weakly reachable. Processing actually happens once for each strength of weak reference object. Processing is done by `processWeakRoots()`, which the collector must call.
3. **Updating** phase (optional). During discovery, the memory system may have set the `next` and `discovered` fields but the collector has not yet seen these fields. This phase allows collectors that care to take action based on these updated fields.
4. **Queuing** phase. At the end of the garbage collection, the elements of the `tempPendingQ` are pushed onto the queue rooted in the Java static variable `java.lang.ref.Reference.pending`. This queue is drained by Java code that separates the weak reference objects into other various user-level queues (including the finalization queue). The reason that a temporary queue is used in the previous phases is that the processing and updating phases do not have to deal a potentially non-empty global queue. Queuing is done by `queuePendingRefs()`, which the collector must call.
5. **Notification** phase. If any weak reference objects were added in the queuing phase, the Java code (described above) that processes the global queue is notified. Notification is done by the memory system at the end of a collection (in `getLocksThenGC()`).

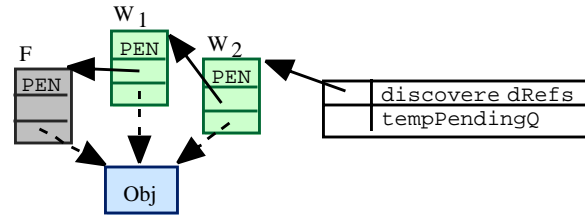
2.2.3.4 Weak References Example

This example shows the discovery, processing, and queuing phases in more detail. Consider a heap that consists of an object `obj` with a `finalize()` method, and two `WeakReference` objects (`w1` and `w2`) that weakly refer to `obj`. The JVM creates a `FinalReference` object `F` to handle the `finalize()` method, and links it into a global list.

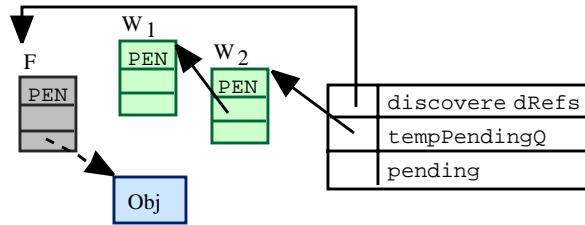
Initial state. Assume that `w1` and `w2` are rooted by Java static variables and there are no longer any strong references to `obj`. If we only look at each weak reference object's implicit state and its `discovered` and `referent` fields, the heap looks like this before a collection. All the weak reference objects are `ACTIVE` (“ACT”).



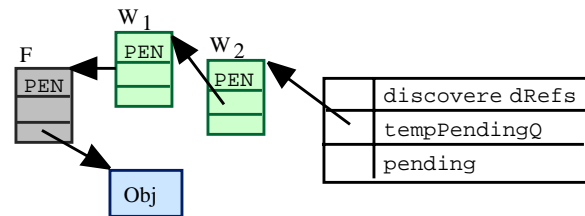
Discovery. While scanning from roots, the collector eventually calls `OBJ_REFS_DO()` on each of the three weak reference objects. `OBJ_REFS_DO()` notices that the weak references are `ACTIVE`, so they are enqueued and the collector does not yet see the `referent` field (or `obj`). Since the memory system sets the discovered fields of these weak reference objects, their states are now `PENDING` (“PEN”).



Processing (part A). First the `WeakReference` objects are processed. The memory system and collector notice that `obj` is not strongly referenced, so `w1` and `w2` are enqueued via their `next` fields on the `tempPendingQ` and the `referent` fields are cleared.



Processing (part B). Next, the `FinalReference` object is processed. Since `obj` is still not strongly reachable, `f` is enqueued and the memory system makes a collector callback (`weakRefCB`) on its `referent` field. The collector will mark or copy `obj` and update the `referent` field if necessary.



Queuing & Notification. At the end of the collection, the weak reference objects are removed from the `tempPendingQ` and placed in the `java.lang.ref.Reference.pending queue`. The memory system notifies Java code that the `pending queue` needs processing.

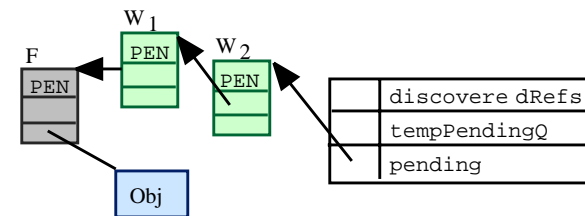


FIGURE 3. Weak Reference Processing

Subsequent states. After the collection completes and the mutator threads are resumed, Java code will eventually process the `pending queue`. The `FinalizerThread` will process the `FinalReference` and clear its `referent` field. A future collection will then notice that there are no references to `obj`, so it may be reclaimed.

The `OBJ_REFS_DO()` macro will treat weak reference objects specially during the “discovery phase” (controlled by `{start|stop}DiscoveringWeakRefs()`). In discovery mode, `obj` will be

linked into a `discoveredRefs` queue if it is a weak reference object that is `ACTIVE`. The macro will not execute *statements* on the `referent` field of a discovered weak reference object. The collector should start the discovery phase while discovering all live objects starting from strong roots (the mark phase of a Mark & Sweep collector, for example), and stop the discovery phase before processing weak roots. If the collector makes further iterations over the object graph, it can use the “non-discovering” version of the macro called `OBJ_REFS_DO_IGNOREING_WEAKNESS()`, which treats all fields of weak reference objects as strong references.

After completing a pass through all objects reachable from strong roots, the collector should call `stopDiscoveringWeakRefs()`, then process the weak roots by calling:

```
[must use]
void processWeakRoots(WeakRootPredicate isDying,
                        RootCallback weakVMRootCB, RootCallback weakRefCB,
                        SimpleFct transitiveScannerCB, void *data)
```

This function will process all weak roots in the JVM, which includes the `discoveredRefs` queue of weak reference objects discovered by `OBJ_REFS_DO()` as well as certain JVM data structures like the string interning table. The collector needs to pass a predicate function as the `isDying` parameter, which should return `TRUE` if the object passed is dying (not marked, or not in “to space,” etc.) The collector also needs to supply several callbacks as described below. The memory system processes the weak roots as follows:

1. For each “strength” of weak reference (from strongest to weakest: soft, weak, final, phantom) on the `discoveredRefs` queue discovered by `OBJ_REFS_DO()` do:
 - 1.1. Invoke `isDying()` on each `referent` field of each reference object of that strength.
 - If `isDying()` returns false, update the `referent` field of the weak reference object by calling the `weakRefCB()` on the field address. Then remove the reference object from the `discoveredRefs` queue (which return the reference to the `ACTIVE` state).
 - If `isDying()` returns true and if processing `SoftReference` objects (and memory is tight), or `WeakReference` objects, then clear the `referent` field and link the weak reference object into the `tempPendingQ`. The `tempPendingQ` is a global root which is always `NULL` at the beginning of GC, but subsequent calls to `processStrongRoots()` or `processRootsRound2()` will see it.
 - 1.2. If processing `FinalReference` or `PhantomReference` objects, then in a second pass, scan and update the `referent` field of each reference object by calling the `weakRefCB()` on the field address. Then link the weak reference object into the `tempPendingQ`. Note that the first pass removed all of the weak reference objects with non-dying referents.
 - 1.3. If the `weakRefCB()` does not scan the transitive closure of the objects reachable from the `referent` fields, then the caller must provide a `transitiveScannerCB` to do this. Otherwise, the collector can pass `NULL` as the `transitiveScannerCB` parameter.
2. For each internal weak data structure in the VM (weaker than Java weak references):
 - 2.1. Invoke `isDying()` on each weak reference.
 - 2.2. In a second pass, invoke `weakVMRootCB()` on each weak reference that the memory system wants to keep alive.

The function `processWeakRoots()` creates a `tempPendingQ` of weak reference objects by linking the objects together using their `next` field. But the collector has never seen these fields. If the collector called `OBJ_REFS_DO()` on objects before they were moved, or if all pointers in the heap

need to be “reversed” or otherwise processed, then the collector must call `updateTempPendingQ()` passing a `weakRefUpdaterCB()` callback.

2.2.3.5 Before and After Collection

There is some bookkeeping that the collector must do before and after a collection. Before starting an iteration over reachable objects (for example, a marking phase), the collector must call `clearClassReferencedBits()` to mark each class structure as “unreferenced,” so that `scanNearClasses()` can mark the referenced classes correctly. The collector must also call `startDiscoveringWeakRefs()` to tell the memory system (`OBJ_REFS_DO()` in particular) to begin discovery phase mode.

After the collection has been completed (and the object graph has been restored), the collector needs to call `queuePendingRefs()` to merge the weak reference objects on the temporary pending list onto the Java static variable `pendingQ`. This global queue may not have been empty at the beginning of GC. If this GC added items to the global queue, the memory system will notify Java code to take care of the queue.

If the collector has moved any objects, it needs to increment `gcMoveCount`.

2.2.3.6 Example Collector: ReallySimpleHeap

ReallySimpleHeap is a mark/sweep collector—a simplified version of the “simpleheap” collector in the EVM. Here is the code for a simple non-moving collector (but not allocation or free lists):

```
void scanObjectCB(java_lang_Object **root, void* data) {
    java_lang_Object *obj = *root;
    NearClass *ncls = getNearClass(obj);
    if (hasLiveMark(ncls)) return;                /* Already done. */
    setNearClass(obj, setLiveMark(ncls));          /* Mark it. */
    scanNearClasses(obj, ncls, scanObjectCB, data);
    OBJ_REFS_DO(obj, ncls, if (*refPtr) scanObjectCB(refPtr, data));
    /* note that "refPtr" is defined by OBJ_REFS_DO macro */
}

bool_t isDyingCB(java_lang_Object *obj, void *data) {
    return !hasLiveMarkObj(obj);
}

void mark() {
    clearClassReferencedBits();
    startDiscoveringWeakRefs();
    processStrongRoots(scanObjectCB, scanObjectCB, scanObjectCB, scanObjectCB,
                       FALSE,          /* Don't scan all classes */
                       NULL);          /* No data needed */
    stopDiscoveringWeakRefs();
    processWeakRoots(isDyingCB, scanObjectCB, scanObjectCB,
                    NULL,          /* scanObjectCB did transitive closure */
                    NULL);          /* No data needed */
}

void freeDeadObjectCB(java_lang_Object *obj, void *data) {
    if (isDyingCB(obj, NULL)) myFreeObject(obj, ...); /* link onto free list */
}
```

```

void sweep() { /* walk over heap, putting unmarked objects onto a free list. */
    specificObjectsDo(freeDeadObjectCB, NULL);
}

bool_t specificGC(java_int wordsRequested, bool_t fullCollection) {
    mark();
    sweep();
    queuePendingRefs(); /* Tell Java code about weak reference objects */
    return TRUE; /* It was a full GC, so sweep classes. */
}

```

The use of recursion in `scanObjectCB` above would result in poor performance and stack use, but makes the example simpler.

2.2.4 Expansion

A collector typically expands the heap when asked to by the memory system (see the description of `getLocksThenGC()` above). Some collectors also call their `specificExpandHeap()` as a way to allocate the initial heap in `specificInitHeap()`. Either way, the collector can count on having the heap lock and that all threads are suspended.

```

[must define]
uintptr_t specificExpandHeap(uintptr_t minWords,
                               uintptr_t suggestedWords)

```

The collector should attempt to expand the heap by at least `minWords` up to `suggestedWords`. The memory system currently uses an aggressive heap expansion strategy, and suggests doubling the heap size at each expansion. The result should be 0 if the expansion failed, otherwise the actual number of words added to the heap. The sizes refer to the number of words available for objects.

2.2.5 Iteration

The memory system provides a function `allObjectsDo()` that iterates over all objects in the heap. The memory system will suspend all threads, update the threads' LABs, then call the collector's heap iterator:

```

[must define]
void specificObjectsDo(ObjIterator fct, void *data)

```

The collector must call `fct` exactly once for each object in the heap, passing it a reference to the object and `data`. The collector is allowed to call `fct` with objects that would be “dead” if a GC was done before the call to `specificObjectsDo`.

2.3 Read/Write Barriers

A collector may also implement barrier macros, which give the collector hooks into the JVM to monitor various activities. Barriers don't actually do a read, write, or allocate; they just do any necessary bookkeeping. There are barriers for Java heap and Java static variable access, but not for Java stack access (because of type-indeterminate bytecodes like “dup”).

A collector must define all of the following macros, but it may leave the macro bodies empty for all unnecessary operations. There are several general rules for barriers:

- Read barrier macros must be an expression returning a value, though this value will be ignored. This allows callers to use read barriers in comma-expressions.

- Write barrier macros take the object being modified (heap barrier only), the address of the field/variable to be modified, and the new value. Write barriers are called before the new value is written so the barrier can see the old value and the new value.
- All barrier macros can only be called when the caller is inconsistent.

[must define]

```
bool_t READ_HEAP_BARRIER(java_lang_Object *obj,
                          java_lang_Object **refAddr)
void WRITE_HEAP_BARRIER(java_lang_Object *obj,
                          java_lang_Object **refAddr,
                          java_lang_Object *rr)
bool_t READ_STATIC_BARRIER(java_lang_Object **refAddr)
void WRITE_STATIC_BARRIER(java_lang_Object **refAddr,
                           java_lang_Object *rr)
```

The read/write barriers are enforced by the convention in the VM of only accessing objects through the `memsys` interface or the LLNI interface (which uses `memsys`). The `memsys` interface provides a variety of object class and hash code accessors, object field accessors, array element accessors, and static field accessors. The only code which is allowed to directly manipulate object state is the `memsys` interface and the collector itself.

If a collector does not need any read or write barriers, or it can do its bookkeeping in batches, it should define the macros `canBatchReadBarrier` and/or `canBatchWriteBarrier` to `TRUE`. If a collector can do its bookkeeping in batches, then it should also define one or both of these macros:

[must define]

```
void BATCH_HEAP_READ_BARRIER(java_lang_Object **startAddr,
                              java_int numWords)
void BATCH_HEAP_WRITE_BARRIER(java_lang_Object **startAddr,
                               java_int numWords)
```

If supported, these macros will be called before copying object arrays (in `JVM_ArrayCopy()`), and before cloning objects with reference fields (in `inconsistentClone()`). `startAddr` points to the first word to be read or written. Unlike the other write barriers, `BATCH_HEAP_WRITE_BARRIER` does not get to see the new values.

Note that there is no fast way to get exact information about where pointers are located when cloning objects, so batch barriers are best suited for collectors that use inexact remembered sets (which remember the object accessed rather than the word accessed).

There are other barriers that give the collector hooks into the JVM.

[must define]

```
void NEW_INSTANCE_BARRIER(ExecEnv* ee, java_lang_Object *refAddr)
void FAST_NEW_INSTANCE_BARRIER(ExecEnv* ee, java_lang_Object *refAddr)
```

The new instance hooks are called after every object has been allocated and initialized. The object's class information will be set. The barrier is called within the same inconsistent region as the allocation. This barrier allows a collector to do any per-object bookkeeping. The `FAST_NEW_INSTANCE_BARRIER` is only called during an allocation out of a thread's LAB, so the collector may make certain assumptions (such as the LAB is always in the youngest generation).

Finally, the handle barrier detects writes to global roots by being called every time an LLNI handle is updated. It is called before the write takes place.


```
[must define]
void WRITE_HANDLE_BARRIER(ExecEnv* ee, Ijava_lang_Object refAddr,
                           java_lang_Object *rr)
```

2.3.1 Example Barriers

ReallySimpleHeap doesn't require any barriers, so the following example comes from the EVM's generational heap. The generational heap uses card tables for remembered sets, so it defines:

```
#define READ_HEAP_BARRIER(obj, refAddr)      TRUE
#define WRITE_HEAP_BARRIER(obj, refAddr, rr) setCTEModPtr((word32*)(refAddr))

#define canBatchReadBarrier  TRUE
#define canBatchWriteBarrier TRUE

#define BATCH_HEAP_READ_BARRIER(startAddr, len)
#define BATCH_HEAP_WRITE_BARRIER(startAddr, len) \
    setCTEModRange((word32*)(startAddr), (len))
```

Note that `setCTEModPtr()` is a static inline function in the generational collector. This combines the better type-checking and debugging of function calls with the efficiency of macros.

2.4 Supporting Collector Routines

In addition to the core routines, there are also a number of support routines that must be defined. These provide support for information about the state of the collector, verification, descriptions, and a few object-specific functions.

2.4.1 State of the Collector

There are several functions that need to be defined in order for the VM to know how much memory is available for allocation and how much is in use:

```
[must define]
uintptr_t specificTotalObjectWords()
uintptr_t specificTotalHeapWords()
```

These two functions report how much memory has been used. The first reports the total amount of memory in words that is available for allocating objects. The second reports the amount of memory in words used by the collector including memory for supporting data structures including “to” space, card tables or remembered sets, and similar structures. `specificTotalHeapWords()` is used by debugging and heap analysis tools, and does not need to track every single byte.

```
[must define]
uintptr_t specificMaxAllocation()
```

This function reports the size of the largest available chunk of free memory in the heap in words. The memory system calls this after a collection to determine if the allocation request that triggered the collection will succeed. This function is allowed to return a conservative estimate at the cost of expanding the heap earlier than needed on occasion.

```
[must define]
uintptr_t specificFreeHeapWords()
```

This function returns the approximate number of words free in the heap at a point in time. The result is approximate because the collector must calculate this quantity without locking the heap.

2.4.2 Collector Verification

An important part of debugging the behavior of the VM is the ability to verify that the collector and the heap it manages are in good shape. To support this process, 3 functions must be defined. While these functions do not return anything, they may abort if there are any discrepancies.

```
[must define]
void specificVerifyHeap()
```

This function walks over the heap and makes sure that any collector-specific invariants hold for the heap. It can be slow since it is only used for debugging purposes, and it may abort the VM.

```
[must define]
void specificVerifyReference(java_lang_Object *obj)
```

This function does any collector-specific checks to make sure that a reference is valid. This can range from verifying that the pointer points into the heap that the collector is managing to verifying bits in the objects header and class, etc. It assumes that `obj` is not null and is properly aligned.

```
[must define]
void specificVerifyIsFree(word32 *from, word32 *to)
```

This function verifies that words in the range `[from, to)` are free and is typically called by the collector itself in debug mode before allocating words.

2.4.3 Descriptive Functions

There are two functions that provide some level of description of a given collector:

```
[must define]
char* specificDescribeMemSys()
char* specificName()
```

The first function, `specificDescribeMemSys()`, returns a description of the specific memory system. This description may consist of one or more lines, each terminated with a newline character. The second, `specificName()`, returns a simple name for the collector. The name should not contain a newline character.

2.4.4 Object-specific Functions

Some collectors make use of the headers of objects during a collection. For example, some collectors move an object's class field to the side in order to link related objects together. Because other parts of the VM may need to access the class information of an object while a collection is in progress, two functions must be provided:

```
[must define]
NearClass* specificGetNearClass(java_lang_Object *obj)
bool_t      specificIsObjectInGCNow(java_lang_Object *obj)
```

The first function will return the actual class of an object even if that class information has been temporarily replaced or obliterated in the object's header. So, unlike IBM punch cards, objects can be marked, mangled, folded, mutilated, or spindled as long as we can use this function to recover the information we need. The second function will return `TRUE` if and only if the object is in an area that is currently being collected. In both cases, the caller is expected to check any general properties, and, in particular, handle the case where the argument is `NULL`.

3. Addenda, Open Issues, and Future Directions

As promised in the Introduction, the memory system interface has changed. Here is a list of known problems and future enhancements.

- Since this paper was written, we have generalized heap expansion to cover resizing in general. This needs to be documented.
- Currently, the JIT compiler has direct knowledge of the write barriers that it must generate. In fact, it doesn't support read barriers yet. Each collector has to implement a barrier-generating function using undocumented routines in the JIT compiler. There is a need to design and document a more flexible system.
- Our generational framework currently supports the unloading of classes after any combination of generations has been collected. Likewise, we are able to do weak-reference discovery across generations.
- We have added support for parallel, concurrent, and incremental GC including lock-free support for load-balancing.
- We use a variety of suspension mechanisms: some based on polling, some through the use of signals. Native code is now “stable” meaning that it is not interrupted or affected by garbage collection suspension.
- We have implemented a general LAB mechanism and have used this to successfully pre-tune objects based on dynamically gathered information about object lifetimes as well as in implementing parallel collection.
- We have experimented with different kinds of write-barriers. In addition to the default card-marking barrier, we have also implemented a barrier that uses sequential store buffers.
- We may add generational support to the “abstract collector” interface. We currently have a flexible generational framework that is built on top of the “abstract collector” interface. We would like to be able to handle “multi-area” allocation and collection for cases like heaps-by-size (large object areas), heaps-by-type, as well as heaps-by-age (generational collection). For our own internal use, we should also document the existing generational framework.
- 64-bit support has not been fully implemented. In this document, a “word” can be counted on to be 32 bits on a 32-bit system. It is not clear what it means on 64-bit system.

4. More Information

4.1 References and Further Reading

1. Gosling, J., Joy, B., Steele, G., *The JavaTM Language Specification*. Addison-Wesley, 1996.
<http://java.sun.com/docs/books/jls/>
2. *Java Native Interface Specification*. Sun Microsystems, 1997.
<http://java.sun.com/products/jdk/1.2/docs/guide/jni/>
3. *JavaTM Platform 1.2 API Specification*. Sun Microsystems, 1998.
<http://java.sun.com/products/jdk/1.2/docs/api/>
4. Jones, R., Lins, R., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
5. Lindholm, T., Yellin, F., *The JavaTM Virtual Machine Specification*. Addison-Wesley, 1996.
<http://java.sun.com/docs/books/vmspec/>
6. Pawlan, M., *Reference Objects and Garbage Collection*. Java Developer Connection technical article, Sun Microsystems, 1998.
<http://developer.java.sun.com/developer/technicalArticles/>
7. Wilson, P., *Uniprocessor garbage collection techniques*. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.
<ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps>.
8. Jones, R., *Richard Jones' Garbage Collection Page*.
http://stork.ukc.ac.uk/computer_science/Html/Jones/gc.html.
9. Chase, D., *GC FAQ*.
<http://www.iecc.com/gclist/GC-faq.html>

Finally, the complete version of this document (*The GC Interface in the EVM*) is available online
<http://www.sunlabs.com/technical-reports>.

4.2 Credits

The design of the memory system interface is a product of the Java Topics Group at Sun Microsystems Laboratories at Burlington, MA, which includes Ole Agesen, Corky Cartwright (Visiting Professor), Dave Detlefs, Christine Flood, Alex Garthwaite, Steve Heller, Tony Printezis (intern), Guy Steele, and Derek White.

Thanks to the Java Topics Group, Glenn Skinner, Mario Wolczko, Bernd Mathiske, and Eliot Moss for reviewing this document.

Hybrid TCP–UDP Transport for Web Traffic

Israel Cidon, Amit Gupta, Raphael Rom and Christoph Schuba

Introduction by Raphael Rom

The World Wide Web is a given today. But that was not always the case. About a decade ago, the WWW was a small experiment (that exploded) and, as such, its designers took some shortcuts to enable them to prove a point — only to discover later that some of those decisions are to haunt us for a while (maybe forever).

One of these decisions was to base HTTP on TCP. HTTP, is the web protocol and TCP is its underlying transport. But the two do not really match. HTTP is a transaction-oriented protocol and TCP is a connection-oriented protocol and that is all the story — the main problem being that to serve thousands of concurrent Web requests requires an equal amount of TCP connections which, in turn, require keeping context for those connections, which leads to large overhead, bad performance, and rigidity. Introducing new concepts such as QoS in web service (and not just the communication portion) proved to be hard to achieve.

An obvious solution was to base the HTTP protocol on UDP and thus avoid the connection set-up for short transactions and allow for new and innovative web-based services. But the world cannot be easily changed, and backward compatibility is necessary. This led to the work described in this report, in which HTTP is based on a hybrid approach: if UDP can be used, it will and should, and will result in better service. For the die-hards, who insist on using TCP, that is also acceptable. And, best of all, when a UDP based system determines that in certain cases TCP connections are preferred, a TCP connection will be put in place.

The work itself consisted of two parts. The first is a general analysis and parameter optimization, which is reported in this paper. It analyzes traffic patterns, UDP parameters, threshold settings, and other web characteristics. This is reported in the paper which you are urged to read. The other part, subsequently done by the authors with some students, was the creation of a complete and comprehensive web service that provides distributed hosting with load balancing based on both communications and processing and with other QoS variations. The entire system was written in the Java™ programming language and indeed has demonstrated the superiority of that approach.

PUBLICATIONS:

I. Cidon, A. Gupta, R. Rom and C. Schuba "Hybrid TCP-UDP Transport for Web Traffic" Pages 177-184 in Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99), Phoenix, Arizona, February 10-12, 1999.

V. Sokhin (I. Cidon and R. Rom supervisors) "Load balancing and quality of service for distributed web services" M.Sc. Thesis, Technion, Israel Institute of Technology, February 2001.

Hybrid TCP-UDP Transport for Web Traffic

Israel Cidon, Amit Gupta, Raphael Rom, and Christoph Schuba

SMLI TR-99-71

January 1999

Abstract:

Most of the Web traffic today uses the *HyperText Transfer Protocol* (HTTP), with the *Transmission Control Protocol* (TCP) as the underlying transport protocol. TCP provides several important services to HTTP, including reliable data transfer and congestion control. Unfortunately, TCP is poorly suited for the short conversations that comprise a significant component of Web traffic. The overhead of setting up and tearing down TCP state amortizes poorly for these small connections. Moreover, emerging modern Web server systems employ HTTP *redirection* for server load-balancing and content distribution; such schemes require setting up (and tearing down) multiple TCP connections for servicing a single client request.

We have designed and analyzed a hybrid scheme to address these issues. The scheme uses either TCP, or the *User Datagram Protocol* (UDP) as the underlying transport protocol for carrying Web traffic. UDP is used for short transfers (including HTTP redirection), while TCP is used for all other transfers. In this manner, we avoid the extra TCP overhead for short connections, but still benefit from the reliable delivery and congestion control that TCP provides. We ran trace-based simulations to quantify the effects of various network parameters (i.e., packet loss rates) on the performance of the hybrid scheme. We observed performance gains exceeding 20-25% with HTTP/1.1-style persistent connections, and over 40-50% without persistent connections. These gains can be improved with further performance optimizations that we describe.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:

israel.cidon@eng.sun.com
raphael.rom@eng.sun.com
christoph.schuba@eng.sun.com

Hybrid TCP-UDP Transport for Web Traffic

Israel Cidon, Amit Gupta, Raphael Rom, and Christoph Schuba

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303

1 Introduction

As the *World Wide Web* (WWW) and other Internet applications, such as real-time audio and video become more and more popular, both the Internet itself and the most popular sites are suffering from severe congestion. This congestion is perceived by users as service delays at best, and as a lack of service at worst.

The implications for companies depending on the availability of the service are grave: *Connection timed out* is read as *Closed for business*. Given the revenue-generating and often mission-critical service that many Web sites provide, high availability is of great importance and desirable to be maintained, even in the presence of multiple system failures.

Most of the Web traffic today uses the *HyperText Transfer Protocol* (HTTP). Traditionally, HTTP uses the *Transmission Control Protocol* (TCP) [15] as its underlying transport protocol. TCP provides several important services to HTTP, including reliable data transfer and congestion control. Unfortunately, TCP is poorly suited for the short conversations that comprise a significant component of Web traffic. The overhead of setting up and tearing down TCP state amortizes poorly for these small connections.

Many Web sites today use a variety of approaches, including server and content replication and distribution, to improve service latency and reliability. Such features are best served if the HTTP address (URL) is looked at and processed before the final destination server is selected. Current HTTP *redirection* solutions require tearing down and setting up multiple TCP connections. The redirection operation itself is a short-lived HTTP connection. Several commercial products such as the BrightTiger¹ use HTTP redirection for content distribution at the added expense of setting up multiple TCP connections.

Others avoid the overhead associated with HTTP redirection but, consequently, suffer from other limitations such as the need to fully replicate the server content (if URL is not looked at before the destination is selected,) or the need to transfer TCP connections with their associated states between servers (see for example Resonate at <http://www.Resonate.com/> or the Cisco LocalDirector at <http://www.cisco.com/warp/public/751/lodir>).

¹<http://www.brighttiger.com/>

This paper takes a different approach and tackles the problem at its roots. We propose a dynamic transport protocol selection for reducing service latency, traffic overhead, and server load by giving services a dynamic choice of which transport protocol to use. Today, Web services usually use a single transport protocol, namely TCP, for all traffic; dynamically selecting the underlying transport protocol (e.g., TCP or UDP) can substantially improve service latency and reduce network traffic and server load. Our approach is conservative in the sense that we analyze and test our proposal against current Web traces. The rationale is that in the future load, content, and geographical distribution of servers within a service provider domain (say a video on demand or network computing services etc.) will further increase the need for redirection operations. Similarly, the increasing use of style sheets as well as the increasing use of cache validation also lead to a large number of short HTTP transactions. This will emphasize the overhead reduction even further.

The mechanisms discussed in this paper apply to many client-server protocols; we concentrate on the *HyperText Transport Protocol* (HTTP) [3] as the canonical example.

Available Internet traffic traces [4] show that a large part of Web traffic consists of short HTTP transactions: about 40% of Web transfers can fit in a single 1500 byte datagram, the size of an Ethernet *maximum transmission unit* (MTU), and approximately 63% will fit into two datagrams. In the case of a single MTU, using TCP requires 4-5 times the number of packets compared to the use of UDP. TCP setup also requires more complex state machine operations and setup of data structures at both the server and the client. Consequently, using UDP reduces network traffic as well as client-observed latency.

This hybrid TCP-UDP scheme combines the best of both worlds: for short transactions, it benefits from stateless UDP's low overheads; on the other hand, for large data transfers, it obtains the desired reliability, resequencing, and congestion control from TCP. The hybrid scheme is attractive because it only requires application-level changes, while the operating system kernel code can remain unchanged. Finally, the scheme is incrementally deployable in the current Internet and is fully compatible with the installed base.

Previous research investigated the cost of high TCP overhead for small connections. We address here two of these schemes: T/TCP [1] and HTTP Performance modeling research by Heidemann, Obraczka, and Touch [5].

Another motivation for the design of the hybrid scheme is to guarantee that our scheme will preserve the "TCP-friendly" property. This means that it should not attempt to benefit a particular service (in terms of better response time or throughput) at the expense of the global network or other well behaving users. We maintain the TCP congestion control for all long-term connections. On the other hand, since TCP congestion control is not effective for short-lived transactions (because of the lack of proper round trip adaptation periods), the use of UDP in these instances does not make a difference in that respect.

TCP for Transactions (T/TCP) [1] was developed as a transport protocol for request/reply type message passing protocols. It avoids the overhead that explicit connection setup and tear-down phases impose for small transactions. The designers of T/TCP defined a reliable request/response handshake with exactly one packet in each direction. T/TCP is designed as a backward-compatible extension to the TCP protocol. For Web traffic, T/TCP behavior and performance is similar to that of persistent-HTTP: the first connection between a pair of hosts requires a three-way

handshake while successive connections avoid this overhead. T/TCP's requirement for a kernel-level implementation is the primary reason why T/TCP is not widely available[10].

Heidemann et al., [5] evaluates the performance of HTTP over UDP. Unlike our scheme (which uses TCP if needed), [5] relies entirely on using UDP as the transport protocol. UDP was augmented with adaptive retransmission and congestion control mechanisms, thereby mimicking the behavior of TCP similar to the *Asynchronous Reliable Delivery Protocol* (ARDP) [9]. The disadvantage of such a scheme is that we need to develop yet another complex protocol with reliability, resequencing and congestion avoidance duplicating the long tedious process of TCP development. Moreover, if this is not a kernel-level implementation, it might result in a slower and less efficient operation.

The remainder of this paper is organized as follows. Section 2 provides background material and motivation. Section 3 explains our hybrid TCP-UDP scheme and discusses its strengths and weaknesses. In section 4, we evaluate the performance of this hybrid scheme via simulations. We conclude in section 5.

2 Background

In this section, we briefly describe the various transport protocols that are involved in our design, e.g., TCP, UDP, and HTTP.

The *Transmission Control Protocol* (TCP) [15] provides a reliable, connection-oriented data stream delivery service. This reliability comes at a cost, though: TCP requires a three-way handshake for connection setup, normally a graceful connection tear down and a datagram acknowledgment process. In addition, its reliability, sequencing and congestion control mechanisms introduce extra computation overhead as well. While the setup and tear down costs amortize well over long connections, they are expensive for short connections. Its adaptive congestion control, general resequencing and ARQ mechanisms are excessive for a single or few datagrams exchange.

The *User Datagram Protocol* (UDP) [12] provides a best-effort datagram service and, therefore, can operate in a more efficient manner than TCP. For example, UDP requires no connection setup or tear down, no acknowledgments, and little protocol state machine processing. On the flip side, UDP does not offer any of TCP's reliable delivery or congestion control behavior.

HTTP defines a request/reply protocol, where client applications can request data from servers by providing a *Universal Resource Locator* (URL). HTTP is used to address many different types of resources, including text, image, audio, video, executable files, index search results, and database query results.

Figure 1 illustrates a typical packet exchange for an HTTP GET request between a client and a server. Time progresses from the top of the figure downwards. First, the client establishes a TCP connection to the server with the three-way handshake (first three packets, labeled SYN, SYN+ACK, and ACK). Once the connection is established, the client transmits the GET request to the server. The GET request contains a URL to be fetched and served (labeled HTTP GET request). The server acknowledges the receipt of the request packet (labeled ACK) before it replies with the requested resource (labeled HTTP reply). The final four packets (FIN+ACK and ACK in both directions) are TCP control packets for graceful connection shutdown.

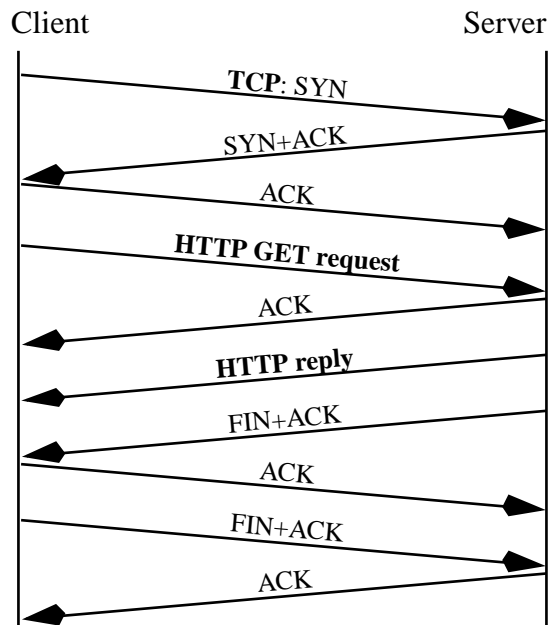


Figure 1: HTTP GET request and reply over TCP

In Figure 1, only two packets seem to be *useful* (i.e., carrying data): one is the HTTP GET request, and the other one the HTTP reply. All other packets represent TCP overhead. Under the HTTP protocol Version 1.0, each transfer requires a separate TCP connection. HTTP Version 1.1 introduced *persistent HTTP connections* to address this problem. Figure 2 illustrates persistent HTTP: multiple HTTP GET requests (and their responses) use the same TCP connection.

As of end-1998, the use of persistent HTTP is limited by several factors. About 50% of installed HTTP browsers and a large portion of HTTP servers are not capable of using persistent HTTP.

One may expect that all pages for a Web site (i.e., with a common server identifier) reside on the same server. For example, the URLs `http://www.sunlabs.com/people/index.html` and the URL `http://www.sunlabs.com/research/index.html` share the server identifier `www.sunlabs.com` and we may expect that both URLs will be fetched from the same Web server. In practice, many Web sites are set up to use different servers for different URLs. In the example described above, the first URL may be served by the server `www.people.sunlabs.com` while the second is served by `www.research.sunlabs.com`. We call this approach *heterogeneous content provisioning*.

A site may be organized in this manner because providing the client with a common site access address is appealing for administrative and business reasons. Furthermore, the nature of the data and its amount might call for its distribution (static or dynamic) among multiple servers.

There are many reasons why a site may be organized in this manner. Firstly, providing the client with a common site access address is appealing for business reasons. It is easier for clients to

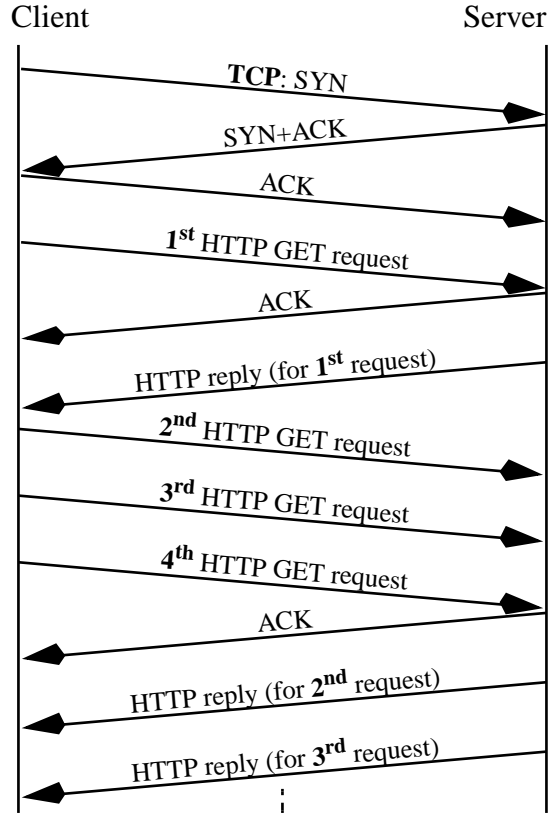


Figure 2: Example of multiple HTTP GET requests and replies over a persistent HTTP connection

remember a single commonly used site name and it gives the service provider the flexibility to index his Web services independently of their physical location. Secondly, the nature of the data and its amount might call for its distribution (static or dynamic) among multiple servers (for example, video clips might be served best by a customized video server.) Such heterogeneous provisioning may be dictated by administrative concerns (for example, multiple administrative domains). Also, a Web site may include content from multiple sources and of variable popularity; it may not be attractive to move all content to a single server, or to several fully-replicated servers.

The HTTP REDIRECT mechanism (illustrated in Figure 3) is used to support heterogeneous content. HTTP server A hands off HTTP requests from clients to server B by sending a REDIRECT as a response to an HTTP GET request. Note that the HTTP REDIRECT is part of a short HTTP transfer. Using TCP for an HTTP request–HTTP REDIRECT pair requires at least 7 packets (usually 9-10 packets), while two packets suffice if UDP is used as the underlying transport protocol.

We already mentioned that TCP transport is expensive for small payloads. Here is another example of a scenario in which UDP is a sufficient and more efficient choice for HTTP data transport than TCP. Section 4 then presents our analysis and simulation results supporting this claim, even

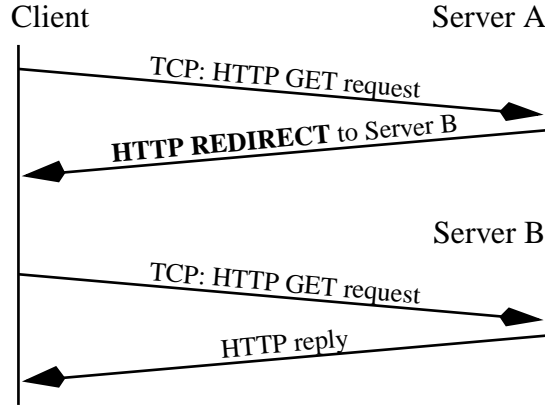


Figure 3: HTTP REDIRECT by Server A to Server B

in the presence of packet loss and corruption.

Although resources have uniform identifiers (URLs) and share a common prefix, they are usually not accumulated at a single location. Therefore, most HTTP servers operate as an intermediary, single point of access that retrieves the requested resources from a variety of other services. For example, if a requested file is not on the local file system of the HTTP server, before it can be served to the HTTP client, it first needs to be retrieved through a network file system. Database queries (results) are relayed by the HTTP server to (from) the appropriate database system. There are often multiple dedicated servers used behind the uniform front end of the HTTP server. This discussion illustrates how resources are often not arranged according to the views presented to clients. Rather, resources are distributed according to administrative structures and control (e.g., file systems that represent project file space of different engineering groups), functional criteria (e.g., customer database queries are processed by the database server, whereas files are served by a network file system), or because their collective size exceeds storage available locally.

Instead of replicating and moving data to a single HTTP server, one can imagine using multiple servers, each located closer to a subset of the data. Servers would serve only the data that is locally available to them. Files would be organized preserving administrative and organizational structures. We call this approach *heterogeneous content provisioning*. Its implementation requires the availability of multiple HTTP servers that provide to clients the illusion of a single service. The HTTP protocol includes a protocol message for the redirection of HTTP requests to other servers. It is called an HTTP REDIRECT and is illustrated in Figure 3. HTTP server A hands off HTTP requests from clients to server B by sending a REDIRECT as a response to an HTTP GET request. Using TCP for an HTTP request–HTTP REDIRECT pair requires a total of at least 7, and usually 10 TCP packets. Using UDP, two datagrams are sufficient. Additionally, the overall number of bytes sent over the network is smaller with UDP.

3 Design

A good solution for reducing the TCP overhead for small connections should have the following characteristics:

- transparency to users,
- backward compatibility,
- opportunity for incremental deployment, and
- low runtime overhead in space and time (if any)

3.1 Proposal: Hybrid TCP-UDP Transport

One approach is to use UDP instead of TCP as the transport protocol for HTTP traffic. However, in the presence of unreliable network layer communications, reliability services and congestion management need to be added to UDP to make this a viable proposal. We decided against putting such functionality into UDP.

Instead, we use a hybrid approach where short connections are served using UDP and large connections use TCP as its transport protocol. In this manner, the TCP overhead for short connections is avoided, but the benefits from TCP's well-tuned timers, retransmission, congestion control, and error recovery mechanisms are preserved. In this scheme, clients first attempt to use UDP as their transport protocol, and fall-back to using TCP, if UDP turns out to be the wrong choice for the requested URL. The fall back mechanism provides the following guarantees:

- If any of the initial UDP packets are lost, the loss is gracefully handled by switching to TCP.
- If the contacted HTTP server does not implement the hybrid scheme, the client will re-try with TCP.

Figure 4 presents this algorithm in more detail. A *hybrid* capable HTTP client sends the HTTP GET request using UDP as the underlying transport protocol. The client starts a timer at the time the request is sent.

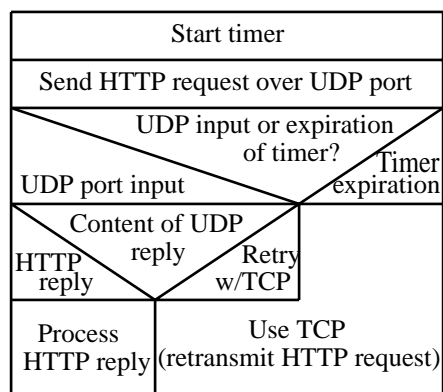


Figure 4: Algorithm executed at HTTP client application to receive HTTP service over either TCP or UDP

When the server processes the client's request, it can choose from one of the following alternatives:

1. If the response is small enough (say, it fits into one datagram), the server returns it using UDP. Small Web pages and most cache validation and HTTP REDIRECT responses fall into this category.
2. The server can ask the client to re-try using TCP if the reply is too large to fit into a single UDP packet. At this time, the server can also ask the client to try a different URL. This approach is useful if the server generated the reply dynamically and attempts to avoid generating the reply a second time for the subsequent re-try with TCP. This extension can be implemented with a new HTTP return code.

At the client side, one of the following three events can happen:

- The client gets a response from the server. If the reply contains the desired HTTP reply, the client processes the data. If the server asks the client to re-try (a different URL and/or using TCP), the client does so.
- If the server does not handle HTTP packets sent over UDP, the client may get an ICMP (*Internet Control Message Protocol* [14]) error message (destination unreachable/protocol unreachable). In this case, the client should re-try using TCP.
- If the timer expires, the client should re-try with TCP.

Figure 5 illustrates the packet exchange in case the timer expires. This timeout feature provides reliability and backwards compatibility with servers that do not use the hybrid TCP-UDP scheme. We recommend that this timeout interval be set the same as the corresponding TCP initial timeout

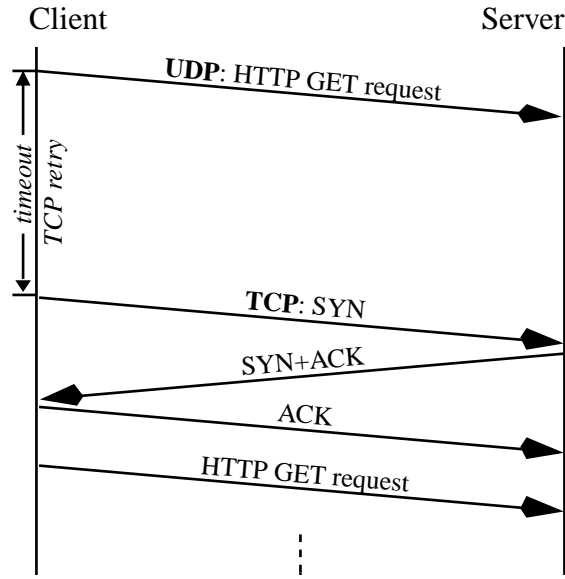


Figure 5: Implicit fall-back from UDP to TCP: One of the UDP request/response packets got lost or the server is not UDP capable

interval. As the upgraded clients can work with servers that do not implement the hybrid scheme, this fall-back mechanism supports gradual, incremental deployment in the Internet.

Figure 6 demonstrates the packet exchange where the server requests the client to resend the HTTP request over TCP.

For heterogeneous content, this hybrid scheme presents another interesting option (bypassing the client completely): as the HTTP server receives HTTP requests (with the URL) without any initial state establishment, the HTTP request can be forwarded and served by another server (which masquerades as the first server in its response) without further client involvement.

For heterogeneous content, our hybrid scheme offers a new mechanism to redirect requests; this new mechanism is transparent to clients. When the HTTP server receives an HTTP request over UDP, the request can be forwarded and served by another server without further client involvement. The server that sends the HTTP response must masquerade as the first server, or the client would not accept the HTTP reply. It is interesting to see that we can forward a single HTTP request through a chain of servers before we find one that can respond to the client, and it would still be completely transparent to the clients.

The use of UDP makes this optimization possible without any kernel-level changes in the server operating systems. This scheme can be used to improve Web cache validation, robust anycasting, and transparent proxies.

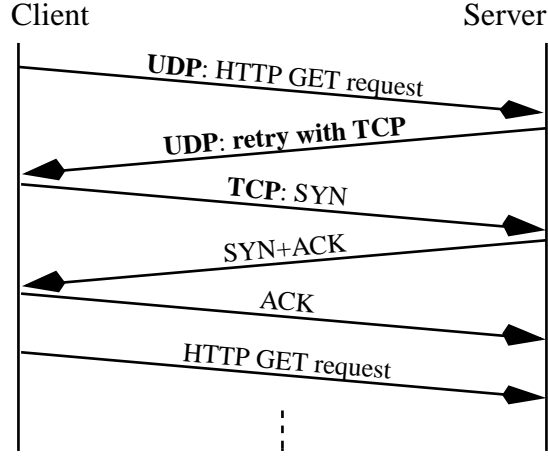


Figure 6: Explicit fall-back from UDP to TCP: Server is UDP capable, but the requested resource warrants use of TCP

Furthermore, we can use HTTP proxies to incrementally deploy the hybrid TCP-UDP scheme without modifying the installed client browsers.

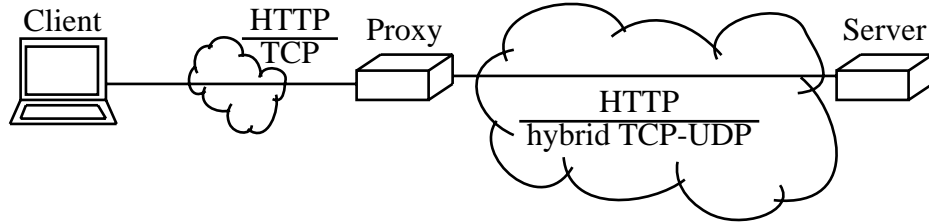


Figure 7: Incremental deployment of hybrid TCP-UDP scheme

The hybrid TCP-UDP scheme can be deployed incrementally by using proxy servers. Figure 7 illustrates a scenario where clients have not yet upgraded, but where the HTTP retrieval in the wide area takes advantage of our proposal.

Using UDP as the underlying transport protocol does not raise new security concerns. While it is relatively (compared to TCP) easier for intermediate routers to alter HTTP replies if they are sent over UDP, the same basic problem exists in TCP and is referred to as *TCP hijacking* [2, 6]. It can be solved using end-to-end data integrity security services, such as cryptographically signed digital signatures protecting the payload and control information of communications.

The same argument can be brought forward against the possibility of *smurf*-style denial of service attacks, where a third party causes many servers to reply to a single client by spoofing its source

address on the HTTP/UDP request message, causing the client to be overloaded. [16, §4] classifies a variety of solutions against this type of attack.

The hybrid TCP-UDP scheme improves the utility of HTTP for all three entities involved in HTTP transfers: clients, servers, and the network. The scheme achieves:

- reduced browsing latency for clients because latency introduced by the 3-way handshake for TCP connection establishment is avoided,
- reduced load for servers because fewer TCP connections are set-up, maintained, torn down, and because fewer packets are processed, and
- reduced network traffic because fewer TCP control packets are needed.

4 Performance Evaluation

In this section, we quantify the gains obtained by using the hybrid TCP-UDP scheme; our analysis concentrates on answering the following questions:

- How do the use of persistent HTTP and the choice of connection parameters (e.g., max keep-alive time, number of concurrent connections) affect the savings resulting from the use of the hybrid scheme?
- How do other network-related parameters, including network loss rates and MTU size, affect the savings?
- How are these savings then affected by various parameter choices for the hybrid TCP-UDP scheme, as described in Section 4.2? We are especially interested in the *max-udp* parameter. UDP is used only when the HTTP response is small enough to fit within *max-udp* packets.

Answering these questions, the following sections present the results from several simulations on our C++/Tcl based simulator. We performed several simulation experiments to answer the above-mentioned questions and we present here the results from these experiments. We ran these simulations on our own C++/Tcl-based simulator. The goal was to make the experiments realistic so that the results obtained can be transposed to implementations. For this reason, we decided against using synthetic workloads. Instead, we relied on HTTP trace data to drive our simulations. This trace data consists of 18 days' worth of HTTP traces gathered from U.C. Berkeley's Home IP service [4] and of traffic traces obtained within Sun Microsystems' internal network. The traffic for port 80 (HTTP) was recorded. All other protocols (or ports) were excluded from these traces. The traces [4] amounted to 9 million connections over 18 days, thereby providing us with about 500K HTTP transfers per day.

To get an initial feeling for the relevance of the approach we first derived some basic statistics from the traces. First, we derived the Cumulative Distribution Function (CDF) of the size of the replied data in the servers' response. A curve-fitting experiment indicated that a shifted exponential distribution of the type $1 - \exp[-.00035(x - 80)]$ (where x is the reply size in bytes, and 80 is the size

of the shortest reply) provides a very close approximation, in particular for response sizes smaller than 4 Kilobytes. These data are depicted in Figure 8. A noteworthy result of this distribution is that approximately 40% of the replies would fit into a single 1500 Byte packet. These results are in line with those reported in [7].

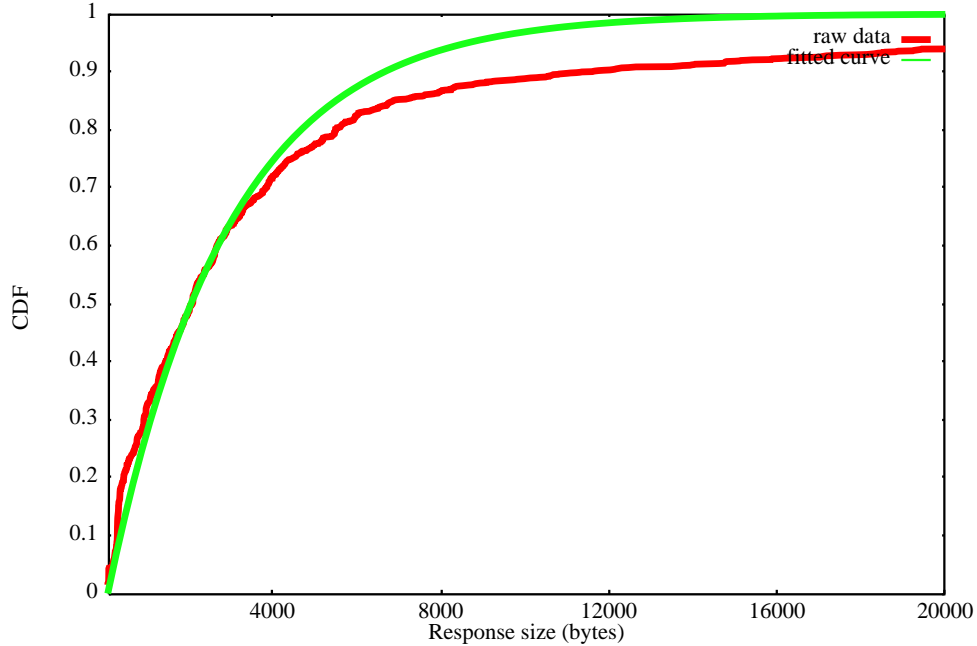


Figure 8: CDF of reply size

We also examined the statistics of HTTP transactions with respect to persistence. Figures 9 and 10 depict the histogram and CDF of the HTTP transactions that took place over the entire measurement period. Noteworthy here is the dominance of the occasional transaction. For example, 88% of the source destination pairs conducted 40 or fewer transactions over the entire period, and 98% conducted fewer than 100 transactions. Examining the inter-arrival times of these pairs reveals that these transactions were randomly distributed over the entire measurement period.

These results are very encouraging. They indicate that for a vast majority of the transactions, persistence would not help because the underlying TCP connection would not stay open long enough to allow its reuse. Therefore, most connections would have to be reestablished, and, given the distribution of the reply size, most connections would complete with a very small number of reply packets.

With these encouraging initial results, we conducted more elaborate experiments testing the approach under a variety of parameter changes as indicated above.

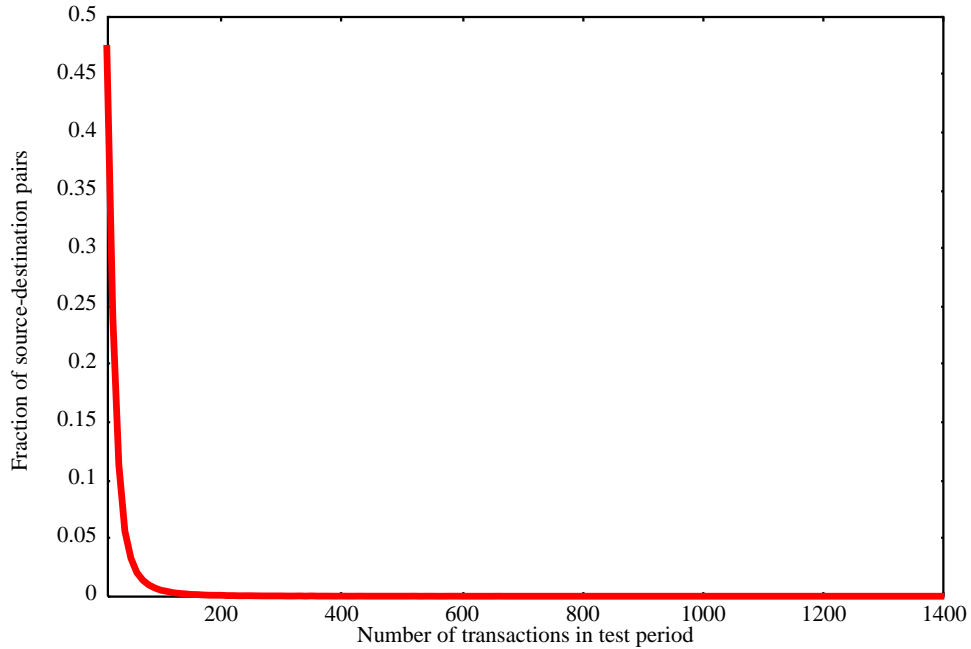


Figure 9: Histogram of transaction densities

4.1 Evaluation Metrics

The following metrics are used for the performance evaluation:

Number of bytes/packets transferred This metric captures the network load, as well as one aspect of the load on the servers and the clients.

Browsing latency This metric captures the performance as observed by the clients.

Number of connections set up This metric captures one aspect of the load on the servers (extra work required to set up and maintain TCP connection state, as well as extra cost in searching *Process Control Block* (PCB) lists).

The choice of our third metric, number of connections set up, was motivated by the following reasons. Our packet traces were HTTP logs. It was impossible for us to obtain the inter-packet traffic patterns that are needed for the first two metrics. Furthermore, the number of connections is the appropriate metric for evaluating HTTP redirection (as well as cache validation). These requests, and the corresponding responses, tend to be small enough to fit into the minimum-MTU IP packets.² Also, this metric is linked to the other two metrics. If each HTTP transfer requires

²The Internet Protocol requires the underlying network to support IP packets at least as large as 576 bytes [13, §3.1].

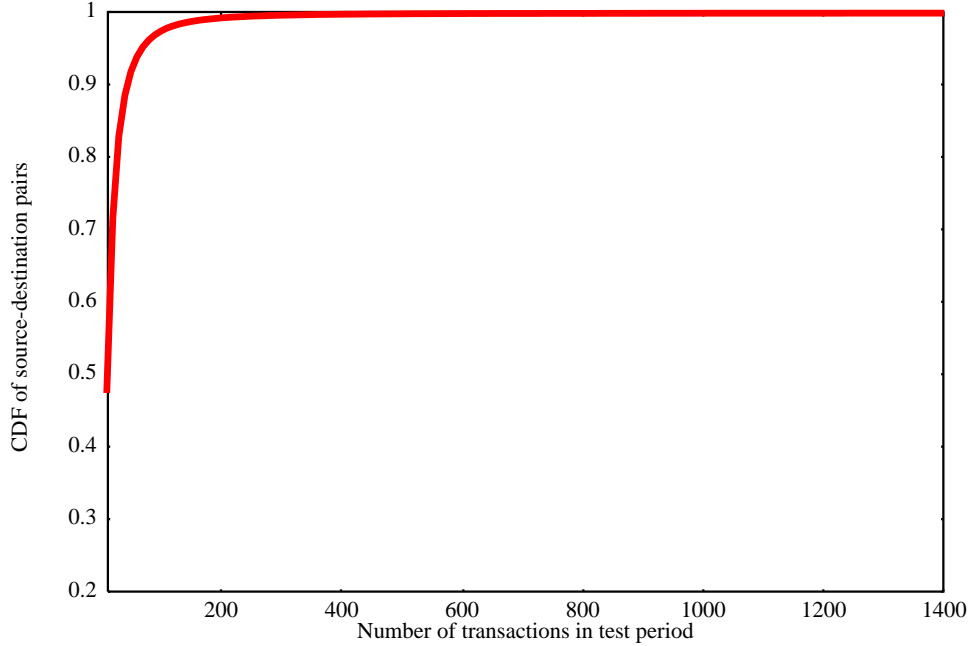


Figure 10: CDF of transaction densities

a new TCP connection, more bytes/packets will be seen on the network (connection setup and teardown packets) and clients will observe extra latency (due to connection setup delays as well as slow-start latency).

We convert this metric to a unitless ratio (fraction of connections saved) by performing each experiment twice. If an instance requires 100 distinct TCP connections under the current system, and it requires only 80 distinct TCP connections under the proposed hybrid TCP-UDP scheme, we obtain the resulting fraction of $(100 - 80)/100 = 20\%$.

4.2 Experiments

We performed many sets of experiments, each time varying one of four workload parameters:

Persist This parameter describes the fraction of requests from the clients that support persistent HTTP. The parameter value ranges from 0.0 to 100.0; a value of 70.0 implies that 70% of requests originate from clients that support persistent HTTP.

Loss-rate This parameter describes the fraction of packets that are lost in transit. The parameter value ranges from 0.0 to 1.0; a value of 0.10 implies that with 10% probability a packet will be dropped in the network. For these simulations, we assumed that packet losses are independent (not bursty, zero correlation). This is a pessimistic assumption because it overstates the

number of connections that will lose at least one packet, thereby diminishing the overall benefits of the hybrid TCP-UDP scheme.

Max-udp This parameter controls the server policy for choosing between use of UDP and TCP in the hybrid TCP-UDP scheme. A value of max-udp=4 implies the server will try to use UDP for all conversations where it can send the entire data in up to 4 packets. Otherwise, it directs the client to use TCP.

MTU This parameter describes the *Maximum Transmission Unit* (MTU) for packets in the network. An MTU size of 1460 implies that the network MTU allows for up to 1460 bytes payload (packet size minus TCP header, IP header, and link-level headers). We assume that the end hosts send MTU-sized packets whenever possible. The larger the MTU, the more likely it is that the hybrid TCP-UDP scheme would avoid setting up a TCP connection (we avoid a TCP connection when HTTP data size is less than MTU-max-udp, and the network does not lose any of these packets).

In our simulations, we included further parameters for controlling the behavior of persistent HTTP. A keep-alive parameter was set to 60.0, meaning that a persistent HTTP connection would be closed after 60 seconds of inactivity. Also, a max-connect parameter was set to 1024. Thus, the server will only support 1024 concurrently active connections. Inactive connections were closed according to a *Least Recently Used* (LRU) policy.

4.3 Results

The simulation results were reasonably similar for the various traffic traces. For all graphs in this Section, the x-axis quantifies the varied parameter (e.g., loss rate) and the y-axis quantifies the evaluation metric (e.g., fractions of connections saved, expressed as a percentage).

Loss Rate and Max-Udp

The graph in Figure 11 shows the effect of the network packet loss rate and the setting of max-udp parameter in the server on the overall performance of the hybrid TCP-UDP scheme. For this experiment, we assumed an MTU of 1460 bytes, as well as full persistence (a very conservative assumption), i.e., all clients support persistent HTTP, and all requests are multiplexed on existing TCP connections wherever possible. As the graph shows, with the very conservative policy of setting max-udp value to 1 (use UDP only if all data from server fits into one packet), the hybrid scheme gains as much as 18%-19% with the typical (around 1%) loss rate observed in many intranets, as well as on the Internet. Even with very high packet losses (around 10% loss rate), the hybrid scheme outperforms the traditional mode by 12%-13%. The graph also depicts the additional benefits that using a higher max-udp value would provide: up to 85% improvement over the current schemes. Even with a max-udp setting of 4 to 10, we can expect to see benefits of 35% to 66%, even under the assumption that persistent HTTP is used exclusively.

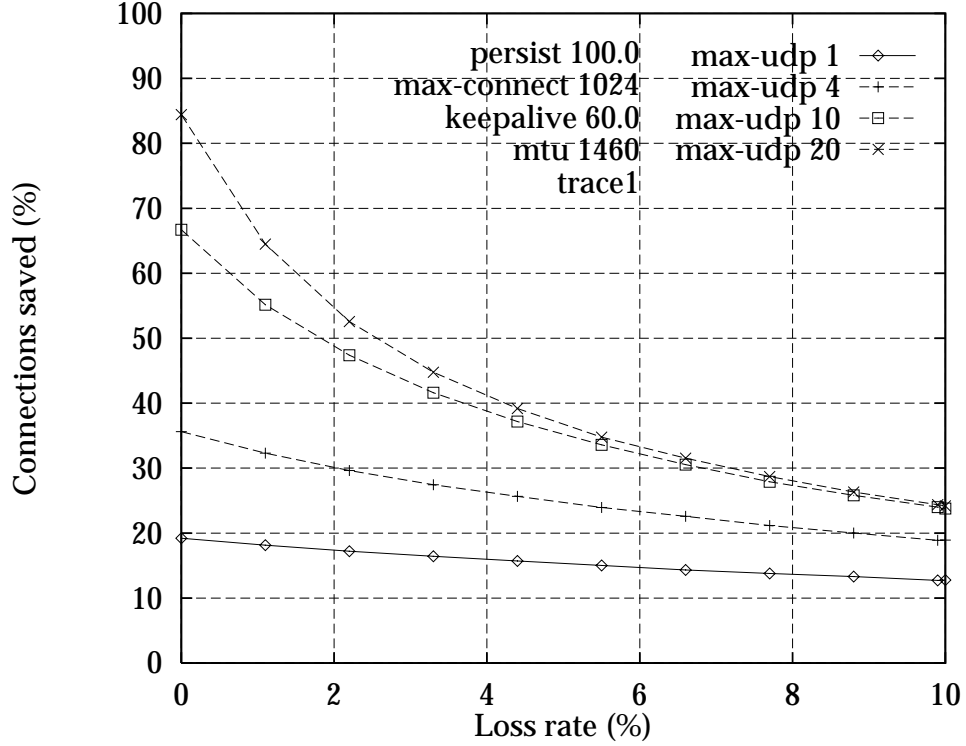


Figure 11: Effect of loss rate and max-udp parameter

Persistent HTTP and Loss Rate

The graphs in Figures 12 and 13 show the effect of persistent-HTTP and network packet loss rates on the overall performance of the hybrid TCP-UDP scheme. The max-udp parameter is set to the value 1 in Figure 12 and the value 4 in Figure 13; the MTU remains 1460 bytes. We performed this experiment to evaluate the performance in the presence of heterogeneous clients (some support persistent HTTP; others do not), as well as to see the effect of the max-udp parameter in this system. As both figures show, the performance is dominated by the presence of non-persistent clients. Even with 80% persistence (i.e., only 20% do not support persistence), the results are close to the performance with zero persistence. As expected, the results are much better for max-udp of 4 (as compared to max-udp value of 1). As Figure 12 shows, with max-udp set to 1 and with zero persistence, the hybrid scheme provides a gain of 25%-40% over the traditional schemes, depending on the overall packet loss rate. The gains decrease very slightly as the persistence parameter increases to well beyond 80%. Even in the presence of 90% to full persistence, the hybrid scheme provides performance gains of 15% to 30%, except when the packet loss rate goes up to 20% (i.e., pathological conditions). With the max-udp set to 4, the performance gains tend to be around 70% for moderate loss rates, decreasing to around 60% for 70%-80% persistence. Even with 10% packet

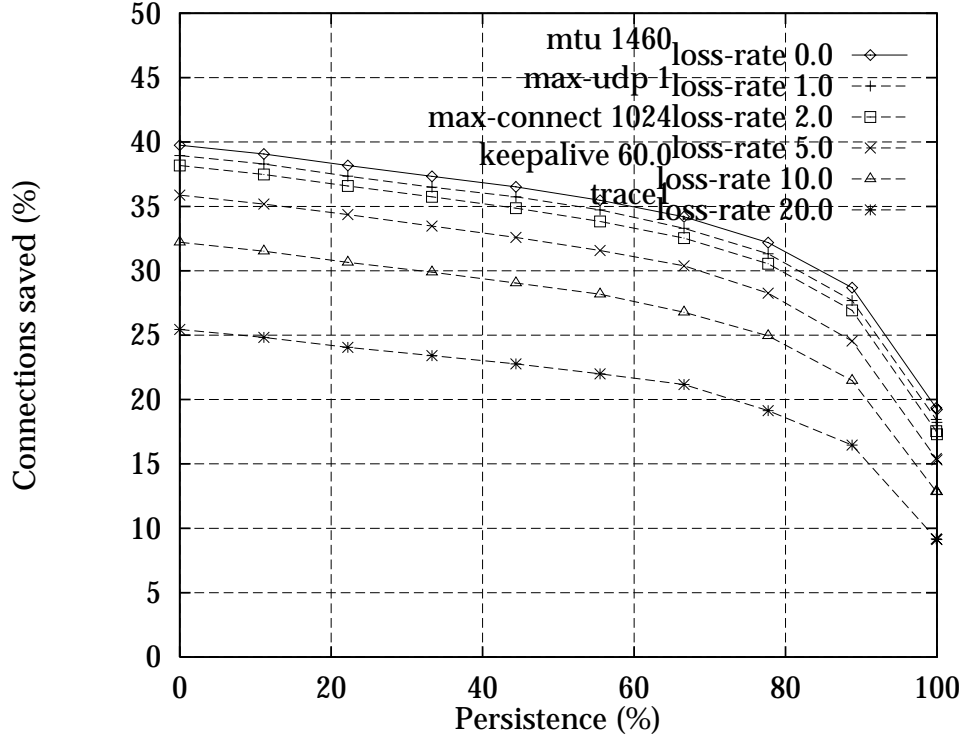


Figure 12: Persistence and loss rate (a)

loss rate, the hybrid scheme gains around 40% at 80% persistence.

Max-udp and Loss Rate

Figure 14 illustrates the effect of the max-udp parameter and the network packet loss rate values on the overall performance of the hybrid TCP-UDP scheme. We chose worst-case values for the MTU and the persistence parameters: the MTU parameter is set to the value 460 which reflects the lower MTU settings that some people use over 28.8K modems and other low-speed links. Most of the Internet links, as well as higher-speed links to end-users, however, use higher MTU values (typically around 1.5 Kilobytes)[8]. We also assume full persistence (i.e., all HTTP transfers are multiplexed on existing TCP connections whenever possible). We performed this experiment to evaluate the effect of max-udp parameter on the system performance, with varying loss rates. As the graph shows, the performance gains due to the hybrid TCP-UDP scheme increase steadily with increasing value(s) of the max-udp parameters, and are slightly reduced under reasonable loss (around 1% packet loss rate). Even in the presence of relatively high loss rates (about 5%), we observe 15% gains with the max-udp equal to 3. This result is especially interesting in the context of recent research in increasing the TCP initial window (to 4 packets or 4Kb, whichever is lower).

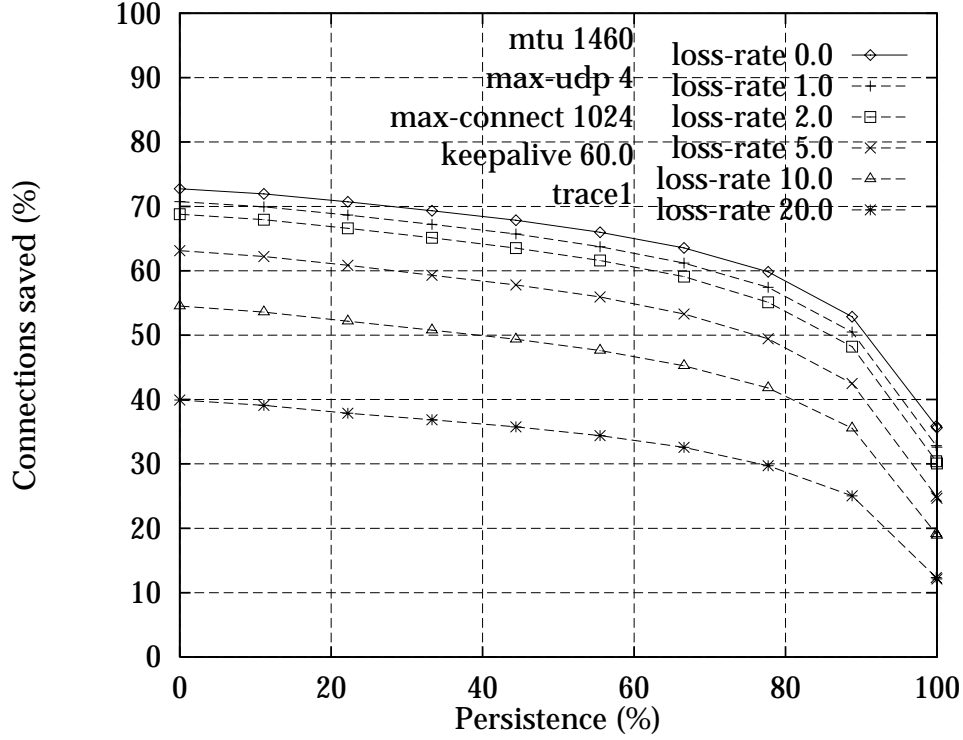


Figure 13: Persistence and loss rate (b)

MTU and Loss Rate

The graphs in Figure 15 and Figure 16 show the effect of the network MTU values and packet loss rates on the overall performance of the hybrid TCP-UDP scheme. The max-udp parameter is set to the value 4 in Figure 15 and the value 1 in Figure 16; the persistence parameter is set to the value 70% in Figure 15 and the value 100% (full persistence) in Figure 16. We performed this experiment to evaluate the performance, for varying MTU values and loss rates, in the presence of heterogeneous clients (some support persistent HTTP; others do not), as well as to see the effect of the max-udp parameter in this system. As both figures show, higher MTU values lead to increased performance gains due to the hybrid TCP-UDP scheme; these gains are reduced somewhat due to the packet losses in the network, but the gains remain high even with significant (around 2%) packet loss rates. As Figure 15 shows, the gains are much higher with heterogeneous clients (70% persistence); setting max-udp to 4 results improves the performance gains, even in the presence of pathological network behavior (10% to 20% packet loss rate).

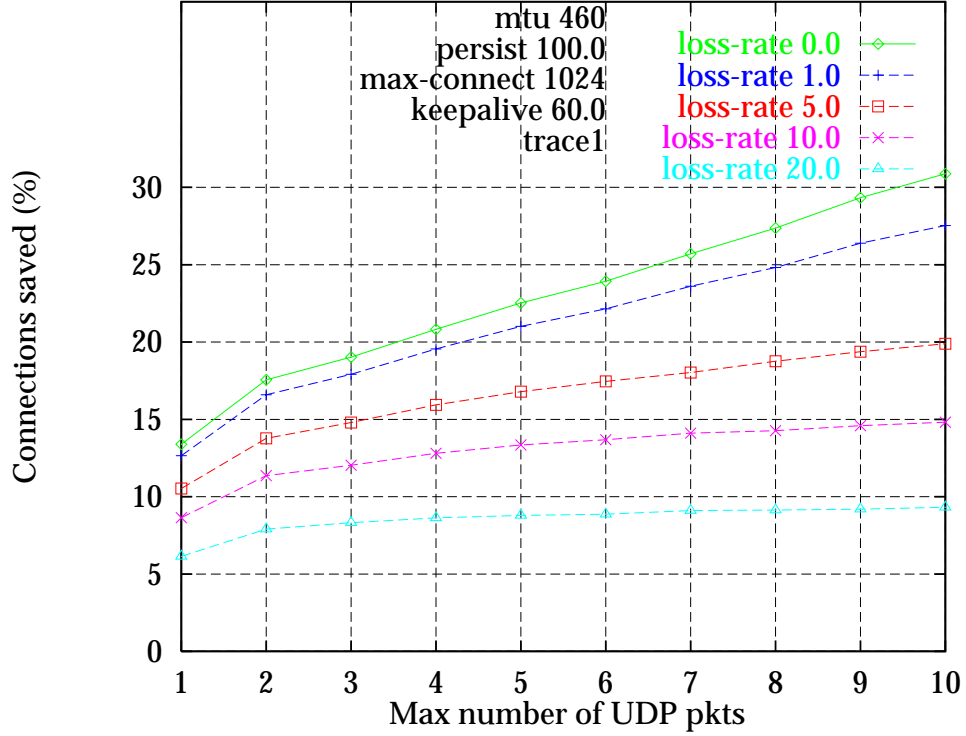


Figure 14: Max-udp and loss rate

4.4 Summary of Simulation Results

The hybrid TCP-UDP scheme provides significant performance gains over the traditional use of TCP as the only transport protocol for HTTP traffic. We observed performance gains exceeding 20-25% with persistent HTTP clients, and over 40-50% with clients without persistent HTTP.

In heterogeneous environments, the system performance was dominated by the presence of non-persistent clients. Even with 70% to 80% persistence, the performance gains of the hybrid scheme were close to that of a system with zero persistence.

The hybrid TCP-UDP scheme's performance gains were reduced somewhat by network packet losses, but the gains remained significant even under pathological network behavior (10% to 20% packet loss rate).

It is beneficial to try sending more than 1 packet via UDP, especially if the network MTU is small (around 500 bytes). However, we must appropriately consider and trade-off the increased likelihood of network congestion due to the increase in the max-udp parameter.

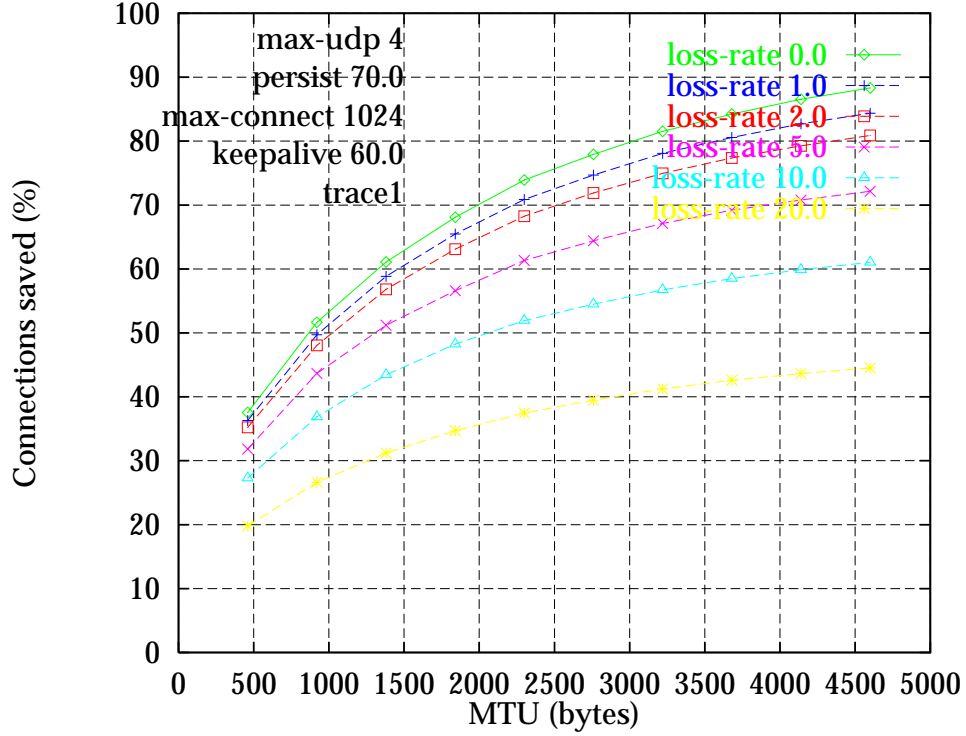


Figure 15: Maximum Transmission Unit (MTU) and loss rate (a)

5 Conclusions and Outlook

This paper introduced and analyzed a hybrid TCP-UDP transport layer scheme for HTTP. The hybrid TCP-UDP scheme combines the low cost of using UDP with the high reliability and congestion control features of TCP. It benefits from the low overhead of using UDP for short transfers and, for large transfers, it is able to use TCP for reliable delivery and good congestion behavior. Our simulation experiments verified these gains. We observed performance gains exceeding 25% with persistent HTTP clients, and over 50% for clients without persistent HTTP, even in the presence of high network packet losses. The performance gains for mixed environments (i.e., clients using persistent HTTP, as well as traditional HTTP) were similar to those of a system where no clients used persistent HTTP. The hybrid scheme is attractive because it only requires application-level changes, while the operating system kernel code can remain unchanged. Finally, the scheme is incrementally deployable in the current Internet and it is fully compatible with the deployed base.

We are currently exploring some promising optimizations to further improve system performance. First, the clients can use some heuristics to guess (on a per-transfer or per-session basis) if they can use UDP for successful transmission. For example, the clients can keep track of whether the server supports the hybrid scheme at all, or predict the file size based on the size of the (expired)

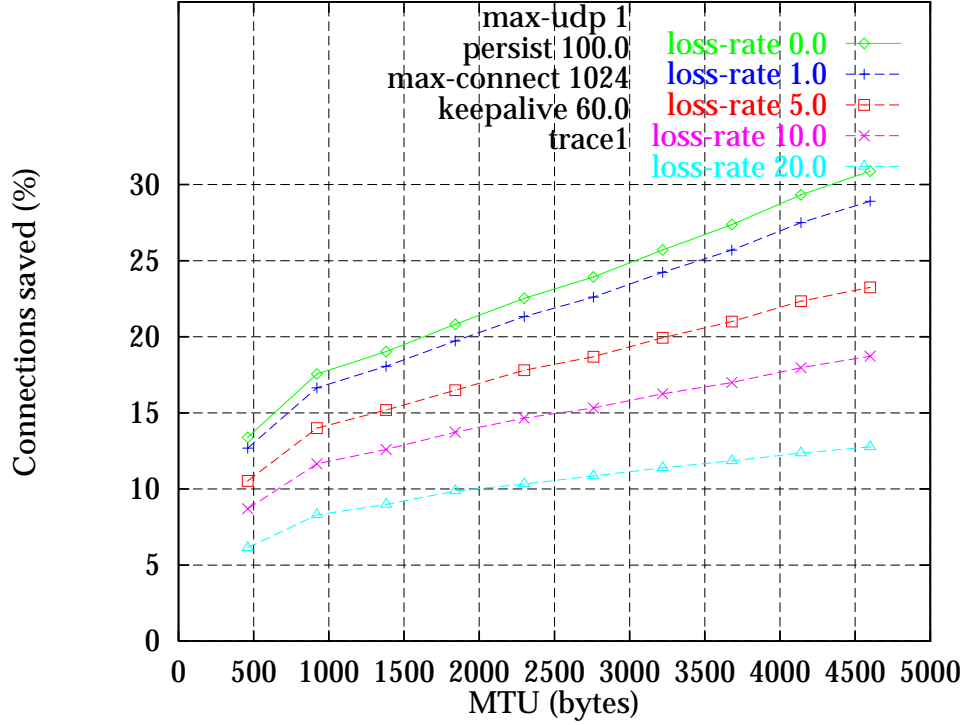


Figure 16: Maximum Transmission Unit (MTU) and loss rate (b)

cached copy of a previously retrieved version, or assume that the Postscript and PDF files tend to be large, while README files tend to be small. Second, the servers can adapt the max-udp parameter based on the observed network loss rates. Third, the clients can avoid redundant data retransmissions by using sub-ranges appropriately.

We also expect that further performance gains can be observed, if recent trends in Web traffic continue. The increasing use of style sheets will lead to smaller HTTP transfers, and so will the increasing use of cache validation and HTTP redirection. The performance gains can be further increased by setting the server policy to use UDP more aggressively (for example, when the data can fit into 4 packets, analogous to some recent proposals to increase the initial TCP window size[11]).

Acknowledgments

We are grateful to Geoffrey Baehr, Charles Perkins, Erik Guttman, and Abhishek Chauhan for their insightful comments on the drafts of this paper.

References

- [1] Bob Braden. *RFC-1644 T/TCP - TCP Extensions for Transactions*. Network Working Group, July 1994.
- [2] CERT. *IP Spoofing Attacks and Hijacked Terminal Connections, CA-95:01*. Computer Emergency Response Team, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1995.
- [3] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. *RFC-2068 Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group, January 1997.
- [4] Steven D. Gribble. UC Berkeley Home IP HTTP traces. July 1997. Available at <http://www.acm.org/sigcomm/ITA/>.
- [5] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the Performance of HTTP Over Several Transport Protocols. *IEEE/ACM Transactions on Networking*, 5(5):616–630, October 1997.
- [6] Laurent Joncheray. A Simple Active Attack Against TCP. In *Proceedings of the 5th UNIX Security Symposium*, pages 7–19, Salt Lake City, Utah, June 1995. USENIX.
- [7] Bruce Mah. An Empirical Model of HTTP Network Traffic. In *Proceedings of Infocom’97*, pages 593–600, Kobe, Japan, April 1997.
- [8] Jeffrey Mogul and Steve Deering. *RFC-1191 Path MTU Discovery*. SRI International, Menlo Park, CA, November 1990.
- [9] B. Clifford Neuman. *The Virtual System Model: A Scalable Approach to Organizing Large Systems*. PhD thesis, University of Seattle, Washington, 1992.
- [10] Venkata N. Padmanabhan. Private communication, 1997.
- [11] Kedarnath Poduri and Kathleen Nichols. *RFC-2415 Simulation studies of increased initial TCP window size*. Network Working Group, September 1998.
- [12] Jon Postel, editor. *RFC-768 User Datagram Protocol*. Network Information Center, August 1980.
- [13] Jon Postel, editor. *RFC-791 Internet Protocol*. Information Science Institute, University of Southern California, September 1981.
- [14] Jon Postel, editor. *RFC-792 Internet Control Message Protocol*. Information Sciences Institute, University of Southern California, September 1981.
- [15] Jon Postel, editor. *RFC-793 Transmission Control Protocol*. Information Sciences Institute, University of Southern California, September 1981.
- [16] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a Denial of Service Attack on TCP. In *Proceedings of the Symposium on Security and Privacy*, pages 208–223, Oakland, California, May 1997. IEEE.

Distributed Tutored Video Instruction: Experiments Comparing Face-to-Face with Virtual Collaborative Learning

Randall B. Smith, Michael J. Sipusic, Robert L. Pannoni

Introduction by Randall B. Smith

This project was a vision of former Sun Labs Director Bert Sutherland. He had been impressed by the "Tutored Video Instruction" (TVI) studies of his longtime friend Jim Gibbons, a Stanford professor and, later, Dean of Stanford's School of Engineering. Gibbons showed that students can achieve higher grades if, rather than attending lecture, they study a videotape of the lecture with a small group of fellow students and a facilitator (or "tutor"). This perhaps surprising result suggests that physical and temporal co-presence with the lecturer is not required for student learning. In fact, perhaps the lecturer's presence interferes: after all, students can pause a videotape and discuss the material in a less intimidating setting than is afforded by the conventional lecture hall.

Bert Sutherland wondered if perhaps the entire TVI group could be "virtualized" — spread out across a network. A distributed group could communicate via networked video and audio, and the lecture videotape stream could be sent over the net as well. Would "Distributed Tutored Video Instruction" (DTVI) be as effective as TVI?

The resulting comparative study is to our knowledge the largest effort ever to compare virtual with co-present small group collaborative learning. Dozens of faculty and researchers at Sun, SERA learning technologies, and two universities were involved in carrying out this series of experiments, which involved over 1000 students.

To begin the tests, John Dutra fabricated a kind of "closed circuit TV" scheme that enabled nine video images to be composited onto each computer screen. While not truly "networked" video and audio (there was no IP traffic involved), the technology enabled a reasonable simulation of the distributed study group setting, and preliminary experiments were underway. Later, Randy Smith brought a screen-sharing technology called "Kansas" to the task. This enabled true IP networked video and audio, as well as shared tools for collaborative note taking, shared web publishing, instructor comments and even remote session debugging (researchers at Sun could "visit" the sessions at remote universities to help fix technical glitches).

The results are encouraging for our networked future. Both TVI and DTVI outperformed conventional lecture as measured by student course grades. Furthermore, the TVI and DTVI grades were statistically identical. The collaborative learning boost apparently does survive a group's transition to video.

PUBLICATIONS:

Sipusic, M.J., Pannoni, R.L., Smith, R.B., Dutra, J., Gibbons, J.F., and Sutherland, W.R., "Virtual Collaborative Learning: A Comparison between Face-to-Face Tutored Video Instruction (TVI) and Distributed Tutored Video instruction (DTVl)," Sun Microsystems Laboratories Technical Report SMLI-TR-99-72.

<http://www.sun.com/research/techrep/1999/abstract-72.html>

Smith, R.B., Sipusic, M.J., and Pannoni, R.L., "Experiments Comparing Face-to-Face with Virtual Collaborative Learning," Proceedings of Conference on Computer Support for Collaborative Learning 1999, Stanford University, Palo Alto, December 1999. pp 558-566.

http://sll.stanford.edu/CSCL99/papers/tuesday/Randall_Smith_558.pdf

<http://kn.cilt.org/cscl99/A68/A68.HTM>

Virtual Collaborative Learning*

A Comparison between Face-to-Face Tutored Video Instruction (TVI) and Distributed Tutored Video Instruction (DTVI)

Michael J. Sipusic[†], Robert L. Pannoni[†], Randall B. Smith^{**},
John Dutra^{**}, James F. Gibbons[†], and William R. Sutherland^{**}

SMLI TR-99-72

January 1999

Abstract:

Tutored Video Instruction (TVI) is a collaborative learning methodology in which a small group of students studies a videotape of a lecture. We constructed a fully virtual version of TVI called Distributed Tutored Video Instruction (DTVI), in which each student has a networked computer with audio microphone-headset and video camera to support communication within the group. In this report, we compare survey questionnaires, observations of student interactions, and grade outcomes for students in the face-to-face TVI condition with those of students in the DTVI condition. Our analysis also includes comparisons with students in the original lecture. This two and a half year study involved approximately 700 students at two universities.

Despite finding a few statistically significant process differences between TVI and DTVI, the interactions were for the most part quite similar. Course grade outcomes for TVI and DTVI were indistinguishable, and these collaborative conditions proved better than lecture. We conclude that this kind of highly interactive virtual collaboration can be an effective way to learn.

*The full Sun Labs Technical Report is available online and on the accompanying CD. Because the report is 90 pages long, we instead include the following paper originally presented at the 1999 Computer Supported Cooperative Learning Conference. This paper summarizes the technical report, and includes further analysis and follow-up studies.

**Sun Microsystems Laboratories

[†]SERA Learning Technologies
2570 West El Camino Real, Suite 300
Mountain View, CA 94040



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email addresses:

sipusic@socrates.berkeley.edu
pannoni@sera.com
randall.smith@sun.com
john.dutra@sun.com
gibbons@ee.stanford.edu
bert.sutherland@sun.com

Experiments Comparing Face-to-Face with Virtual Collaborative Learning

Randall B. Smith

Sun Microsystems Laboratories

Michael J. Sipusic, Robert L. Pannoni

SERA Learning Technologies

Abstract: We report on set of studies conducted over two years involving over 1000 students at two universities. The main study compares three conditions: conventional classroom lecture, a face-to-face collaborative learning technique called Tutored Video Instruction (TVI), and the virtual-world counterpart of TVI, Distributed Tutored Video Instruction (DTVI). The main study involved over 700 students in 6 courses. When using final course grade as an outcome measure, it has been previously established that TVI students outperform lecture students. Therefore the comparison of interest for us is between DTVI and TVI: would the audio and video technology used to support a distributed group enable DTVI students to achieve the higher grades attainable in TVI? We found no statistical difference between the grades of the DTVI and TVI students, and both groups outperformed the lecture students.

We also summarize extensive interaction process data and survey data, then report on some more informal studies assessing the usability and effectiveness of an "Enhanced DTVI" system, in which distributed students can not only see and talk over digital networked media, but can take notes together in real time.

Introduction

One-by-one, the technological and cost barriers to high-bandwidth networking are falling. The increasing bandwidth availability inspires visions of a new world where people can work collaboratively without being held hostage to physical proximity. But lost in the frenetic rush toward virtualization is the possibility that there will be an additional cost to the substitution of virtual communication for face-to-face communication. While networks have proven quite adept as a medium for asynchronous and broadcast forms of communication, highly interactive, real-time, multi-way communication has proven more problematic (Sellen 1992).

Problems with technology-mediated communication would seem to be particularly significant in the domain of distance learning. Distance learning is likely to be among the first large-scale uses of high-bandwidth networking technology, because of the obvious savings in time and travel costs. The simplest way to apply this technology is to recreate the familiar classroom lecture environment. In fact for many, the broadcast of standard lecture courses is synonymous with "distance learning." There is, however, an overwhelming body of educational research showing that instructional methods that foster interpersonal discourse and the social construction of knowledge are more effective than methods that rely on the broadcast of information (Cohen 1994). Therefore the rush to virtualize standard lectures may in fact be a retreat toward outdated and less-effective instructional methods.

We attempt to "raise the bar" for distance learning by moving from a classroom transmission metaphor to a collaborative learning metaphor. Collaborative learning techniques have been shown to be consistently superior to traditional classroom lecture both in effectiveness and student satisfaction (Cohen 1994; Johnson and Johnson 1994). However, collaborative learning is highly dependent on communication or *discourse*. Past research shows communication breakdowns in video-mediated settings. We designed this research project to find out whether video-mediated communication could support the rich social discourse required for collaborative learning.

The collaborative learning method chosen for this study is Tutored Video Instruction (TVI). TVI was invented at Stanford University over twenty years ago. In TVI, a small group of students play a pre-recorded videotape of a classroom lecture. During the playing of the tape, a facilitator encourages the students to pause the tape to ask questions or discuss topics. As with other forms of collaborative learning, TVI students have been shown to outperform students who physically attended the lectures (Gibbons et. al., 1977).

Duplicating the success of TVI in a video-mediated environment would seem to be a significant challenge. Like other forms of collaborative learning, TVI groups exhibit frequent and complex social interaction among group members. Students in TVI use verbal and non-verbal communication to negotiate meaning and relationships with group members. Video has been shown to be much less important than audio in broadcast learning environments, but because of the need to support non-verbal communication, the visual channel may play a crucial communication role in TVI. The purpose of this study is to determine whether these rich communication patterns and relationships can be created in a video-mediated environment specifically designed to mimic face-to-face TVI. The virtual version of TVI created for this experiment was called "Distributed Tutored Video Instruction," or DTVI.

Over 700 hundred students from 6 courses and 2 universities were involved in the main experiment. Details of the protocols, procedures, and statistical analysis used in this experiment can be found in a technical report from Sun Laboratories (Sipusic et. al., 1999), though we summarize the main findings here.

We also discuss our experiences with a follow-on system called "Enhanced DTVI." Audio and video conferencing are only the beginning of what computer networks can do, and the Enhanced DTVI system, built in the Kansas shared application construction environment (Smith et. al. 1998, Smith et. al. 1997, Maloney & Smith 1995), is designed to investigate how we might better utilize the technology. In Enhanced DTVI, students watch a tape and interact as they would under normal DTVI, but they can collaboratively take notes as the lecture plays. The notes, which form a hierarchical outline, automatically appear as web pages so that the group members, the instructor, and even students from other groups can view the notes later. Our sample size was too small for significant statistical analysis in this part of our study, but we found the system well-received, and were able to demonstrate that shared note taking among up to 7 students works well. Because the notes form a shared task whose outcome is of potential value to the students, collaboration might be elevated in systems like Enhanced DTVI.

Video-Mediated Communication

Given that the collaborative learning effect is built on content and group-oriented discourse, the ability of technology to support all of the subtle aspects of human communication is a serious

issue for collaborative distance learning. Recent studies of collaborative work done at a distance over a video link have found that there are subtle and unexpected communication difficulties encountered by participants attempting to coordinate their efforts through these systems. Isaacs and Tang (1993) reported that participants using the prototype *ShowMe*TM from Sun Microsystems had difficulty in the real-time management of a number of social behaviors that support conversations, such as turn-taking and the coordination of joint attention through eye-gaze. For many of the same reasons, Fish, Kraut, Root, and Rice (1992) in an evaluation of AT&T's video telephone, *Cruiser*, state "users said they used face-to-face communication rather than the Cruiser system because the Cruiser system was not able to support all the communication demands of conventional work activities."

Communication breakdowns resulting from technology-mediated communication can typically be attributed to three causes:

Network transmission artifacts. Low frame rate, audio and video latency, and artifacts from information compression have been shown to affect people's preference for the media, as well as how they interact through it (Kies et. al., 1996, Isaacs & Tang, 1993). We did not want to position this work so that it might give a negative result valid only for the few years it will take for these transmission artifacts to become a negligible factor. Consequently, in our main DTVI experiment, we used only analog video and audio links so that latency was negligible and image quality was relatively high. In our Enhanced DTVI experiment, we used the purely digital test bed provided by the Kansas system. Even here, the audio quality was high (16 bits at 22 kHz), the latency was barely noticeable, and the video image quality, though perceptibly lower than the analog system, was still quite good (the digital version averaged about 6 Mb per sec of multicast data for the 9 audio and video streams).

Reduced image size. Image size may adversely effect conversation patterns (Monk & Watts, 1995) and the willingness of individuals to interact with (Fish et. al., 1990) or trust (Rocco 1998) remote collaborators. Storck and Sproull (1995) found that students who view their peers through video give them lower competency ratings than they give their co-present peers.

Diminished directionality cues from eye gaze. Difficulty in establishing the location of participants' gaze has been linked to various perturbations in the communication process. These, include exaggerated gestures and staring (O'Conaill et. al., 1993; Colston & Shiano 1995), more formal speech with fewer overlapping utterances (Monk & Watts 1995; Gale 1998; Cohen 1994; Sellen 1992) and difficulty in managing group speech issues such as turn-taking and maintaining or acquiring the floor (Isaacs & Tang 1993).

To the extent that the DTVI system reproduces these problems, we would expect that the collaborative learning processes of DTVI groups would be compromised when compared to their face-to-face TVI equivalents.

The DTVI system

The DTVI prototype was designed to simulate a face-to-face TVI session as closely as possible. The system uses an open microphone architecture in which anyone can speak and be heard at anytime. Students wear stereo headphones with a built-in microphone attached. The group discussion plays in one ear while audio from the course videotape plays in the other. Students can adjust the volume of the two channels independently. They can also mute the microphone.

Analog video distribution was chosen for the prototype to achieve full 30-frames per second (TV quality) video with no compression artifacts or perceptible latency. This idealized environment allowed us to study the learning process without worrying about the effect of digital transmission artifacts. We compared the analog version with a purely digital networked equivalent version, and will discuss these findings in the section on Enhanced DTVI.

Real-time video of each participant is delivered from a camera positioned beside the monitor. These images are arrayed in individual cells of a 3x3 "Brady Bunch" matrix (see Figure 1). For the main DTVI experiment, the matrix displays as an analog video overlay on a 20-inch computer screen. Videotape of the course lecture plays in the lower right cell. Because of the 3x3 matrix limitation, a maximum of seven students (and one tutor) can participate in a DTVI session. Individual cells of the matrix cannot be resized, but students can alternate between watching all nine cells or exploding a single cell to take up the full window. The video window itself is also fully movable and resizable.

Figure 1. The DTVI system presents each user with this "Brady Bunch" view of the group. A videotape image of the lecture is at lower right, and the facilitator (lower center) and up to 7 students participate



The tutor controls the VCR that plays the course video. Students can verbally request that the tutor pause the tape or can send a pause request message to the tutor by pressing a button on the user interface. (The verbal method was by far more common). Later versions of the DTVI software allow students to send private text messages to the tutor. This feature was provided primarily as a way for students to notify the tutor of technical problems when the microphone wasn't functioning correctly. We resisted the temptation to add additional features to the software because we wanted to maintain the direct comparison between TVI and DTVI.

Academic Performance: Lecture vs. TVI vs. DTVI

California State Polytechnic University at San Luis Obispo (Cal Poly) and California State University at Chico were the two institutions in the study. There were six courses used in the experiment. The courses are described briefly here.

Course Title	Symbol	Location	Number of students		
			Lecture / TVI / DTVI		
			Lecture	TVI	DTVI
Architecture, Materials of Construction	ARCH	Cal Poly	34	17	15
Media Esthetics, introduction to recognition and interpretation of media production.	CDES 40	Chico	134	25	26
Telecommunications Industry, covers the technological, historical, and legal development of electronic media in America	CDES 65	Chico	97	37	32
Materials Engineering, introduction to the physical properties of materials.	MATE	Cal Poly	30	45	42
Business Management, graduate level course in the MBA program.	MGT	Cal Poly	0	20	18
Management Information Systems, introduction to the application of computer systems to businesses.	MIS	Cal Poly	24	63	57
Total			319	207	190

To assess academic performance, we converted final letter grades for the course into numbers based on a 4.0 grading scale. We then ran an analysis of variance (ANOVA) using course and method as experimental factors and incorporating overall GPA (as measured at the start of the course) as a covariant. As the table below shows, the mean grade for TVI students and DTVI students is nearly identical while the mean grade for lecture students is lower.

Estimates with GRADE as Dependent Variable

Method	Mean Grade	Std. Error
Lecture	2.83	.052
TVI	3.14	.057
DTVI	3.13	.060

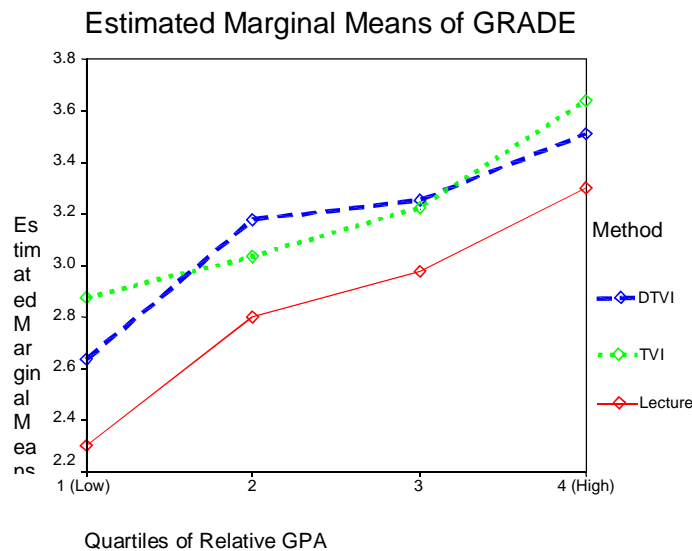
A pairwise comparison shows that TVI and DTVI means are not significantly different from each other at the .05 level ($p=.861$) and that both the TVI and DTVI mean are significantly different from the lecture mean ($p=.000$). Virtualizing TVI does not appear to lessen the collaborative learning effect, at least as measured by course grade.

Pairwise Comparisons based on estimated marginal means

Method I	Method J	Mean Difference (I-J)	Significance	95% Confidence Interval	
				Lower Bound	Upper Bound
Lecture	TVI	-.305 *	.00	-.491	-.120
	DTVI	-.291 *	.00	-.481	-.101
TVI	DTVI	.0144	.861	-.183	.212
	Lecture	.305 *	.00	.120	.491
DTVI	TVI	-.0144	.861	-.212	.183
	Lecture	.291 *	.00	.101	.481

* Significant at the .05 level

Besides looking at grades by course, we looked at whether students of varying academic ability responded differently to TVI and DTVI. We sorted students into quartiles based on their GPA relative to other students in the same course. The chart below shows the mean grades for students in each quartile:



For both TVI and DTVI students, the collaborative learning effect held for students of all levels of academic ability.

Other observations

In addition to student grades, grade point averages, and SAT scores, our main DTVI / TVI comparison data includes a catalogue of over 3000 conversational exchanges, hundreds of survey questionnaires, and several hundred hours of videotaped sessions. The survey questionnaires, the analytic framework for the conversational exchanges, and the statistical treatment of these data are detailed in the Sun Laboratories technical report (Sipusic et. al., 1999). We used this data to search for evidence of process differences between the face-to-face and DTVI conditions that might illuminate the impact of the technology on collaborative learning, and summarize some of the findings here.

Among other things, survey questionnaires assessed student satisfaction. Both TVI and DTVI students reported higher levels of satisfaction with collaborative learning than with traditional course methods. There was no difference in self-reported learning or likelihood of recommending the method to others. However, the TVI students reported enjoying the process slightly more and were slightly more likely to say they would take another course using this method.

We hypothesized that the DTVI technology would influence discourse patterns. However, based on statistical analysis of the conversational exchanges, we found there was a remarkable similarity in discourse patterns between TVI and DTVI. Where we were able to detect small differences, they were in the opposite direction of our original hypothesis. Students in the TVI and DTVI conditions spent the same total amount of time in conversation, but DTVI groups had more conversational turns than their TVI counterparts. We described them as "chattier." DTVI groups also reported higher amounts of humor. Most of this increased interaction occurred in conversation topics outside the content domain. When limiting our analysis to exchanges with positive ratings for content, the only statistically significant difference was an increase in tutor questions in the DTVI groups.

We hypothesized that the technologically mediated interactions would cause the DTVI students to exhibit lower levels of group cohesion. But explicit measures of group cohesion showed no difference between TVI and DTVI groups. For instance, there was no difference in tutors' average group cohesion ratings of conversational exchanges. Nor did students report differences in cohesion in survey questions.

All in all, we conclude that the communication perturbations of video-mediated communication using DTVI are both less evident and less salient than might have been expected based on past research. Whatever "noise" DTVI added to the communication process was subtle enough to not be explicitly recognized by students and to have minimal impact on social discourse. But some process differences did show up in student behavior and attitudes. For instance, while DTVI students reported feeling as connected to their group as did TVI students, they also admitted that if they were looking for emotional energy from their peers, face-to-face collaboration would be a superior environment. DTVI students showed signs of a kind of disengagement, rating both facilitators' content knowledge and course difficulty lower than their TVI counterparts. Finally, DTVI groups showed evidence of a weakening of the discursive reciprocity covenant (one's obligation to sustain and attend to conversation). This may be because of factors such as gaze ambiguity in the DTVI condition. The impact of this reduced social presence and weakening of the reciprocity covenant is mixed. Lower reciprocity may have made it easier for DTVI students to hide or do other work instead of attending to the discussion. But it may also have made it easier for them to publicly take intellectual risks in front of their peers.

Enhanced DTVI

Towards the end of our main two-year experiment, our preliminary study of student grades made it clear that DTVI and TVI would probably be comparable and both would do better than lecture. Since computers can do more than ship video and audio among students, we naturally began to wonder how we might enhance the DTVI interface by supporting the students in other ways. The DTVI system can be immediately applied to many different courses in contrast to

most educational software systems requiring specially authored software content for each course. We decided to build a generic shared note taking tool (Figure 2, 3), so that we could retain this advantage of generality.

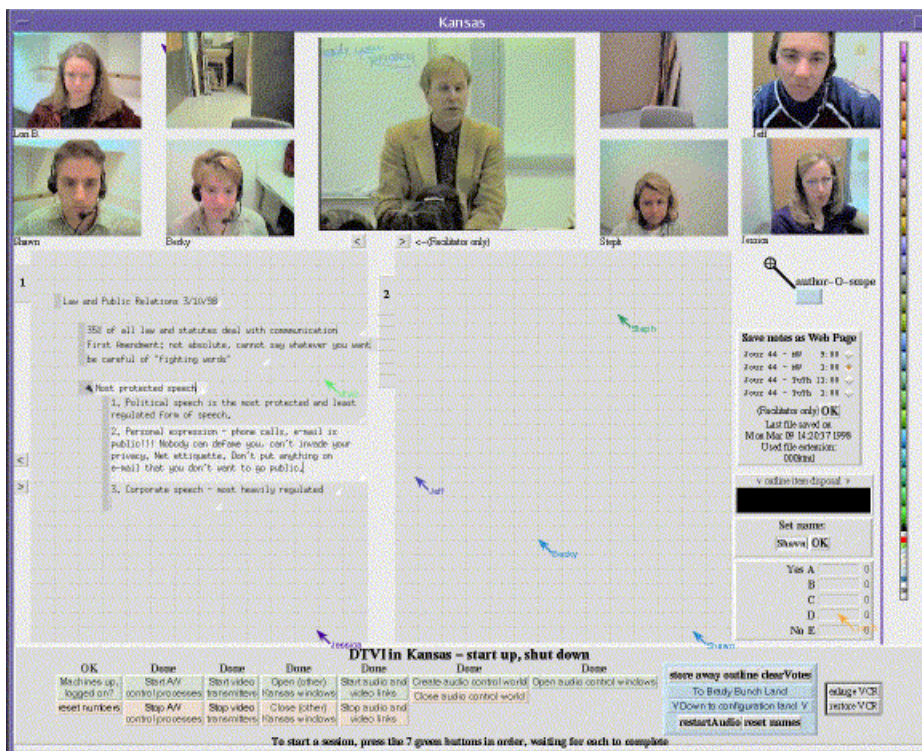


Figure 2. The Enhanced DTVI interface. In addition to the audio and video links among participants, the enhanced version includes a shared outlining tool (shown in close up in Figure 3) with which students can take notes and publish them to the web. The video windows for the students and facilitator are smaller than their DTVI counterparts, though the videotape window is comparable in size.

We used the Kansas system (Smith et. al., 1998; Smith et. al., 1997; Maloney & Smith, 1995), a toolkit for building shared 2-D worlds, to create the outlining facility. Both video and audio in Kansas are digital multicast technologies, whereas the main experiment used analog audio and video. (Students in these more informal digital studies were not part of the main experiment discussed above.) Students initially complained about the tendency for the digital system to crash, but with continued improvements to the system, we were able to get it to run continuously for weeks and sometimes months without problems. Due to our use of compression, the digital image quality was slightly lower than that of the analog system, but we believe such differences are small compared to the differences between a virtual system and co-present collaboration.

The outlining tool lets students take notes by typing onto a gridded page. The resulting text box can be manipulated to create a hierarchical outline. At the end of the session, the facilitator presses a button to generate a web page based on the notes. The students can then review the notes from their web browser at home, or can even view the notes of other sections to see how they differ. The instructor of the course can also visit the page, and make comments to be edited into the page in red text.

This was a small informal study, as our focus was on trying to tune the technology. We set up a preliminary version of the enhanced system mainly as an optional adjunct for one semester (with a total of 32 students in the Enhanced DTVI sections) to debug and assess usability. In a later semester we used the system as a full-blown basis for the sessions in a Journalism class at Chico. The Journalism class had 11 Enhanced DTVI students in 3 sections. Based on surveys and observations, we found that groups informally negotiated their own turn taking protocols, so that normally only one person was typing at a time. They sometimes used the note-taking tool to type comments to each other outside of the verbal channel, though this was uncommon. Interestingly, they generally did not see the group notes as a replacement for their own notes, preferring to have both. The instructor and students were generally enthusiastic about the system. The average of the Enhanced DTVI student grades was again higher than the average of the lecture student grades but statistical claims are weak with this small sample size. Unfortunately, students didn't often visit the web-page notes of other sections as we had hoped. But students particularly liked reading the instructor's comments scattered among their group's notes.

The study suggests a next step in which the Enhanced DTVI process and grades are compared with regular DTVI. Unfortunately, our time and resources were limited, so this system for now simply suggests an encouraging approach to the next generation of virtual collaborative learning.

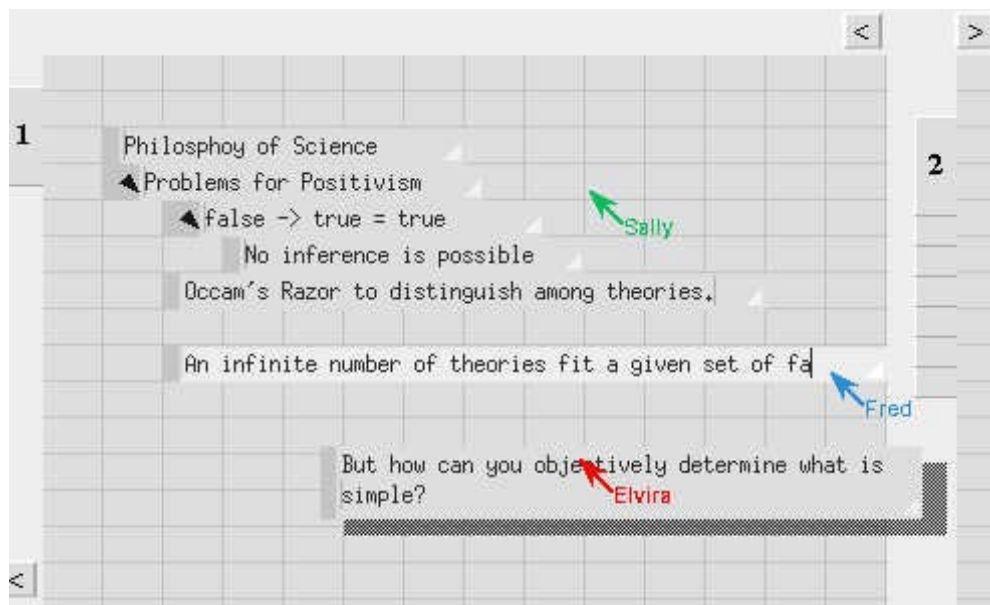


Figure 3. The shared outlining tool enables anyone to type anywhere, creating small text boxes on a gridded page. The text boxes can be picked up from one place, carried and dropped into another.

Conclusions

There is considerable prior research suggesting that communication technology can interfere with discourse. Since collaborative learning is dependent on discourse both for content and for group cohesion, we set out to determine whether the collaborative learning effect would be maintained in a virtual collaborative learning environment. Our analysis of over 700 student

grades suggests that the collaborative learning effect is fully intact with DTVI virtual environment, opening the door to the widespread use of more effective distance learning models than the lecture-based model currently being used.

There are many other lessons to be learned from this research project. These lessons are particularly compelling because of the size, duration, and real-world nature of the experiment.

We cannot make a one-to-one match between elements of the DTVI interface and their impact in terms of social process. But we can say on the whole that the DTVI experience did not provide quite the same degree of "warm fuzziness" that face-to-face interaction provides. We do not know whether it is possible or—given the potential benefits of reduced social proximity (i.e., greater willingness to admit gaps in knowledge, etc.)—even desirable to create a virtual collaboration environment that does provide an equivalent feeling of human contact. But we can say that virtual collaboration maintains a surprising amount of the group cohesion of face-to-face groups and that the collaborative experience is still very rewarding for participants, albeit perhaps in a different way than face-to-face collaboration.

Most importantly, our research shows that video-mediated communication can in fact support both the content and relational components of discourse necessary for effective collaborative learning. In particular, the DTVI environment appears fully capable of reproducing the collaborative learning effect to the extent that learning is measured by university course grades.

Furthermore, we have demonstrated that video-mediated collaboration can generate high levels of user satisfaction. While the DTVI students reported enjoying their experience slightly less than the TVI students, they reported enjoying it much more than a typical classroom lecture.

With DTVI generating higher academic performance and more enjoyment than classroom lecture, distance learning no longer need be considered a poor cousin to face-to-face instruction.

This study opens up many areas for future research. We do not know, for instance, to what extent eliminating network transmission artifacts contributed to our finding the relatively small amount of communication disruption in the DTVI and Enhanced DTVI environments. While much previous research has been done on the effects of latency, audio quality, etc., most of this research has focused on explicit, observable communication breakdowns. The effect of these breakdowns on group cohesion and other social aspects of collaborative discourse deserves more study. We also should point out that, in this experiment, each DTVI student was in a fairly quiet and interruption-free small room or cubicle. Should DTVI be brought to the user's everyday desktop, the distractions of office or home life might diminish one's ability and willingness to attend to the group.

Lastly, the positive response to the enhanced note taking facility suggests that such features may help the collaborative learning process. Facilities such as shared notes provide a common focus that bears directly on the task of attending to and thinking about the videotaped material, and the notes can be referenced for future study so group members have an interest in seeing they are done well. Perhaps other features could be added to specifically target group cohesion. Such enhancements suggest how virtual collaborative learning may be able to produce a stronger collaborative learning effect than a face-to-face collaboration.

Bibliography

- Cohen, E. (1994) Restructuring the classroom: Conditions for productive small groups. *Review of Educational Research*. 64(1): 1–35.
- Colston, H. L. & Shiano, D. J. (1995) Looking and lingering as conversational cues in video-mediated communication. In *CHI '95 conference companion*. 278–279. Denver, Colorado.
- Fish, R. S., Kraut, R. E. & Chalfonte, B. L. (1990) The VideoWindow system in informal communications. In *Proceedings CSCW '90*. 1–11. Los Angeles, CA.
- Fish, R. S., Kraut, R. E. & Root, R. W., & Rice, R. E. (1992) Evaluating video as a technology for informal communication. In *Proceedings CHI '92*. 37–48. Monterey, California.
- Gale, G. (1998) The effects of gaze awareness on dialogue in a video-based collaborative manipulative task. In *Proceedings CHI '98 Summary*. 345–346. Los Angeles, California.
- Gibbons, J. F., Kincheloe, W. R., & Down, K. S. (1977) Tutored videotape instruction: a new use of electronics media in education. *Science*. 195: 1139–1146.
- Isaacs, E. A. & Tang, J. C. (1993) What video can and can't do for collaboration: A case study. *Proceedings of Multimedia*. 93: 199–206. New York: ACM Press.
- Johnson, D.W. & Johnson, R. (1994) *Joining together: group theory and skills*. 5th ed., Englewood Cliffs, New Jersey: Prentice Hall.
- Kies, J.K., Williges, R. C. & Rosson, M. B. (1996) *Controlled laboratory experimentation and field study evaluation of video conferencing for distance learning applications*. Hypermedia technical report, HCIL-96-02 (<http://hci.ise.vt.edu/~hcil/htr/HCIL-96-02/HCIL-96-02.html>).
- Maloney, J. & Smith, R. B., (1995) Directness and Liveness in the Morpheus User Interface. In *Proceedings of the UIST '95 Conference*. 21–28. Pittsburgh, Pennsylvania.
- Monk, A. & Watts, L. (1995) A poor quality video link affects speech but not gaze. In *CHI '95 conference companion*. 274–275, Denver, Colorado.
- O'Conaill, B., Whittaker, S. & Wilbur. (1993) Conversations over video conferences: An evaluation of the spoken aspects of video-mediated communications. *Human-Computer Interaction*. 8: 389–428.
- Rocco, E. (1998) Trust breaks down in electronic contexts, but can be repaired by some initial face-to-face contact. In *Proceedings CHI '98*. 496–502, Los Angeles, California.
- Sellen, A. (1992) Speech patterns in video-mediated conversations. In *Proceedings CHI '92*. 49–59. Monterey, California.
- Sipusic, M., Pannoni, R., Smith, R. B., Dutra, J., Gibbons, J. F., & Sutherland, W. R. (1999) *Virtual Collaborative Learning: A Comparison between Face-to-Face Tutored Video Instruction (TVI) and Distributed Tutored Video Instruction (DTV)*. Sun Microsystems Laboratories Technical Report TR-99-72.

Smith, R. B., Hixon, R., & Horan, B. (1998) Supporting Flexible Roles in a Shared Space. In *Proceedings of Computer Supported Cooperative Work '98*. 197–206. Seattle, Washington.

Smith R. B., Wolczko, M. & Ungar, D. (1997) From Kansas to Oz: Collaborative Debugging when a Shred World Breaks. In *Communications of the ACM*. (40) 4: 72–78.

Storck, J. & Sproull, L. (1995) Through a glass darkly: what do people learn in videoconferences? *Human Communications Research*. 22(2): 197–219.

Authors' addresses

Randall B. Smith (randall.smith@sun.com)

Sun Microsystems Laboratories; 901 San Antonio Rd.; Palo Alto, CA 94303. Tel. (650) 336–2620.

Michael J. Sipusic (sipusic@krdl.org.sg)

21, Heng Mui Keng Terrace; Singapore, 119613. Tel. +65 874 6242

Robert L. Pannoni (pannoni@raport-sys.com)

Rapport Systems; 160 W. Campbell Ave. #364; Campbell, CA 95008. Tel. (408) 871 2517

The Spotless System: Implementing a Java™ System for the Palm Connected Organizer

Antero Taivalsaari, Bill Bush, Doug Simon

Introduction by Antero Taivalsaari and Bill Bush

What is this project all about?

The Spotless project developed a small-footprint Java™ Virtual Machine that later became the K Virtual Machine (KVM) product. The KVM is a key component of Java™ 2 Micro Edition (J2ME), a popular implementation of the Java™ programming language targeted specifically at small mass-market consumer devices such as cell phones, two-way pagers, and PDAs. It has been estimated that more than 100 million J2ME devices will ship worldwide in 2002, and more than 200 million in 2003.

Why/how is it significant to the field of research or to the particular technology it describes?

The Spotless system demonstrated that it was possible to build a Java runtime environment that was significantly smaller (by an order of magnitude) than the mainstream Java runtime environments of its time (1998). Demonstrating this was critical in convincing major consumer device manufacturers such as Motorola and Nokia to install the Java platform in their cell phones.

To which of Sun's products or technologies did it contribute?

K Virtual Machine (KVM)

Java™ 2 Micro Edition (J2ME)

If the paper is a milestone in an ongoing project, what do you envision as a possible future effect of the technology described?

KVM and related technologies will be used in hundreds of millions of wireless devices all around the world. The presence of the Java platform in such devices is likely to fundamentally alter their nature from static, voice-oriented devices to extensible, software-driven platforms that can support dynamically downloaded software and services.

HOW IT ALL BEGAN – by Bill Bush

The project got started when Antero came into my office one day to tell me about the Rex pocket organizer by Rolodex. It was the size of a credit card. We got to talking about how cool it would be to have Java on it, and how it wouldn't be that hard to do. Right then we went to Neil Wilhelm, our boss, who said fine, just put it on a Palm Pilot first, because Pilots were easy to develop for and many people had them (so demos would be easy). It took maybe half an hour to start the project. That was early 1998.

In May of 1998 we showed Spotless at the Sun Labs Open House, open to all Sun employees. In the program, we advertised that we'd put Spotless on any Palms that people brought in. I figured we'd get maybe five or ten adventurous customers. Both Antero and I spent the entire three hours stuffing Spotless onto Palms. The final count of installations was over a hundred. It was a phenomenon. Significantly, at the Open House, we also met the product people who ultimately adopted Spotless.

In August of 1998 several of us working on small Java implementations met with Mark VandenBrink of Motorola, who wanted to put Java on a pager for a demo at JavaOneSM. We showed him Spotless on the Pilot, and he said he'd been waiting two years for it. He was our first real customer.

The KVM (Spotless renamed) was announced at the 1999 JavaOneSM. To promote it, Sun sold at cost, to all conference attendees, Palm Vs loaded with the KVM. We had about six weeks to make the VM reliable and write some semi-interesting applications for it, which was intense, because mistakes would be really visible. It was manageable, though, because by then a product team was starting to form, so there were five or ten of us working on the JavaOne artifacts. JavaOne itself was another phenomenon, like the Open House. People were captivated by the concept, and the reality, of Java in the palm of their hand.

REFERENCE:

The best reference is the J2ME book that was published in May this year:

Roger Riggs, Antero Taivalsaari, Mark VandenBrink, Programming Wireless Devices with the JavaTM 2 Platform, Micro Edition. Addison-Wesley (Java Series), 2001. ISBN 0-201-74627-1.

The Spotless System: Implementing a Java™ System for the Palm Connected Organizer

Antero Taivalsaari, Bill Bush, and Doug Simon

SMLI TR-99-73

February 1999

Abstract:

The majority of recent Java implementations have been focused on speed. There are, however, a large number of consumer and industrial devices and embedded systems that would benefit from a small Java implementation supporting the full bytecode set and dynamic class loading. In this report we describe the design and implementation of the Spotless system, which is based on a new Java virtual machine developed at Sun Labs and targeted specifically at small devices such as personal organizers, cellular telephones, and pagers. We also discuss a set of basic class libraries we developed that supports small applications, and describe the version of the Spotless system that runs on the Palm Connected Organizer.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:

antero.taivalsaari@eng.sun.com
bill.bush@eng.sun.com

The Spotless System: Implementing a Java™ System for the Palm Connected Organizer

Antero Taivalsaari

Bill Bush

Doug Simon

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303

1. Introduction

The majority of recent Java™ implementations have been focused on speed. In contrast, relatively little effort has been spent making Java implementations small. Yet there is a large number of consumer and industrial devices and embedded systems that would benefit from a small Java implementation supporting the full Java programming language [JLS96]. In these domains small system size is usually crucial, whereas speed, while certainly important, is often a secondary consideration. Also, for many consumer device manufacturers there are other criteria, such as portability and a fast learning curve for the developers, that are often more valuable than raw speed.

The Spotless system is a new Java implementation developed at Sun Microsystems Laboratories. It is based on our experience building JavaInJava [Tai98], probably the first Java virtual machine written in the Java programming language. We had two goals. First, we wanted to build the smallest possible “complete” JVM that would support the full bytecode set, class loading, and standard non-graphical libraries. Rather than speed, the main design criteria for the JVM were small size, portability, and readability of the code. Second, we wanted to engineer a small Java implementation that included basic classfile support for small applications.

The current Spotless JVM takes only a few tens of kilobytes (on the order of 30 to 50 kilobytes of static memory on a PC, depending upon compilation and debugging options). It runs anywhere from 30% to 80% of the speed of JDK™ 1.1 without the Just-In-Time (JIT) compiler, which is noteworthy given that there is no machine code in the source code, and that only a few simple optimization techniques are used.

The Spotless JVM supports platform-independent multithreading, copying or non-moving garbage collection (two different garbage collectors have been implemented), and optional quick bytecodes. But, perhaps more importantly, the source consists of only 25 C/C++ files containing approximately 14,000 lines of thoroughly commented, highly portable code. Various debugging and tracing options are provided to help porting efforts. This base version runs on Windows 95, Windows NT, and Solaris™.

In order to test the Spotless JVM on a real-world embedded device, we ported it to the Palm Connected Organizer. We also developed a small set of classfiles that support Palm applications. In this report we summarize our experiences building the JVM, making it fit on the Palm computing platform, and implementing the small class library.

2. Issues in implementing a small Java virtual machine

2.1. Design criteria for the Spotless JVM

The Java programming language was originally designed for use in various consumer devices and appliances such as cable TV set-top boxes and personal digital assistants. The goal was not just to develop a better programming language, but to develop an entirely new, more dynamic, platform-independent way of creating applications for these devices. Rather than having to build a hard-wired, platform-dependent application for each particular device, the Java programming language would allow the applications for these devices and appliances to be enhanced and extended dynamically. Ideally, the same applications would run unchanged in any device supporting the Java virtual machine.

Achieving this goal would open up completely new dimensions for application development for embedded systems, allowing, for example, mobile phone or set-top box manufacturers to develop extensible devices whose customers could flexibly enhance their systems as their needs would grow and new services would become available. This flexibility contrasts with the majority of today's devices, which have a fixed feature set that cannot easily be changed after the device is manufactured. Ultimately, the Java programming language could serve as the basis for a whole new class of extensible devices and appliances, each using a standard Java virtual machine and a fully compatible set of libraries, regardless of the differences in the underlying hardware.

Unfortunately, many consumer devices such as cellular telephones, pagers, wristwatches, bicycle computers, and radios still have far more limited hardware resources than is required by a typical Java virtual machine. Today, the RAM in the majority of embedded devices is still measured in kilobytes, whereas most Java systems typically require at least a few megabytes of RAM to run. Even though the typical amount of memory in consumer devices is constantly increasing, it will still take a long time for memory prices to decline to the point that megabytes of memory will be available in the majority of consumer devices. Furthermore, by the time this happens, it is possible that, as computing becomes more ubiquitous, an entirely new class of low-end devices with much more limited resources will already have emerged. In general, it is likely that at least for the next 3 to 5 years, the amount of RAM in the majority of embedded systems will still be measured in tens or hundreds of kilobytes rather than in megabytes.

The main objective of the Spotless project was to study implementation techniques that would allow the static and dynamic memory footprint of the Java programming language to be reduced substantially, preferably by at least an order of magnitude, without sacrificing the functionality of the Java programming language. In other words, we wanted to implement a small Java system that would need only a few tens or hundreds of kilobytes of memory to run, but that would nevertheless support the complete bytecode set, dynamic class loading, garbage collection, multithreading, and other essential features of the Java virtual machine. Other central goals were portability and ease of understanding.

These requirements contrast with some other embedded Java virtual machine projects that have tried to reduce the memory footprint by removing various language features, such as floating point arithmetic, multithreading or garbage collection (Java Card™), and by replacing dynamic class loading with precompiled, preloaded classes, possibly stored in ROM (Java Card and EmbeddedJava™). While such restrictions are reasonable for many platforms, we felt that there is a huge potential class of devices and applications in which it is crucial to support dynamic class loading and the complete bytecode set. After all, it is the dynamic nature, extensibility, and portability of the platform that is the main selling point of the Java programming language to many embedded systems developers and customers. Furthermore, dynamic class loading is highly desirable for devices implementing the Jini™ distributed architecture.

In general, unlike other embedded Java system development projects, we were unwilling to sacrifice the completeness of the virtual machine. In contrast, we were willing to simplify and reduce the libraries, provided that this could be done in a fashion that would preserve upward compatibility between applications for small systems and for large platforms.

2.2. Why are Java systems so large?

Current Java systems require on the order of a few megabytes to a few tens of megabytes of memory to run. In spite of this apparently large memory footprint, a Java virtual machine is really a relatively simple engine. The bytecode set of a Java virtual machine is fairly compact, and the runtime support needed in a basic virtual machine is relatively small. Nor is there inherently any need for special memory areas other than the standard memory heap(s). Certainly a Java virtual machine is much more complex and memory-consuming than, for example, a Forth programming environment [Bro84], but need not be much more complicated than virtual machines for more closely related programming languages, such as Smalltalk-80 [GoR83] or Self [UnS87]. Also, the general implementation techniques needed for building virtual machines have been studied for two decades and are quite well understood.

Then why do JVMs require so much memory? One explanation is the desire for speed. Modern Just-In-Time (JIT) compilers are pushing the limits of implementation technology, introducing a lot of additional complexity to the virtual machine and often requiring many additional specialized memory regions. Also, the use of native threading and pre-emptive scheduling introduces extra complexity due to mutual exclusion and synchronization. Furthermore, modern high-speed garbage collectors can be fairly complex, usually requiring several separate memory spaces in order to manage multiple generations of objects.

However, by far the most important reason for the large memory consumption of Java systems is the size of the standard class libraries. For instance, the extensive internationalization features that have been built in the standard I/O libraries do not come without a cost. In fact, the majority of the approximately 130,000 bytecodes that are executed during initialization of the JDK 1.1 JVM are spent on initializing the various internationalization and I/O facilities. Similarly, the various portability, security, reflection, and remote invocation features all add extra layers to the libraries, meaning that more and more classes need to be loaded even when invoking seemingly trivial operations. For instance, in JDK 1.1, printing out one string of text requires 20 to 30 classes to be loaded. Even though these advanced features are extremely important and valuable in many applications, their cost is prohibitive for most embedded systems. The presence of a large number of library classes also means that many native functions are needed to implement the platform-specific functionality needed by the library classes, even though a typical application uses only a small fraction of those functions.

Part of the high memory consumption of Java systems can also be explained by the way classfiles are stored. A classfile is a portable, device-independent representation of a class. As such, it is not efficient as a runtime structure. When classfiles are loaded into a Java virtual machine, it has to create more efficient internal representations (such as method tables and field tables to perform efficient runtime lookups) for every class. However, since bytecodes use classfile-specific constant pool indices, it also has to maintain the constant pool information that was originally loaded from each classfile, even though much of the same information is also stored in the method and field tables. This means that a straightforward JVM implementation may inherently have much internal redundancy.

2.3. Building a smaller Java system

In the previous section we argued that the Java virtual machine need not be very complicated or particularly large. If one follows good software engineering principles and practices, and adheres to basic implementa-

tion techniques, it appears actually rather easy to write a JVM that would be considerably smaller than a hundred kilobytes (excluding possible C/C++ runtime libraries). Such a straightforward bytecode interpreter implementation of the JVM might not be very fast, but there are ways to improve performance without requiring much additional memory.

Similarly, by carefully employing a few simple techniques, it is possible to greatly shrink the class libraries.

- (1) Start with nothing and add only what is absolutely necessary, rather than starting with the complete JDK and removing what was not needed. Much of the standard JDK is simply not too big to fit on a memory-limited platform.
- (2) Remove dependencies between classes. The standard JDK has many dependencies, and, if one method is needed from a class, the entire class has to be loaded. Dependencies were removed by:
 - folding classes together, such as `System` and `Runtime`;
 - extending core classes, such as adding `isInteger` and `toInteger` to `String`, thereby avoiding the need for the `Integer` class;
 - removing pass through functions, such as `System::exit`, which calls `Runtime::exit`; and
 - inlining calls, such as calls to `System::arrayCopy`.

Also, starting with nothing makes it easier to discover dependencies in an incremental fashion and deal with them appropriately.

- (3) Remove alternate forms of functions; for example, `String(char[])` is simply a call to `String(char[], 0, <length-of-char-array>)`. Note that, with smart inlining in a performance-oriented implementation, much of the overhead of these forms is eliminated.
- (4) Eliminate classes that are seldom used or that can be added by the user if necessary, such as `ThreadGroup`.
- (5) Avoid classes that create many short term objects, such as `Point` (by, for example, using `x` and `y` coordinates as parameters rather than `Point` objects).
- (6) Reduce the number of distinct exceptions to the bare minimum, because each exception is its own class with all the associated overhead.
- (7) Take advantage of native platform support for I/O and graphics. AWT in particular is inappropriate and too large for embedded platforms.

The ideas described above were used in the development of the Spotless system. We started by building a straightforward, small, consistently written, low-performance bytecode interpreter. When we ported it to the Palm Connected Organizer we initially allowed host system functions (PalmOS functions in our case) to be called directly from the virtual machine to speed up development and to study memory constraints and other platform-specific restrictions in more detail. After getting the basic system to run, we started studying space-efficient optimization techniques to improve the performance of the virtual machine, and, in parallel, started implementing a subset of the libraries that would be compatible with the standard libraries but with a substantially smaller memory footprint. The results of this work are described below.

3. Overall design of the Spotless JVM

The overall design of the Spotless JVM is similar to that of our JavaInJava virtual machine [Tai98]. Like JavaInJava, the Spotless JVM is a new implementation based on the Red Book specification [JVM96]. The virtual machine is implemented around a straightforward bytecode interpreter with some optimizations. The

representation of the internal structures is conventional in the sense that each class has a constant pool, method table, field table and interface table (see Figure 1). However, unlike JavaInJava, where every internal structure is implemented as a separate object, in the Spotless JVM most of the structures have been implemented as linear tables to provide faster access and to conserve memory space. Also, unlike JavaInJava, we have also taken advantage of various low-level features of the C programming language to achieve better performance.

Portability was one of the main goals in the Spotless JVM design. For this reason, non-portable code (such as functions to obtain information about the host computing system) has been minimized and limited to a single file. Even multithreading and the garbage collector have been implemented in a completely platform-independent fashion. Rather than relying on external interrupts, the multitasker performs thread switching on the basis of the number of bytecodes the current thread has executed, making thread switching more deterministic and easier to debug, thus facilitating porting efforts. This approach is practical in part because we are not targeting multiprocessor platforms.

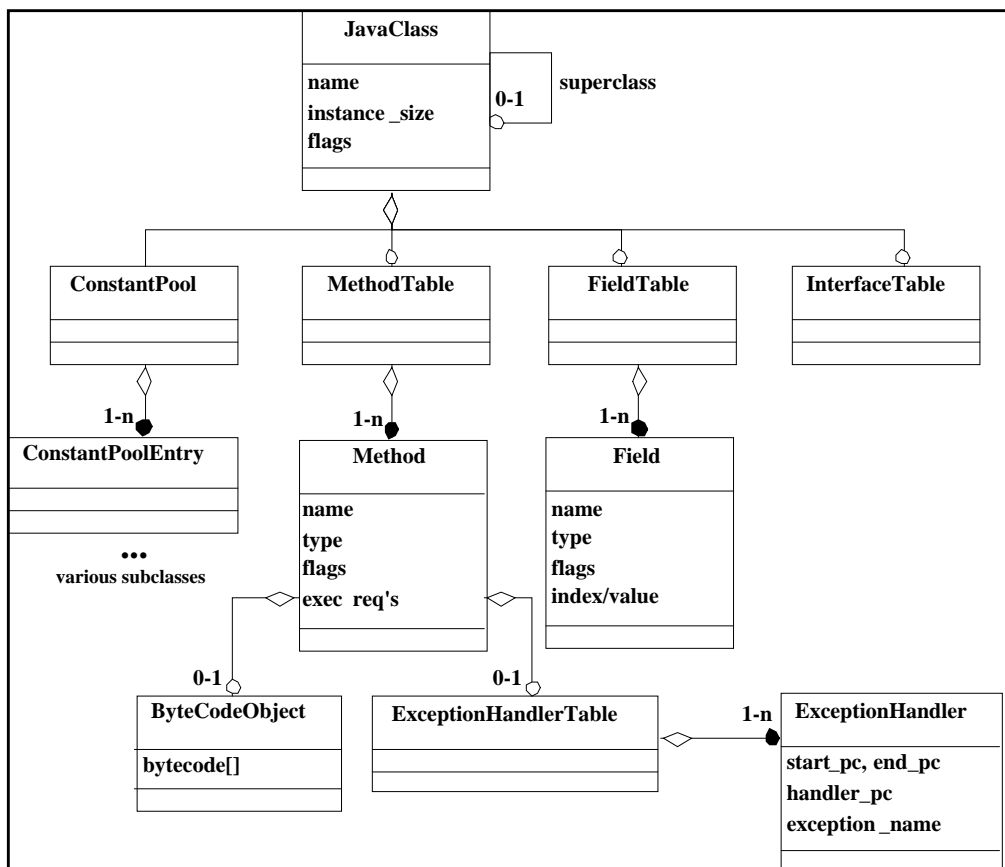


Figure 1: Internal class representation.

Garbage collector(s). Two different garbage collectors have been implemented. The original collector used a copying garbage collection algorithm with multiple memory spaces. However, this approach turned out to consume far too much memory for devices that have only a few tens of kilobytes of memory available. In general, modern generational garbage collection algorithms tend to be unsuitable for applications with extremely limited memory. For this reason, a simpler, non-moving, single-space mark-and-sweep garbage collector was written. The new collector operates well with heap sizes of just a few tens of kilobytes. Both collectors are handle-free, that is, object references are always direct rather than indirect.

Native function support. To minimize memory usage, the native functions and wrappers used to call host system functions have been implemented by making them a part of the virtual machine, rather than by using the Java Native Interface. Currently, the extent of native function support varies by platform. On Windows and Solaris it is extensive enough to cover most non-graphical libraries; due to size constraints it is very limited on the Palm organizer. There is no support for AWT, Swing, or any other graphics libraries, so graphics are done by calling platform-specific graphics functions.

C/C++ runtime libraries. The Spotless JVM has been implemented primarily in C, with small pieces in C++. The system is thus somewhat dependent on the C runtime libraries. This could be a problem on small platforms that do not have support for C libraries. However, the number of C runtime functions that the virtual machine calls has been minimized, to ease implementation or replacement of those functions. If any of the debugging modes in the source code are enabled, then the target platform must support the `fprintf` function in the standard C I/O library (`stdio`), or a function with the same interface as `fprintf`.

The Spotless JVM source is usually compiled using a target-system-specific C cross-compiler rather than a regular compiler. Development for the Palm organizer was done on PCs, for instance. It should be remembered that many embedded systems platforms do not have support for all the features of the C language (for instance, floating point support is not available on many small platforms), and hence the cross-compiler might not be able to generate support for certain bytecodes.

Compilation options; optimizations. The Spotless JVM source code includes many compile-time options for tuning virtual machine parameters (such as the size of the internal stacks), for debugging (to help porting efforts), and for choosing various optimizations (size versus speed). Most of the optimizations in the Spotless JVM are focused on minimizing space. In general, we would like to minimize the amount of memory required without slowing down the virtual machine, but in some situations the virtual machine developer will have to choose between size and speed.

4. The Spotless system on the Palm Connected Organizer

Because of its small size, our original target device candidate for the Spotless system was the PC-card-sized Rolodex REX personal organizer sold by Franklin Electronic Publishers (<http://www.franklin.com/rex/>). However, development tools for that device were not readily available, so we chose the Palm Connected Organizer by 3Com, because it is very popular, is well supported in terms of cross-platform development, and is a typical embedded device with limited dynamic memory.

The Spotless system for the Palm organizer is composed of five major components:

- the virtual machine itself,
- a small set of class libraries,
- a database and user interface for storing and managing classfiles,
- utilities for moving classfiles onto the organizer from a desktop machine, and
- demo applications that run on the Palm organizer.

Spotless JVM on the Palm organizer. Porting the Spotless JVM to the Palm computing platform was relatively straightforward using version 4 of the Metrowerks Palm development environment. Some data alignment problems were encountered. The JVM was originally designed to run on devices with four-byte (32-bit) data alignment, whereas the Palm organizer uses two-byte (16-bit) data alignment by default. Because of this, many four-byte read and write operations failed or caused unexpected results.

The limited memory model of the Palm computing platform was also problematic. Even though the Palm devices can have megabytes of RAM, the amount of real, directly addressable, dynamic RAM is small. The Palm computing platform supports a maximum of 96 kilobytes of dynamic RAM. A PalmPilot Professional device has only 64 kilobytes of dynamic RAM; older PalmPilot devices have only 32 kilobytes. The rest of the memory, which is static RAM, behaves like a RAM disk and has to be accessed using special PalmOS database API calls. We originally planned to use the static RAM as virtual memory for the Spotless JVM, but unfortunately the database API calls turned out to be so slow that this was not practical. Since the PalmOS system (the operating system of the Palm computing platform) itself normally uses 10-20 kilobytes of dynamic RAM, the current Palm Spotless JVM has less than 40 kilobytes for the heap on the PalmPilot Professional and less than 64 kilobytes on the Palm III. This imposes serious restrictions on the kinds of applications that can be run.

The PalmOS system has no direct support for standard C or C++ libraries. Many of the individual library functions are supported, but the names of the functions differ from those in the standard libraries (for example, `strcpy` is `StrCopy`, and `sprintf` is `StrPrintf`). Also, the PalmOS does not have support for C or C++ style input/output (such as the `stdio` library). For this reason, the PalmOS port of the VM includes its own support for `fprintf` (including the standard output streams `stdout` and `stderr`), which made debugging much easier. Debugging was also assisted by developing on the Palm OS Emulator (freely available from 3Com at <http://www.palm.com/devzone/pose/pose.html>), which is integrated nicely with Release 4 of Metrowerks Codewarrior for PalmOS.

The current Palm Spotless JVM supports the complete bytecode set, full class loading, garbage collection, and various Palm native functions. The total size of the executable (PRC file) containing the virtual machine, the user interface (discussed below), and native function support for the small class libraries is currently 40 kilobytes.

Some memory reducing techniques were added to the Palm Spotless JVM. Lazy class loading is performed, in which a class is loaded only during execution when one of its methods is needed; only `Object` and `Class` are loaded at initialization. Minimal perfect hashing is used for the native function table, so that the table has no empty or chained entries. Duplicate strings in loaded classes are eliminated by using a canonical string pool, which on average saves roughly 50% of the memory used for string storage.

A small set of class libraries. A minimized set of class libraries supporting the Palm platform was developed, including a very small subset of `java.lang` and a set of new libraries specific to the Palm computing platform (but of more general utility for personal organizers and perhaps small devices generally), called *spotless*.

java.lang.Object. `Object` is required as the root of the object hierarchy. A number of its methods have been retained, as they are useful even on a small platform. The `getClass`, `toString`, and `hashCode` methods can be used for debugging at the source level using `printf` style debugging. Also, `hashCode` and `equals` provide a good base for a simple hash table implementation. The remaining methods all involve thread synchronization. Combined with the `Thread` class, these methods provide the threading capabilities that are integral to the Java environment.

java.lang.Class. `Class` is fundamental and has to be supported, although it is unclear how often its reflective capabilities are used. Given that the majority of its methods are native, it is easier to include it in the core set of classes, since adding native methods later requires access to the VM source code.

java.lang.String and *java.lang.StringBuffer.* Both classes are used by the `javac` compiler to implement string constants, and are present in code that uses string constant expressions even if they are not explicitly used

in the source. In the Spotless system, String does all the conversion of primitive types to and from Strings that is normally done by the primitive wrapper classes (such as Integer and Boolean). For example, new methods in String provide Integer conversion functionality (specifically `isInteger` and `toInteger`). StringBuffer uses these conversion facilities in its versions of `append`. The `append` methods are the primary reason StringBuffer has to be provided in addition to String. These methods are generated by `javac` to implement expressions such as `str = "Age: " + age;`

java.lang.Runtime. This class is critical in a memory-limited embedded environment. It is a combination of the standard Runtime and System classes. It provides methods for examining the runtime environment. It is particularly useful in the absence of the OutOfMemoryError class. The `currentTime` method differs slightly from the `currentTimeMillis` found in `java.lang.System` in that it returns the time elapsed since VM initialization instead of some well known epoch. This is because the fine grain timer of the underlying OS is reset upon each OS reset, rather than set to some absolute time. It also stops while the device is in sleep mode. The Runtime class also provides a method to return an optionally seeded Random number.

java.lang.Thread. This class provides basic threading functionality. Threads can be started, put to sleep, and yielded. There are also static methods that return the number of threads in the system and return a handle on the current thread. There are no methods to stop a thread explicitly, as these have unclear semantics and have been deprecated as of JDK 1.2. In keeping with our general minimalist philosophy, no threads are created implicitly, and there is no support for thread groups, which are too heavyweight and can be created for specific applications if needed. Threading is nonetheless of great value because it greatly simplifies event driven applications.

java.lang.Throwable, java.lang.Exception, java.lang.RuntimeException, java.lang.NullPointerException, java.lang.IndexOutOfBoundsException, and java.lang.Error. IndexOutOfBoundsException and NullPointerException are the core exceptions. The other classes are required by `javac` when code uses exception handling. Applications can also define their own exceptions, but should do so only when necessary, because exception handlers consume relatively large amounts of memory. In all other error cases, an error message is printed, indicating the class, method, and bytecode offset where the error occurred. Future plans to reorganize the memory management of classes will most likely result in more of the standard exceptions being supported.

java.lang.Runnable. This interface allows objects to run in a thread without their class having to extend the Thread class.

As mentioned above, in addition to the `java.lang` classes, there are classes supporting a Spotless application framework. These classes provide a simple model for event handling and simple graphics capability.

spotless.Spotlet. The Spotlet class implements pen-based event handling. It consists of handler methods that are invoked when associated events occurs. The handler methods are `penDown`, `penUp`, `penMove`, and `keyDown`. Applications using this class are called *spotlets*. Spotlets must register with the Spotlet class to get and relinquish the event focus, using the `register` and `unregister` methods. Event handlers run in a dedicated VM thread. The VM remains alive if a spotlet is registered, even if no events are being processed. The VM will conserve power by putting the Palm device to sleep after an interval with no activity.

spotless.Graphics. The Graphics class provides straightforward access to the underlying functionality of the Palm platform. Methods exist to draw shapes (`drawLine`, `drawRectangle`, and `drawBorder`), display strings (`drawString`), get display-specific properties (`getHeight` and `getWidth`), set a clipping rectangle (`setDrawRegion` and `resetDrawRegion`), move a region (`copyRegion`), and draw bitmaps (`drawBitmap`). Numerous drawing modes are available; plain, grayscale, inverted, and erase.

spotless.Bitmap. The Bitmap class is used to represent simple black and white bitmaps. Having a bitmap object simplifies the use of bitmaps, since the bitmap data (pixels) need only be specified once (during construction of a Bitmap object) as opposed to every time a bitmap is drawn.

Database and user interface for storing and managing classfiles. One of our main goals in developing the Spotless JVM was to implement a small virtual machine that would support dynamic loading of classfiles. However, the Palm computing platform does not have a file system. For this reason, we had to implement our own database for storing and managing classfiles. This was accomplished by using the database and heap APIs of the PalmOS libraries. Rather than storing classfiles in a normal file system, the classfiles are stored in the RAM database located in the static RAM of the Palm device. In order to make it possible to manage the Palm classfile database, we also had to implement a simple user interface, known as the *class manager*, for displaying, invoking, and deleting classes and packages. Screen snapshots illustrating the user interface are shown in Section 5.

Classfile download utilities. Ideally, we would like to have a complete, integrated Java programming environment for the Palm computing platform, allowing classes to be written, compiled, debugged, and executed on the Palm organizer itself. Unfortunately, like many other small devices, the Palm organizer does not have enough memory to support a complete programming environment. Furthermore, writing source code with a pen instead of a keyboard would be tedious. The small screen size would also make the editing of source code rather difficult. In order to avoid these problems, software development takes place on a desktop machine such as a PC or a UNIX workstation. Regular Java development tools can be used for writing source code and compiling the source files into classfiles. Classfiles are then transferred to the Palm device using one of two mechanisms.

HotSync classfile data conduit. The HotSync classfile data conduit we implemented is an extension of the Palm computing platform's standard HotSync software used for synchronizing an organizer with a desktop computer. The PC version of the data conduit is implemented as a dynamically linked library (DLL) invoked automatically during a HotSync operation. When invoked, the conduit transfers new or updated classfiles from the Palm software directory on the PC to the classfile database on the Palm organizer.

Database packager. The database packager is simpler than the data conduit. The packager, called *mkpdb* and written in Java, packages classfiles into a pdb (Palm data base) file that can be installed with the Palm Install tool, just like a prc (application binary) file.

At some point either utility may be extended to perform standard bytecode verification and annotate classes for the class manager (indicating whether a class is executable—has a main entry point, or is a system class).

Demo applications. Various demo programs have been implemented, illustrating the capabilities of the Palm Spotless JVM. These programs range from simple graphical and numerical tests to fractals and three-dimensional graphics.

5. Using the Spotless JVM on the Palm Connected Organizer

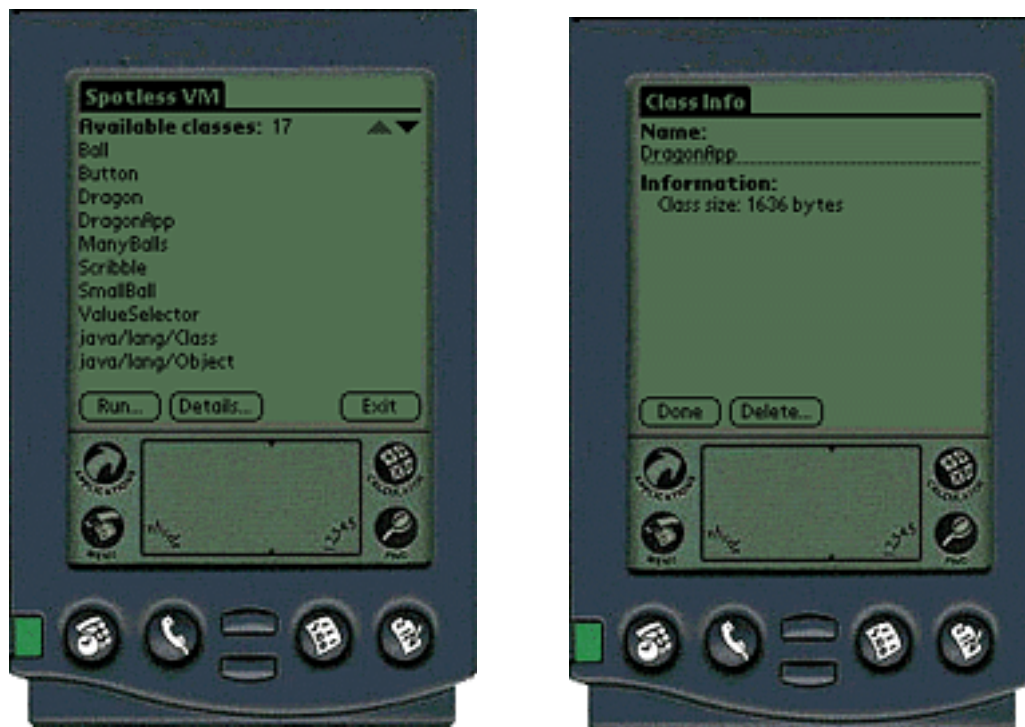


Figure 2: Spotless class manager display and class details dialog.

As discussed above, the Palm version of the Spotless JVM is built around an application known as the *class manager*. The class manager serves as the primary interface to classfiles on the Palm organizer, allowing the user to see which classes and packages are available, launch Java applications, see details of classes, and delete them as necessary.

The primary class manager display, entitled “Spotless VM”, appears on the left in Figure 2, listing the currently loaded classfiles. The “Class Info” dialog appears on the right, showing the name and the size of a selected classfile, and providing the option to delete it.

To run a classfile, the user selects it from the list of available ones and presses the “Run” button. Another display, entitled “Run Class”, then appears, and is pictured in Figure 3. This display allows the user to supply options and command line parameters to the virtual machine, and, if the “Verbose” option is turned on, follow the initialization progress of the virtual machine. The “Scroll Delay” list, shown in expanded form on the left in Figure 3, is used to control scrolling of the standard output stream during execution. The “ViewOutput” button opens another display, shown in Figure 4, that allows the user to view the contents of the two output streams, stdout and stderr, after execution. These buffers are cleared when the “Cancel” button is pressed and the user returns to the primary class manager display.



Figure 3: Class invocation dialog before execution (left) and after execution (right).

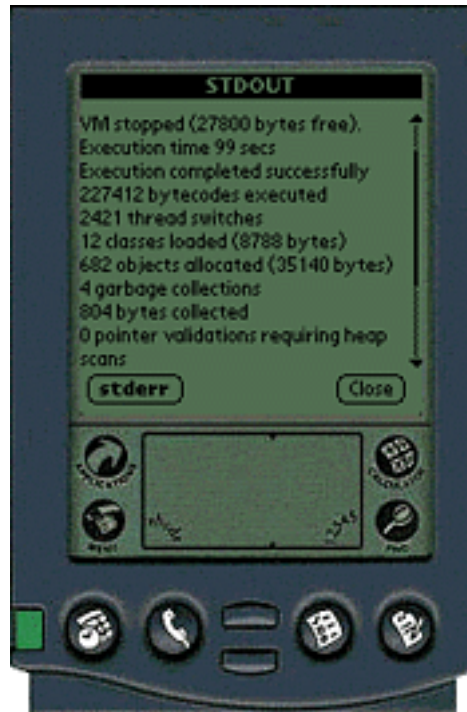


Figure 4: Output stream display after execution.



Figure 5: Java applications running on the Palm organizer.

Figure 5 shows two Java programs (DragonApp and Scribble) running on the Palm organizer. These programs use the spotless application framework presented earlier. Each program is a single spotlet; the user can switch between the two by pressing the arrow in the lower right hand corner.

6. Current status and experiences

The design and implementation of the Spotless JVM was started in February 1998. The first version of it ran in April, and a suite of small demonstration programs were running in May. Garbage collection was implemented in June. The minimized class libraries were working in November.

One of the main things we learned during the implementation of the Spotless system is that by restricting the use of the standard libraries, a Java system can easily fit in a few tens of kilobytes.

Since performance was never a major consideration for us, we do not have detailed benchmarks. Being a pure bytecode interpreter, the Spotless JVM cannot compete with virtual machines with a JIT compiler. The Windows 95/NT version runs between 30% and 80% of the speed of Sun's standard JDK 1.1 on Windows 95/NT without JIT, usually performing better on large applications and worse on those that involve a lot of numerical computation. This is reasonable, given that there is no machine code in the Spotless JVM source code, and that internal virtual machine registers—such as the instruction pointer and the stack pointers—are stored in regular C variables rather than in machine registers. By adding a few dozen lines of machine code to the most critical locations in the source code, performance could probably be substantially improved.

Measuring and comparing the performance of the Palm Spotless JVM is even more problematic, because there are no real points of reference. Since the PalmPilot Professional model we used for measurement only has a 16 MHz Motorola Dragonball (68328) processor with a 16-bit external data bus, the performance of the virtual machine on it is nowhere near its performance on a modern PC or UNIX workstation. However, the performance of the simple graphics programs seems surprisingly snappy. An empty loop (100,000 iterations) takes about 15 seconds to execute on the PalmPilot Professional—about five times slower than in Ruka, a simple Forth-like threaded code interpreter that we wrote earlier to understand the memory and performance limitations of the Palm device. This difference appears to be due to the fact that Java compilers (javac in particular) generate four bytecodes (IINC, ILOAD, LDC, IF_ICMPLT) for a simple loop control structure, whereas a Forth-like interpreter typically uses a single instruction for performing the same operation.

7. Future directions

The Spotless system is a fairly complete JVM implementation with a high quality source code base, and has enough library support to enable the writing of small Java applications. Nonetheless, it is still an experimental system missing various JDK features. As such, the system may be used as the basis for more research and experimentation.

On the research side we are interested in carefully adding functionality to both the java.lang subset libraries and to the spotless library. We are also interested in designing and implementing more interactive development environments for small devices, to avoid the lengthy edit-compile-download-run cycle. There are various other relevant research areas, such as the real-time aspects of an embedded JVM, that should be studied in detail.

On the more practical side, a Palm database access framework and explicit class unloading would be very useful for writing more practical applications. Better performance could be obtained by rewriting the interpreter loop in assembler. It would also be interesting to rewrite some of the standard Palm applications (most of which are small) in the Java programming language.

We are also interested in porting the Spotless JVM to other devices such as the Rolodex REX organizer or palm-sized PCs. In general, there are many potential devices, including cell phones, pagers, credit card readers, and personal organizers, that could be target devices for porting efforts.

8. Conclusions

In this paper we have summarized our experiences in building the Spotless system, a Java system for the Palm Connected Organizer. We described the general design constraints for a small Java system, presented the overall design of the Spotless JVM and class libraries, and then discussed our experiences building the Spotless system for the Palm organizer. Screen snapshots illustrating the user interface built for the virtual machine were also included.

The Spotless project shows that a Java system does not have to be particularly large or complicated. In general, the results of the project have been encouraging, since they show that it is relatively easy to build a JVM that fits in a small device with limited memory, such as the Palm organizer. The Spotless JVM is based on a conceptually clean and simple overall design and has a portable, well commented source code base that contains various options and hooks to help porting efforts. The project has also shown that the Java libraries can be both reduced and augmented to support programming on embedded devices.

9. References

- Bro84 Brodie, L., *Thinking Forth: A Language and Philosophy for Solving Problems*. Prentice-Hall, 1984 (2nd edition 1994).
- GoR83 Goldberg, A., Robson, D., *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.
- JLS96 Gosling, J., Joy, B., Steele, G., *The Java Language Specification*. Addison-Wesley, 1996.
- JVM96 Lindholm, T., Yellin, F., *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- Tai98 Taivalsaari, A., *Implementing a Java Virtual Machine in the Java Programming Language*, Technical Report SMLI-98-64, Sun Microsystems Laboratories, March 1998 (23 pages).
- UnS87 Ungar, D., Smith, R.B., “Self: the power of simplicity”. In Meyrowitz, N. (ed): *OOPSLA'87 Conference Proceedings* (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987. pp.227-241.

The Interactive Performance of SLIM: A Stateless, Thin-client Architecture

Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt

Introduction by J. Duane Northcutt

This paper provides results from work that began as a Sun Labs research project (named NewT), evolved into a product development project (known as Corona), and finally resulted in the Sun Ray™ product. This was the last in a series of research projects by a group that was formed when Sun Labs was founded and spanned the Lab's first decade. This group's preceding projects included: HDTV Workstation, OS and Network Support for Digital Audio and Video, Video over ATM, High-Speed Networking with ATM, Dynamic Auctioning of Network Bandwidth, Microeconomic Models of System-Level Resource Management, SunTuner, TV Datacasting, Sun Media Center, Java Media APIs, and NetCam.

The key observation behind this work is that new trade-offs between computation and communication are enabled by the fact that bandwidth is rapidly becoming less expensive and more abundant. This paper shows how it is possible to create low-cost, fixed-function devices that can deliver a high-end multimedia workstation experience, while simultaneously avoiding the usual demand for upgrades as application resource requirements increase. Based on the concepts put forth in this paper, Sun Ray™ became the first (and to date, the only) technically and economically viable "thin client" product.

The NewT project was an attempt to explore the limits of how thin a client could be made without compromising the user experience, even as applications evolve in unanticipated ways. Work prior to this has primarily been focused on the "stripped-down PC" approach to thin clients, which attempted to lower costs by restricting functionality. Sun Ray is different from all types of thin clients in that it does no computation locally and retains no (hard) state locally. This work established that no stable architectural point exists between the traditional PC (where all computation and state is local), and the fixed function Sun-Ray-like device (where all computation and state exists remote from the client).

This research project led to the creation of a business unit (the Information Appliances Group) and influenced a wide variety of other groups within Sun. It provided the newly-formed education group with a product to suit its market needs, and also provided the desktop group with an alternative client device for the low end of its markets. In addition, the Information Appliances Group acquired StarDivision in order to obtain a Solaris-based office application suite (StarOffice™) that would meet the needs of targeted Sun Ray markets.

The Sun Ray product was built by a small group, in an extremely short period of time, at an exceptionally low development cost. The initial product development (both hardware and software), and the initial internal deployment, were done entirely with the NewT prototypes created in Sun Labs, and a single (eight CPU) multiprocessor server.

The Sun Ray product would not have been possible without Sun Labs. It provided the environment that made the genesis of the idea possible, as well as the resources and support to allow the concept to be made real and properly explored. Even well after the product introduction, there were many assertions that Sun Ray was not possible, and so it seems quite likely that the product would never have gotten past the viewgraph stage were it not for the incubating function provided by the leadership of Sun Labs.

The interactive performance of SLIM: a stateless, thin-client architecture

Brian K. Schmidt*, Monica S. Lam*, J. Duane Northcutt†

*Computer Science Department, Stanford University
{bks, lam}@cs.stanford.edu

†Sun Microsystems Laboratories
duane.northcutt@sun.com

Abstract

Taking the concept of thin clients to the limit, this paper proposes that desktop machines should just be simple, stateless I/O devices (display, keyboard, mouse, etc.) that access a shared pool of computational resources over a dedicated interconnection fabric — much in the same way as a building's telephone services are accessed by a collection of handset devices. The stateless desktop design provides a useful mobility model in which users can transparently resume their work on any desktop console.

This paper examines the fundamental premise in this system design that modern, off-the-shelf interconnection technology can support the quality-of-service required by today's graphical and multimedia applications. We devised a methodology for analyzing the interactive performance of modern systems, and we characterized the I/O properties of common, real-life applications (e.g. Netscape, streaming video, and Quake) executing in thin-client environments. We have conducted a series of experiments on the Sun Ray™ 1 implementation of this new system architecture, and our results indicate that it provides an effective means of delivering computational services to a workgroup.

We have found that response times over a dedicated network are so low that interactive performance is indistinguishable from a dedicated workstation. A simple pixel encoding protocol requires only modest network resources (as little as a 1Mbps home connection) and is quite competitive with the X protocol. Tens of users running interactive applications can share a processor without any noticeable degradation, and many more can share the network. The simple protocol over a 100Mbps interconnection fabric can support streaming video and Quake at display rates and resolutions which provide a high-fidelity user experience.

1 Introduction

Since the mid 1980's, the computing environments of many institutions have moved from large mainframe, time-sharing systems to distributed networks of desktop machines. This trend was motivated by the need to provide everyone with a bit-mapped display, and it was made possible by the widespread availability of high-performance workstations. However, the desktop computing model is not without its problems, many of which were raised by the original UNIX designers[14]:

“Because each workstation has private data, each must be administered separately; maintenance is difficult to centralize. The machines are replaced every couple of years to take advantage of technological improvements, rendering the hardware obsolete often before it has been paid for. Most telling, a workstation is a large self-contained system, not specialised to any particular task, too slow and I/O-bound for fast compilation, too expensive to be used just to run a window system. For our purposes, primarily software development, it seemed that an approach based on distributed specialization rather than compromise could better address issues of cost-effectiveness, maintenance, performance, reliability, and security.”

Many of the above issues still apply today, despite the fact that higher-performance personal computers are readily available at low cost. It has been well-established that system administration and maintenance costs dominate the total cost of ownership, especially for networks of PC's. In addition, there are many large computational tasks that require resources (multiprocessors and gigabytes of memory) that cannot be afforded on every desktop.

A great deal of research went into harvesting the unused cycles on engineers' desktop machines[1]. However, the return to time-sharing a centralized pool of hardware resources is the most effective and direct solution to this problem. Thin-client computing is the modern version of the old time-sharing approach, and there have been numerous projects that focus on thin-client strategies. For example, the Plan 9 project completely redesigned the computing system with an emphasis on supporting a seamless distributed model[14]. The terminal at the desktop in Plan 9 is still a general-purpose computer running a complete virtual memory operating system, but it

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

is intended to run only programs with low resource demands.

More recently, thin-client-based system architectures have been developed, examples of which include X Terminals, Windows-Based Terminals, JavaStations, and other Network Computers. In these systems the desktop unit executes only the graphical interface of applications, which allows more sharing of computing and memory resources on the server at the expense of added network traffic in the form of display protocols, such as X[13], ICA[3], and RDP[10]. The protocols are highly optimized for specific software API's to reduce their bandwidth requirements. Thus, they typically require a nontrivial amount of processing resources and memory to maintain environment state and application data, such as font libraries and Java applets. Users are still subjected to some degree of system administration, such as network management and software upgrades.

Finally, the thinnest possible clients are dumb terminals which only know how to display raw pixels. The MaxStation from MaxSpeed Corporation is refreshed via 64Mbps dedicated connections, which can only support a resolution of 1024x768 with 8-bit pixels[9]. However, a standard 24-bit, 1280x1024 pixel display with a 76Hz refresh rate would require roughly 2.23Gbps of bandwidth. The VNC viewer, on the other hand, allows access to a user's desktop environment from any network connection by having the viewer periodically request the current state of the frame buffer[16]. While this design allows the system to scale to various network bandwidth levels, its interactive performance (even on a low-latency, high-bandwidth network) is noticeably inferior to that of a desktop workstation.

1.1 SLIM: a stateless thin-client architecture

The premise of this paper is that commodity networks are fast enough to use a low-level protocol to remotely serve graphical displays of common, GUI-based applications without any noticeable performance degradation. This leads us to take the notion of thin clients to the limit by removing all state and computation from the desktop and designing a low-level hardware- and software-independent protocol to connect all user-accessible devices to the system's computational resources over a low-cost commodity network. We refer to such thin-client architectures as SLIM (Stateless, Low-level Interface Machine) systems, and we illustrate their major components in Figure 1.

As an extreme design with the thinnest possible terminals, SLIM maximizes the advantages of thin-client architectures: resource sharing, centralized administration, and inexpensive desktop units to minimize cost per seat. In addition, the SLIM architecture has the following distinctive characteristics:

- *Statelessness.* Only transient, cached state (such as frame buffer content) is permitted in SLIM consoles; the servers maintain the true state at all times. Thus, the user is isolated from desktop

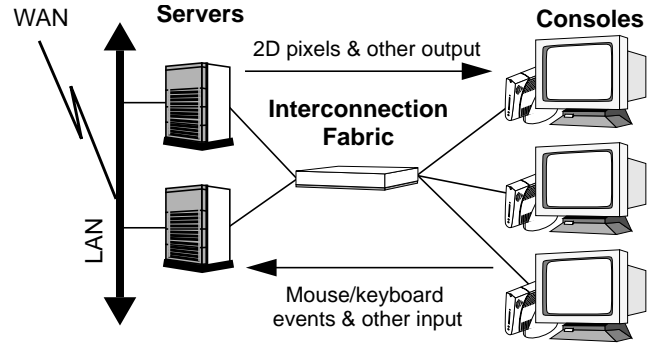


Figure 1 Major components of the SLIM architecture.

failures and may move freely between terminals. In our implementation, users can simply present a smart identification card at any desktop, and the screen is returned to the exact state at which it was left.

- *Low-level interface.* The low-level SLIM protocol is designed to move raw I/O data between the servers and consoles. Thus, a SLIM console is merely an I/O multiplexor connected to a network. The protocol can be implemented by simply redirecting server I/O to SLIM consoles over the network at the device driver level. Thus, applications on the server require no modification. SLIM consoles are not tied to a particular display API (e.g. X, Win32, Java AWT), and so applications running on different system architectures with different API's can be easily integrated onto a single desktop.
- *A fixed-function appliance.* As it merely implements a fixed, low-level protocol, a SLIM console bears more similarity to a consumer-electronics device than a computer. There is no software to administer and upgrade; its simplicity makes it amenable to a low-cost, single-chip implementation; and finally, a SLIM console requires no fan and generates no unwanted noise.

1.2 Contributions of this paper

This paper presents the SLIM design and provides a quantitative analysis of the two questions critical to the success of the SLIM architecture: (1) the extent to which today's interconnect can support graphical displays of interactive programs, and (2) the resource sharing advantage of such a system.

This paper focuses on the workgroup environment where SLIM consoles are connected to the servers via a private, commodity network carrying only SLIM protocol traffic. In particular, our experiments are performed on a dedicated 100Mbps switched ethernet. SLIM consoles are intended as replacements for desktop machines, which may then be moved to the machine room as additional servers. Installations will still require their usual complement of file servers and large servers for computational-intensive tasks. We show that SLIM systems are indistinguishable from

dedicated desktop units for a wide range of interactive, GUI-based applications and certain classes of multimedia programs. However, high-end, display-intensive applications, such as 3-D modelling, are outside the scope of the current implementation. Our results indicate that service to the home over broadband connections would have acceptable performance, but this approach would not work well in high-latency, low-bandwidth settings such as dial-up connections to the internet.

This work was done in cooperation with the research group at Sun Microsystems Laboratories that developed the recently announced Sun Ray™ 1 enterprise appliance product, which is an embodiment of the SLIM architecture. The prototype of the product was used by the developers (over 60 engineers, managers, marketing personnel, and support staff) as their only desktop computing device for the past year, and they have found their interactive experience to be indistinguishable from that of working on high-end, workstation-class machines. This paper attempts to quantify the interactive performance of the SLIM architecture scientifically, using the Sun Ray 1 prototype and product as a basis for the experiments.

While much research went into developing the methodology and benchmarks for analyzing a system's computational aspects (e.g. the SPEC benchmarks), little has been done in the way of evaluating the interactive performance of a system. Thus, part of this paper's contribution is a methodology for analyzing interactive systems. The findings of this paper are also useful for measuring the interactive performance of other systems.

Our methodology is to base the experiments on modern, highly-interactive applications such as Photoshop and Netscape, as well as streaming video and 3-D games. We measure these applications along dimensions that govern their interactive performance, such as input rates, display update rates, and bandwidth requirements. Since interactive jobs require actual users to provide input, we conducted a set of user studies to collect application profiles. As such experiments are expensive to run, we logged all the information related to their network traffic and resource utilization. In this way, we can investigate different aspects of the system by post-processing the data, rather than conducting more user studies. The data collection requires minimal resources, and thus does not perturb the results.

A difficult problem we had to address was how to measure the effect of sharing on interactive performance and to determine the level of sharing that can be supported by a system. Our solution is to utilize a yardstick application, i.e. one with well-defined characteristics. We quantify the effect of sharing on interactive performance by measuring the additional latency experienced by the yardstick application under different loading conditions.

Using this methodology, we evaluated the performance of the Sun Ray 1 implementation and compared it with the X protocol. Our results demonstrate that the SLIM protocol is capable of supporting common, GUI-based applications with no performance degradation. Surprisingly, the

low-level SLIM protocol does not translate to a greater bandwidth demand when compared to X. This is because X was found to only optimize applications with low-end bandwidth requirements. A Sun Ray 1 console can display a 720x480 video clip at 20Hz and allows Quake to be played at a resolution of 480x360. The server, and not the bandwidth to the console, turns out to be the bottleneck for these applications. Finally, our experiments show a high potential for resource sharing. Depending on the application class, anywhere from 10 to 36 active users can share a 300MHz processor without any noticeable degradation of interactive performance. While the network is often regarded as the major performance bottleneck in thin-client systems, it can support more active users (by an order of magnitude) than the processor and memory.

Centralizing all computing resources is not without its disadvantages, and there are still many opportunities for further research to combine the best of centralized and distributed computing. Distributed computing offers users isolated performance guarantees, higher levels of security and privacy, user-level customization of the system, higher tolerance to server and network failures, and off-line computing. These topics are beyond the scope of this paper.

The rest of this paper is organized as follows. Section 2 provides an overview of the SLIM architecture. Section 3 describes our experimental methodology, and Sections 4–6 contain the experiments and their results. We begin with a stand-alone assessment of each architectural component, followed by a characterization of interactive applications, including input, display, and communication requirements. Then, we explore the effects of resource sharing on interactive performance. Section 7 discusses multimedia support, and Section 8 compares the SLIM approach with related systems. We summarize our findings and discuss future work in Section 9.

2 The SLIM architecture and implementation

In this section we describe the design and rationale for each of the components in the SLIM architecture: the interconnect fabric, the SLIM protocol, the consoles, and finally the servers. In addition, we present the details of the Sun Ray 1 implementation[12][18] on which our experiments are based.

2.1 Interconnection fabric

In the SLIM system, raw display updates are transmitted over the network to display devices. Thus, the SLIM interconnection fabric (IF) is the centerpiece of the architecture, and yet it is perhaps the simplest component. It is defined to be a private communication medium with dedicated connections between the desktop units and the servers. Such functionality can easily be provided by modern, off-the-shelf networking hardware. The Sun Ray 1 supports a 10/100 Base-T ethernet connection, and we used switched, full-duplex 100Mbps ethernet with Foundry FastIron Workgroup switches in our experiments.

The IF is defined in this manner so that the system can make response time guarantees and thereby provide high

interactive performance, regardless of loading conditions. Although it is possible to share bandwidth on the IF, care must be taken to ensure that response time does not suffer as a result. In addition, there is no need to provide higher level (e.g., Layer 3 and above) services on the IF, nor the complex management typically provided on LAN's.

2.2 The SLIM protocol

The SLIM protocol takes advantage of the fact that the display tends to change in response to human input, which is quite slow. Thus, instead of refreshing the monitor across the IF, we achieve considerable bandwidth savings by refreshing the display from a local frame buffer and transmitting only pixel updates. Although the display is refreshed locally, it represents only soft state which may be overwritten at any time. The full, persistent contents of the frame buffer are maintained at the server.

The protocol consists of a small number of messages for communicating status between desktop and server, passing keyboard and mouse state, transporting audio data, and updating the display. The display commands are outlined in Table 1. They compress pixel data by taking advantage of the redundancy commonly found in the pixel values generated by modern applications. For example, the BITMAP command is useful for encoding text windows.

Command Type	Description
SET	Set literal pixel values of a rectangular region.
BITMAP	Expand a bitmap to fill a rectangular region with a (foreground) color where the bitmap contains 1's and another (background) color where the bitmap contains 0's.
FILL	Fill a rectangular region with one pixel value.
COPY	Copy a rectangular region of the frame buffer to another location.
CSCS	Color-space convert rectangular region from YUV to RGB with optional bilinear scaling.

Table 1 SLIM protocol display commands.

In contrast, most other remote display protocols (e.g. X[13] and ICA[3]) send high-level commands (e.g. "display a character with a given font, using a specific graphics context"), which require considerable amounts of state and computation on the desktop unit. By encoding raw pixel values, the SLIM protocol represents the lowest common denominator of all rendering API's. To take advantage of this protocol, applications can be ported by simply changing the device drivers in rendering libraries. For example, we have implemented a virtual device driver for the X-server, and all X applications can run unchanged.

Although the SLIM protocol is intended to be used primarily by low-level device drivers, applications may utilize a software library to transmit display operations in a domain-specific manner. For example, we have implemented a video playback utility and a 3-D game by converting each display frame to YUV format and using the CSCS command to transmit the data directly to the console. Although these applications could function as

regular X clients, this approach is more efficient and produces higher quality results. The library itself contains packet sequencing commands, methods for obtaining geometry information and input events, as well as bandwidth and session management routines.

Another feature of the SLIM protocol is that it does not require reliable, sequenced packet delivery, whereas other remote display protocols (e.g. X[13], ICA[3], and VNC[16]) are typically built on top of a reliable transport mechanism. All SLIM protocol messages contain unique identifiers and can be replayed with no ill effects. Since the preferred IF implementation is a dedicated connection between console and server, errors and out-of-order packets are uncommon. Thus, systems using the SLIM protocol can be highly optimized with respect to error recovery. In the Sun Ray 1 implementation, SLIM protocol commands are transmitted via UDP/IP between the servers and consoles, and its application-specific error recovery scheme allows for more efficient recovery than packet replay and avoids "stop and wait" protocols.

2.3 SLIM consoles

A SLIM console is simply a dumb frame buffer. It receives display primitives, decodes them and hands off the pixels to the graphics controller. This allows us to make the desktop unit a cheap, interchangeable, fixed-function device. It is completely stateless and runs neither an operating system nor any applications, and the performance a user experiences is decoupled from the desktop hardware.

The Sun Ray 1 enterprise appliance includes four major hardware components: CPU, network interface, frame buffer, and peripheral I/O. It utilizes a low-cost, 100MHz microSPARC-IIep processor with 8MB of memory, of which only 2MB are used. The network interface is a standard 10/100Mbps ethernet controller, and the frame buffer controller is implemented with the ATI Rage 128 chip (with support for resolutions up to 1280x1024 at a 76Hz refresh rate and 24-bit pixels). A four-port USB hub is included for attaching peripherals, including keyboard and mouse. The firmware on the console simply coordinates the activity between the components, i.e. it moves data between the network and appropriate devices.

2.4 SLIM servers

In the SLIM architecture, all processing is performed on a set of server machines which may include a variety of architectures running any operating system with any application software. These servers neither replace nor remove the standard set of servers common today, e.g. file servers, print servers. They merely consolidate and reduce the computing resources that were previously on desktops.

We only need to add to these servers a small set of system services: daemons for authentication, session management, and remote device management. The authentication manager is responsible for verifying the identity of desktop users; the session manager redirects the I/O for a user's session to the appropriate console; and the remote device manager handles peripherals attached to the system via a SLIM console.

3 Evaluation methodology

In this section we present our evaluation methodology and compare it with some related techniques.

3.1 Our approach

Our performance analysis of the system consists of four steps: (1) establish the basic performance of each component (the IF, server, and consoles) of the SLIM system using stand-alone tests, (2) characterize GUI-based applications by their communication requirements, evaluate how well the SLIM architecture and protocol support such applications, and compare the result with that of the X protocol, (3) evaluate the opportunity for sharing offered by the SLIM architecture by measuring its interactive performance under shared load and also by obtaining load profiles of actual installations of the system, and (4) measure the limit of SLIM by evaluating its performance on multimedia applications.

Category	Application
Image Processing	Adobe Photoshop 3.0
Web Browsing	Netscape Communicator 4.02
Word Processing	Adobe Frame Maker 5.5
PIM	Personal Information Management tools (e.g. e-mail, calendar, report forms)
Streaming Video	MPEG-II decoder and live video player
3-D Games	Quake from id Software

Table 2 Benchmark applications.

To ensure that our results would have the widest applicability, we have chosen a set of commonly-used, GUI-based applications, as listed in Table 2. The set covers a wide range of resource and display demands. We instrumented the SLIM protocol driver used by these applications so as to measure the I/O properties of their human interface. Measurements were taken by having a group of 50 people separately run the applications for at least ten minutes on the Sun Ray 1 prototypes with a very lightly loaded server. The resulting profiles are thus indicative of real-world use, and they reflect performance for individual users in isolation. Depending on the nature of the experiments and machine availability, we employed a variety of test configurations during our study, and we summarize them in Table 3.

To characterize interactive performance under shared load, we use an indirect method. We create a highly interactive application with a fixed and regular resource requirement, and instrument it to measure the response time its user experiences. We run this application with different system loads and use its measured latency to gauge the system's responsiveness. To simulate multiple active users we use load generators to "play back" the resource profiles that were recorded during the user studies. This approach has the advantage of providing a consistent metric across systems and applications.

Experiment	Desktop Unit	Server				
		Model	Processor(s)	RAM	Swap	OS
IF response time	Sun Ray 1 prototype	Ultra 2 workstation	1 296MHz UltraSPARC-II	512MB	1GB	Solaris 2.6
x11perf	Sun Ray 1	Enterprise E4500	8 336MHz UltraSPARC-II	6GB	13GB	Solaris 2.7
User studies	Sun Ray 1 prototype	Ultra 2 workstation	2 296MHz UltraSPARC-II	512MB	1GB	Solaris 2.6
Processor sharing	Simulated	Enterprise E4500	10 296MHz UltraSPARC-II	4GB	4.5GB	Solaris 2.7
IF sharing	Same as server	Ultra 2 workstation	2 296MHz UltraSPARC-II	512MB	1GB	Solaris 2.7
Multimedia tests	Sun Ray 1	Enterprise E4500	8 336MHz UltraSPARC-II	6GB	13GB	Solaris 2.7

Table 3 Hardware configurations for empirical studies. In all cases the interconnection fabric was 100Mbps ethernet with Foundry FastIron Workgroup switches. All machine hardware is manufactured by Sun Microsystems, Inc.

3.2 Related techniques

Endo *et al.* have also attempted to measure the interactive performance of a general system[7]. They argue for the use of interactive event latency (i.e. response time) metrics to evaluate system performance, which is similar to our own measurement goals. They instrumented Windows NT versions 3.51 and 4.0 as well as Windows 95 and recorded processing latency for each input event for a set of stand-alone applications. The main difficulty they experienced is that it is impossible to know exactly how much processing time is needed to service an event without instrumenting each application. So, they had to rely on heuristics, which prevented them from studying more than one simultaneously active application. Our indirect benchmark technique addresses this problem. In addition, their approach is aimed at quantifying the interactive performance of stand-alone desktop systems, such as a PC, whereas we are interested in evaluating remote display systems which include a network component.

Another related evaluation technique is capacity planning to determine the number of users a server can support[4][19][21]. Such studies are useful for making resource allocation decisions, but they do not adequately assess interactive performance. The major problem with this technique is that it tends to take a long-term, coarse-grained view of system activity, thereby losing information on interactive performance. In addition, user activity is modelled in one of three ways, all of which have their drawbacks. First, canned scripts or macros are frequently used, but they have no interactive delays and thus merely measure throughput. Second, recorded scripts with interactive delays are used to emulate users, but timing dependencies between the client (e.g. application) and server (e.g. the X-server) make such emulations only valid when the system is in underload. Finally, queueing theory is used to model users based on average resource profiles. While this approach has the advantage of being able to leverage well-known mathematics techniques to make precise predictions, it cannot model the system in

overload. In all cases, our indirect benchmark approach provides a better assessment.

4 Performance of SLIM components

We begin our analysis of the SLIM system with a stand-alone characterization of each major component in the Sun Ray 1 implementation. The results are summarized in Table 4 and Table 5.

Benchmark	Result
Response time over a 100Mbps switched IF	550 μ s
x11perf / Xmark93	3.834
x11perf / Xmark93 — no display data sent on IF	7.505

Table 4 Stand-alone benchmarks for the Sun Ray 1.

4.1 Response time over the IF

Response time is a major concern for any remote display architecture. Other approaches, such as X terminals and WinTerms, process input events locally on the desktop. A SLIM system, however, must transmit all input events to a server and then wait for it to send back the results. Humans begin to notice delays when latency enters the 50–150ms range[17]. To provide a high-quality interactive experience, it is crucial that response time remain within these bounds.

Typical local area networks are shared and carry a variety of traffic. Depending on the current load, latency can be highly variable, and it is difficult to make response time guarantees in such an environment. We ensure that latency remains within the prescribed limits by using a switched, private network for the interconnection fabric. In the SLIM architecture each desktop unit has a direct connection to the servers, and only SLIM protocol traffic is carried over it. Since there is no interference or other outside effects, the added latency is exactly the round-trip delay over the interconnect.

To demonstrate the effectiveness of the SLIM approach, we wrote a simple server application which accepts keystrokes from a SLIM console and responds by sending characters to the console. We measured the total elapsed time from the instant a keystroke is generated at the SLIM console to the point at which rendering is complete and the pixels are guaranteed to be on the display.

With the test setup shown in Table 3, the elapsed time was found to be 550 μ s. In contrast, if we type the characters in an Emacs editor, as opposed to our simple application, the delay is found to be 3.83ms. Clearly, in such an environment the communication medium is a negligible source of latency, and the end result is that the response time which users experience is effectively dependent on the processing time on the server. Users are, therefore, unable to distinguish between the response times observed from a remote SLIM console and a monitor connected directly to the server itself.

It is interesting to note that on a stand-alone workstation, the service time is much higher at 30ms[11]. This is most likely due to buffering of input in the keyboard device driver so that block data transfers can be

used. Such an optimization is not required for network I/O since it can be memory-mapped directly into a user’s address space. Thus, the SLIM implementation can actually be more responsive than a dedicated workstation.

4.2 Server graphics performance

A key factor in the display performance of any thin-client system is how well the server can generate the protocol. In the Sun Ray 1 implementation of the SLIM architecture, the X-server is responsible for translating between the X and SLIM protocols and transmitting display update commands over the IF. To analyze the graphics performance of a SLIM server, we ran the SPEC GPC x11perf benchmark.

The x11perf benchmark is older and no longer used, but it provides a useful indication of performance. We ran the benchmark on the system shown in Table 3, and we used the Xmark93 script to generate a single figure of merit from the results. The X-server achieved a rating of 3.834. Although the X-server must interpret the commands and transmit display data to the console, its Xmark value is quite comparable to values for X terminals, which receive commands and interpret them locally. For example, the NEC HMX reported a value of 4.20 to SPEC. If we eliminate the actual transmission of SLIM protocol commands, the Xmark performance rating of the X-server improves to 7.505, indicating that network operations represent a significant performance factor for this benchmark.

4.3 Protocol processing on the desktop

The performance-limiting factor on a SLIM console is the highest sustained rate at which it can process protocol commands. To determine this limit, we created a server application which transmits sequences of protocol commands to a Sun Ray 1 console up to the point where the terminal cannot process the transmitted commands and begins to drop them. Once these sustained rates were determined for various command types and sizes, we calculated the protocol processing cost in terms of a constant overhead per command as well as an incremental cost per pixel. The results are presented in Table 5.

Protocol Command	Startup Cost	Cost per Pixel
SET	5000 ns	270 ns
BITMAP	11080 ns	22 ns
FILL	5000 ns	2 ns
COPY	5000 ns	10 ns
CSCS (16 bits/pixel)	24000 ns	205 ns
CSCS (12 bits/pixel)	24000 ns	193 ns
CSCS (8 bits/pixel)	24000 ns	178 ns
CSCS (5 bits/pixel)	24000 ns	150 ns

Table 5 Sun Ray 1 protocol processing costs.

The SET command has a fairly high incremental cost because pixels must be expanded from packed 3-byte

format to 4-byte quantities suitable for the frame buffer. On the other hand, the FILL and COPY commands are very inexpensive. The BITMAP command has a high start-up cost in order to set the state of the graphics card and must be amortized over a large number of pixels to be of the most benefit. The color space convert and scale (CSCS) command not only has a high start-up cost in order to configure the graphics controller, but it also has a fairly high cost per pixel. Still, it is more efficient than the SET command when fairly large numbers of pixels are sent, which is usually the case for applications that utilize image or video data. Also, it provides a significant reduction in bandwidth, making it worthwhile despite the high processing overhead.

5 Characterization of interactive applications

The SLIM architecture is based partly on the observation that the display is typically updated in response to user activity. We expect humans to have a relatively slow rate of interaction, and so the display should be quiescent much of the time, thus requiring modest resources to provide good performance. To verify this observation, we analyzed the interaction and display patterns of modern applications during active use and demonstrate that SLIM can meet their demands. Most of the results in this section apply to any architecture, not just SLIM.

To characterize I/O requirements of interactive programs, we use the GUI-based benchmark applications listed in Table 2, i.e. Photoshop, Netscape, Frame Maker, and PIM. As mentioned in Section 3.1, we employ user trials to gather data during real-world use of the applications. The data were collected on two identical systems, as listed in Table 3. Each server had a 1Gbps uplink from its IF and ten terminals attached to it. The servers were completely underloaded at all times, and users had the impression that they were working on a dedicated workstation. Thus, the gathered traces are indicative of stand-alone operation.

To obtain these results, we instrumented the SLIM protocol driver in our implementation of the X-server. All X and SLIM protocol events were recorded and timestamped. The logging overhead is not measurably significant and therefore does not perturb the results. These logs enable us to determine input and display characteristics as well as network traffic requirements.

This section is organized as follows. We begin by analyzing the rates of human interaction, followed by a characterization of the sizes of display updates induced by that interaction. Next, we assess the compression efficiency of the SLIM protocol as well as its ability to scale down to lower bandwidths. We continue with an analysis of the protocol processing costs on both the console and server, and we conclude with a comparison of the overall bandwidth requirements of the benchmark applications under the SLIM and X protocols.

5.1 Human input rates

Based on the data gathered in the user studies, we first analyzed typical rates of human interaction. We defined

input events to be keystrokes and mouse clicks. All input events are transmitted to the server for processing, and mouse motion events do cause some display updates for these applications (e.g. rubberbanding). However, such motion events are demarcated by a button press and release, which serve to toggle the drawing mode. In these cases the motion-induced display updates are more naturally regarded as a single update caused by the initial button press. For each of the GUI-based benchmark application sessions, we calculated the frequency of input events, and Figure 2 presents the results.

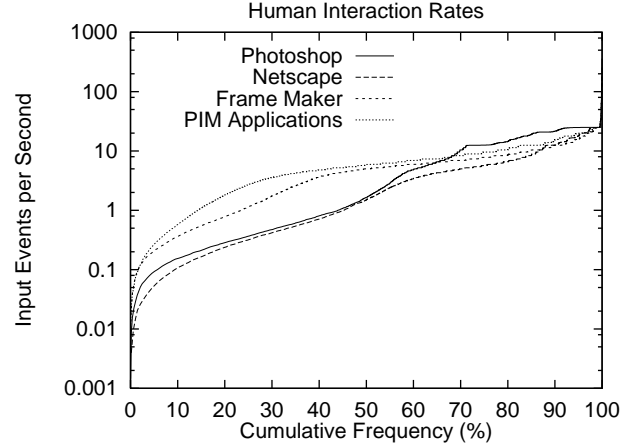


Figure 2 Cumulative distributions of user input event frequency. Input events are defined as keystrokes and mouse clicks. Histogram bucket size is 0.005 events/sec.

The most important thing to note in this graph is that human input rates are typically much lower than monitor refresh rates (which are at least 60Hz). In particular, we see that for each application, less than 1% of input events occur with frequency greater than 28Hz. This is important because it indicates an application-independent “upper bound” on the rate at which humans generate input. Also, roughly 70% of all events occur at low frequencies, i.e. less than 10Hz (or more than 100ms between events). Despite the diversity of these applications, the interaction patterns are quite similar. Even so, Netscape and Photoshop tend to be much less interactive than Frame Maker or PIM, as indicated by their substantially larger percentage of events occurring at least one second apart.

5.2 Pixel update rates

Next, we consider the pixel changes induced by human input. Correlating input events to display updates can only be done by instrumenting all applications. However, source code is frequently unavailable, and adding correct instrumentation to a complex program, such as Netscape, is prone to errors. Therefore, we use the following heuristic to estimate the correlation between input events and display updates: all pixel changes that occur between two input events are considered to be induced by the first event. Although this is not true in all cases, it provides a close enough approximation for the applications in this experiment over the course of the ten-minute user sessions.

To obtain these values, we sum the number of pixels affected by SLIM display primitives recorded between the events, and we present the results in Figure 3.

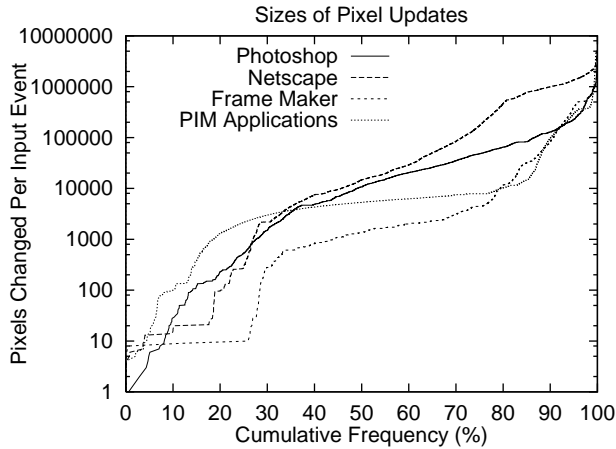


Figure 3 Cumulative distributions of pixels changed per user input event. Input events are defined as keystrokes and mouse clicks. Histogram bucket size is 1 pixel.

This graph depicts the cumulative distributions of pixels altered per event. All displays used in these experiments had a resolution of 1280 by 1024 pixels (i.e. 1.25Mpixels). The important thing to note is that display updates typically affect only a small fraction of the full display area. For example, nearly 50% of all input events for any application cause less than 10Kpixels to be modified. Further, only 20% of Frame Maker or PIM events affect more than 10Kpixels, and only 30% of Netscape or Photoshop events affect more than 50Kpixels. Also, note that Netscape is more demanding than Photoshop, but as we will later see, its compressed bandwidth requirement is much lower.

This result, coupled with the rates of human interaction, indicate that (for this class of applications) the contents of the display change only slowly and moderately over time. This is important because it has a significant effect on the SLIM architecture, namely that even as display requirements increase over time, human input rates (and therefore required update rates) are unlikely to change. Thus, it is unlikely that SLIM consoles would need to be upgraded for users of these types of applications.

5.3 Compressed pixel update rates

To assess the effectiveness of the SLIM pixel encoding techniques, we analyze the compression factor afforded by each of the SLIM display commands. In addition, we examine the sizes and network transmission delays of display updates once they have been compressed with the SLIM protocol.

In Figure 4 we present the data reduction afforded by each of the SLIM protocol display commands for the GUI-based benchmark applications. The important observation to make is that the protocol provides a factor of 2 compression for Photoshop and a factor of 10 or more for all other applications. The FILL command is extremely

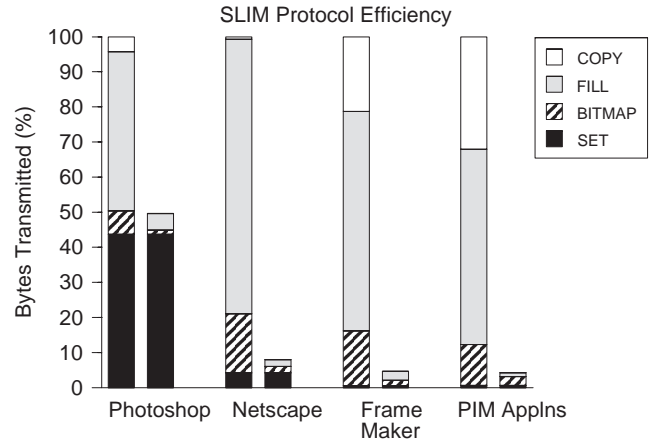


Figure 4 Efficiency of SLIM protocol display commands. The left-hand bar in each set depicts uncompressed data, and the right-hand bar depicts use of the SLIM protocol. CSCS is not used in these benchmark applications.

effective, reducing bandwidth by 40%–75% across the applications. The BITMAP and COPY commands are utilized to different extents, but they also provide substantial compression. PIM and Frame Maker enjoy particularly large savings because they employ a great deal of bicolor text and scrolling. The SET command represents pixel data which cannot be compressed via the SLIM protocol, but only Photoshop has a substantial portion of such commands. Thus, we can see that the protocol has done a good job of compressing the pixel data where possible.

In Figure 5 we present the cumulative distributions of the amount of SLIM protocol data transmitted per user input event for the GUI-based benchmark applications. From this graph we can see that (over a 100Mbps interconnection fabric) the transmission delays will be quite small. For example, even a large update of 50KB incurs only 3.8ms of transmission delay. More specifically, only 25% of Photoshop and Netscape events require more than 10KB to encode the display update, and only 5% require more than 50KB. Frame Maker and PIM have even lower requirements; only 17% of events require more than 1KB for display updates, and only 2% require more than 10KB.

5.4 Scalability of the SLIM protocol

Although the SLIM protocol can comfortably accommodate the GUI-based benchmark applications with a low-latency, 100Mbps interconnection fabric, it is useful to consider how well it would operate over lower bandwidth connections. To obtain a sense of how interactive performance would be affected, we modified the X-server to restrict the bandwidth it could utilize, and then we ran our normal window sessions in the constrained setting.

At 10Mbps users could not distinguish any difference from operating at 100Mbps. To simulate a high-speed home connection, such as a cable modem or DSL, we

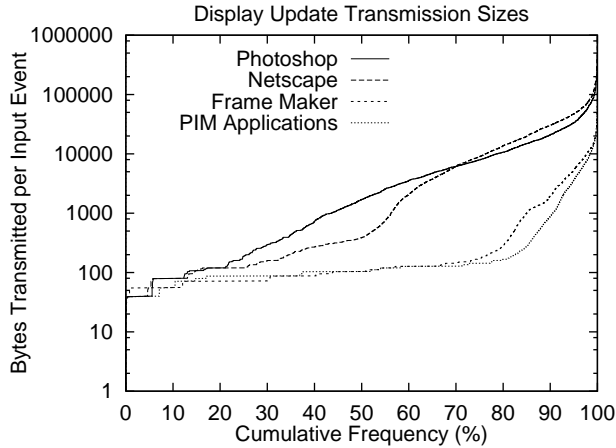


Figure 5 Cumulative distributions of SLIM protocol data transmitted per user input event. Input events are defined as keystrokes and mouse clicks, and the histogram bucket size is 1 byte.

tested the system with 1–2Mbps bandwidth limits. Performance was quite good, with only occasional hiccups when large regions had to be displayed. Finally, to simulate low-speed home connections such as ISDN or telephone modems, we tested the system with 56–128Kbps bandwidth limits. We found performance to be extremely poor. It is possible to accomplish tasks such as reading e-mail or editing text, but the experience is painful. Of course, the SLIM protocol was not designed for such low-bandwidth connections, and optimizations like header compression and batching of command packets could have a dramatic effect.

To quantify this experience, we used the protocol logs from the resource profiles mentioned above and simulated transmitting the packets over lower bandwidth connections. We chose Netscape as a representative example and recorded the packet transmission delays in excess of the delays experienced at 100Mbps. Figure 6 depicts the cumulative distributions for a variety of bandwidth levels.

At 10Mbps the added packet delays are always less than 5ms, which is well below the 50–150ms threshold of human tolerance. The added packet delays at 1–2Mbps approach 50ms, entering the realm where humans would notice but consider acceptable. However, at 56–128Kbps we see a sharp increase in packet delays beyond 100ms, indicating that response time would be unacceptably high.

5.5 SLIM protocol processing costs

A key goal of SLIM is to minimize the resources on the desktop. However, reading protocol commands from the network and decoding them for display incurs processing overhead. The CPU is mostly idle until a display command arrives, followed by a burst of activity which entirely consumes the processor. To analyze this cost, we monitored the service time on the Sun Ray 1 prototype console for the SLIM protocol commands sent as part of each display update during the user studies described above. In Figure 7

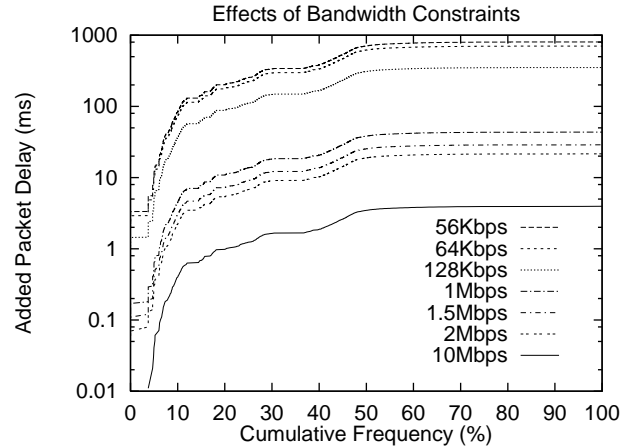


Figure 6 Cumulative distributions of added packet delays for Netscape traces of SLIM protocol commands captured at 100Mbps and retransmitted on simulated networks with lower bandwidths. Bandwidth is averaged over 50ms intervals, and the histogram bucket size is 0.01ms.

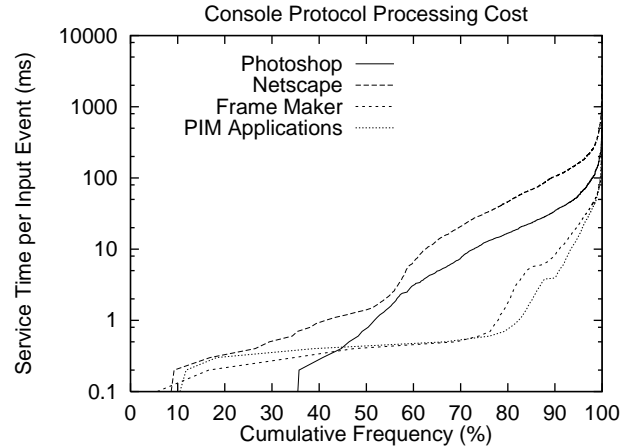


Figure 7 Cumulative distributions of display update service times on the Sun Ray 1 prototype console. Histogram bucket size is 0.1ms.

we report the cumulative distributions of these display latencies.

The important thing to note in this graph is that response time is almost always below the threshold of perception, i.e. in 80% of all cases service time is below 50ms. A small fraction of service times exceed 100ms and may be noticeable, but note from Figure 3 that there are correspondingly large display updates, for which human tolerance is typically higher.

The total service time for a display update is the combination of these console service times and the network transmission delays discussed in Section 5.3. As we can see, the network adds little extra cost, and the total service time is quite short. Thus, we conclude that the simple, low-performance microSPARC-IIep is more than adequate to meet the demands of these applications. In fact, an even lighter weight implementation could possibly be used, and

a combined processor and memory implementation of the SLIM console could provide exceptional performance at an extremely low cost point.

Finally, although we have focussed on the cost of decoding protocol messages on the desktop, the server also incurs overhead due to encoding pixel values and generating protocol messages. However, this overhead is extremely low, accounting for a mere 1.7% of the X-server's total execution time when servicing the benchmark applications. Thus, we conclude that the added cost of protocol processing to send the pixel updates is marginal compared to the savings in bandwidth it affords.

5.6 Average bandwidth requirements

To summarize the results from this section and put them in perspective in relation to other thin-client architectures, we calculated the network bandwidth requirements for our set of benchmark applications under three protocols. In Figure 8 we present the average network bandwidth for each application using the X protocol, the SLIM protocol, and a simple protocol in which all the changed pixels are transmitted (labelled "Raw Pixels" in the chart).

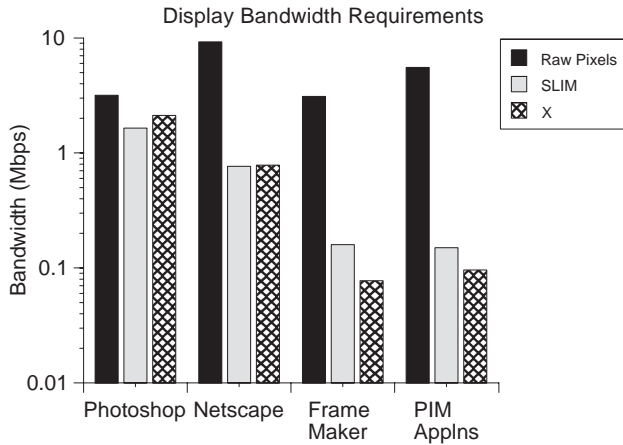


Figure 8 Average bandwidth consumed by the interactive benchmark applications under the X, SLIM, and raw pixel update protocols.

There are two important points to observe in this graph. First, it is quite interesting to note that the X and SLIM protocols have similar bandwidth requirements. X performs slightly better on the Frame Maker and PIM applications which were, in fact, the classes of programs for which X was optimized. However, this is insignificant because the bandwidth requirements of these programs are so low. On the other hand, Photoshop and Netscape represent a new class of applications in which image display is the common operation, and they require an order of magnitude more bandwidth. In these cases, SLIM outperforms X, and the absolute size of the bandwidth differential is substantially larger than for the other applications. We thus conclude that the SLIM protocol is at least competitive with X in terms of bandwidth requirements while providing a much simpler, lower-level interface which requires substantially less computing resources.

The second item to note is that the overall bandwidth requirements are quite small. We have already seen that network-induced delays do not adversely affect interactive performance, and this result demonstrates that network occupancy is also low. This implies that there is substantial opportunity to share the interconnection fabric which we discuss in the following section.

6 Interactive performance for multiple users

The key advantage of SLIM and other thin-client architectures is the savings provided by sharing computational resources. In this section we explore how interactive performance changes when there is interference from other simultaneously active users.

Many engineering workgroups have servers configured to support the requirements of their demanding computational activities, and we will continue to have these compute servers for executing long-running jobs in a SLIM system. Here we are interested mainly in the sharing of machines for running those applications typically executed on our desktops.

6.1 Sharing the server processor

We want to determine how many active users a server can support while still providing good interactive performance. As discussed in Section 3, this is difficult to determine and is a subjective assessment that may be different for each individual. Because human perception is relatively slow, users can tolerate fairly large response times. Thus, a processor can be oversubscribed while still providing good interactive performance. To account for this type of scenario, we must model the system in overload. However, large-scale user studies are impractical, and script-based techniques will fail due to timing constraints that cannot be met.

Our approach is to use a load generator to simulate active users. In addition to the traffic logs mentioned above, we supply the load generator with detailed per-process resource usage information, including CPU and memory utilization. These resource profiles were collected during the user studies with a tool that samples the number of CPU cycles consumed and physical memory occupied by each process at five-second intervals. This tool is similar to that presented in [20], and its overhead per analyzed process was measured and found to be insignificant. The load generator reads a resource usage profile and mimics its consumption of CPU, memory, network, disk and other resources over time. It does not replay the recorded X commands, SLIM commands, or other high-level operations. It merely utilizes the same quantity of resources in each time interval as the original application did. In this way, we can produce the correct level of background load even when the resources are oversubscribed.

While the simulated user loads are running, we use an application with well-known properties and requirements as a yardstick to gauge interactive performance. This application repeatedly consumes 30ms of dedicated CPU time to simulate event processing, followed by 150ms of "think time." We defined the application in this way so that

it would be more demanding than any other interactive application. In particular, it requires nearly 17% of the server, which is greater than the average CPU requirement for Photoshop (14%), Netscape (13%), Frame Maker (8%), and PIM (3%). Also, the interrupt rate is equivalent to a fast typist and is well beyond the sustained rates for any of our benchmarks. When the system becomes overutilized, the amount of real time needed to process the simulated event will exceed 30ms. We measure the value of this added delay as we increase the number of simulated active users.

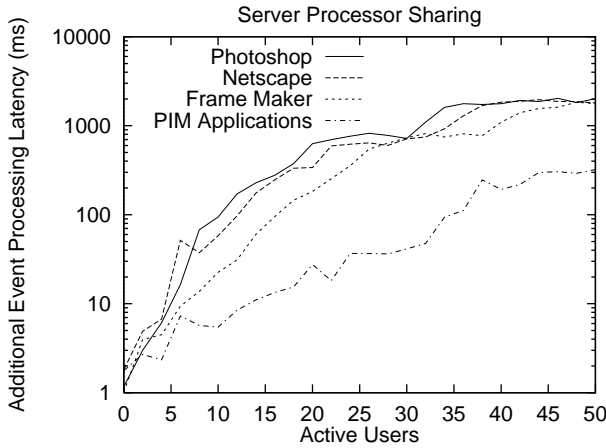


Figure 9 Average latency added to 30ms processing time for events occurring at 150ms intervals as the number of active users increases (1 active CPU). User processor loads are generated with a trace-driven simulator which “plays back” previously recorded resource usage profiles.

The experimental setup is listed in Table 3, and the server had a single processor enabled. We modelled both CPU and memory loads for the active users, and the results are presented in Figure 9. To put them in perspective, we ran the GUI-based, benchmark applications ourselves under the experimental conditions and found that when the added delay on the yardstick application reached around 100ms, interactive performance was noticeably poor. As we can see from the average CPU requirements listed above, the processor on the server is significantly oversubscribed at this point. This is important because it demonstrates that a system can provide good performance for interactive applications even when the processor is fully utilized. In particular, we could tolerate up to about 10-12 simultaneously active Photoshop users, 12-14 Netscape users, 16-18 Frame Maker users, or 34-36 PIM users on the server (in addition to the yardstick application). Of course, other people will have different tolerance limits and application mixes, but this experiment helps quantify how well a system will perform in a multi-user environment.

Another issue we wanted to investigate was how well the system would scale as processors were added to the server. As the number of CPU’s increases, contention for shared caches and memory bus bandwidth also increases, potentially reducing the scalability of the system. So, we chose Netscape as a representative application and used the

same setup described above. We varied the number of active CPU’s in the system from 1 to 8, with a corresponding increase in the number of active users. Figure 10 presents the results, which we normalized by reporting the number of active users per processor.

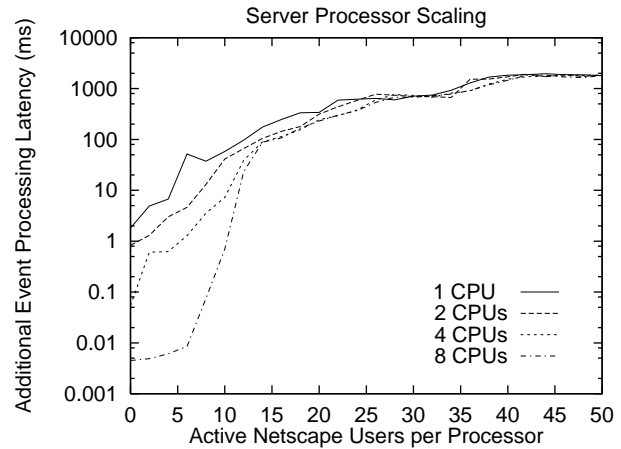


Figure 10 Average latency added to 30ms processing time for events occurring at 150ms intervals as the number of active Netscape users increases (1-8 active CPU’s). User processor loads are generated with a trace-driven simulator which “plays back” previously recorded resource usage profiles.

We can see from the graph that the system scales quite well with no obvious contention effects. In addition, note that when the system has a relatively smaller number of active users per CPU, configurations with more processors outperform those with less. The reason is that a multiprocessor system is better able to find a free CPU when one is required.

6.2 Sharing the interconnection fabric

Although the preferred embodiment of the SLIM interconnection fabric is a dedicated private network, significant reductions in cost are possible by sharing bandwidth on the IF. To analyze the cost of sharing the SLIM interconnection fabric, we again used the load generator discussed above to play back the network portion of the resource profiles in order to simulate active users on the IF. While the background traffic is being generated, we used an application with well-known properties as a yardstick to gauge interactive performance. This application simulates a highly interactive user with sizeable display updates by repeatedly sending a 64B command packet to the server followed by a 1200B response and then 150ms of “think time.” We measure the average round-trip packet delay as the number of active users is increased.

The experimental configuration is listed in Table 3. We used three workstations: one to act as a SLIM console, one to serve as a sink for background SLIM traffic, and one to act as a server. The machine which played the role of an active console executed the application described above and recorded round-trip packet delays for the network traffic. The machine acting as a server used the load

generator to send background SLIM traffic to the sink host and responded to incoming packets from the simulated console. All workstations were connected to a network switch. In this way, the link to the server was shared by both the measured and background traffic, making it the point of contention in the system. Figure 11 presents the results for our benchmark applications.

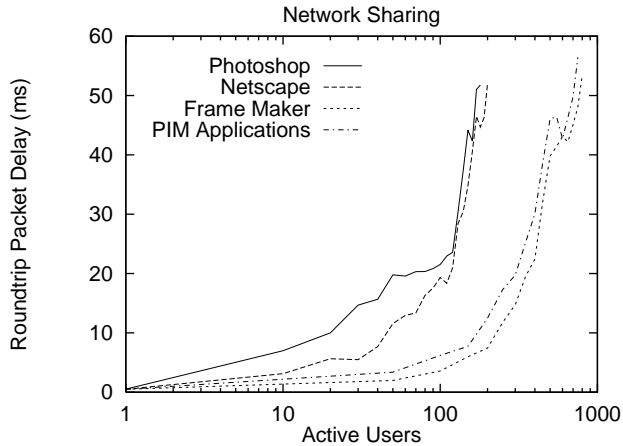


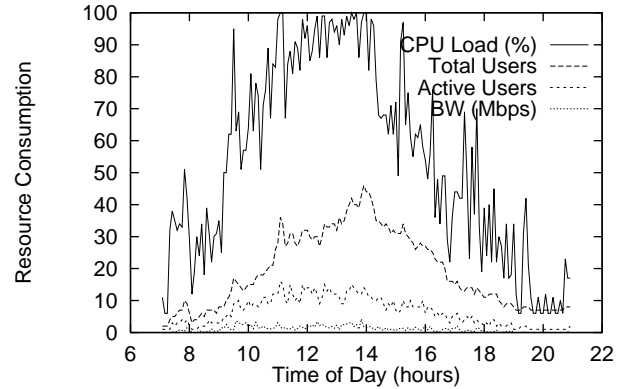
Figure 11 Network round-trip latency for 64B upstream packet followed by 1200B downstream packet. User traffic loads are generated with a trace-driven simulator which “plays back” previously recorded resource usage profiles.

To put these results in perspective, we again ran the benchmark applications for ourselves under the experimental conditions. We found the system to be quite usable until packet delays for the test application hit about 30ms, at which point response time suffered greatly and packet loss became a problem. Thus, we could tolerate up to about 130–140 simultaneously active Photoshop or Netscape users, or about 400–450 Frame Maker or PIM users on a shared network. This is interesting because it demonstrates that the network is a less critical resource than the processor, memory, or swap space on the server.

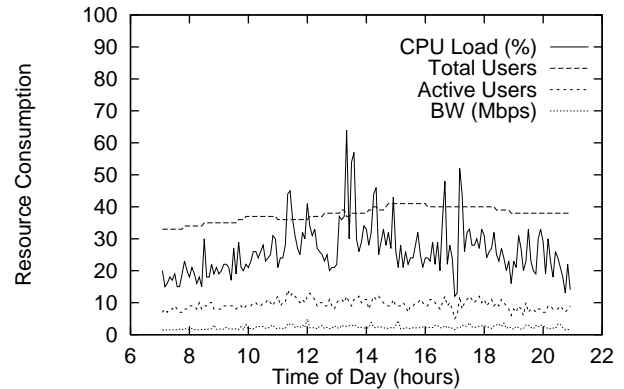
6.3 Case studies

Since the Sun Ray 1 is a commercial product, we have had the opportunity to monitor its use in real-world settings. At two installations we used standard resource monitoring tools (e.g. ps, netstat, vmstat) to collect performance data continuously over the course of several days. Snapshots of aggregate CPU load, aggregate network bandwidth usage, and number of active users were taken every 10 seconds throughout the day. For each site, Figure 12 presents the day-long profile which has the highest average processor load and the highest number of active users. We report the maximum value recorded in each five-minute period.

The first graph presents a load profile from a university lab which was previously supported by a collection of X terminals. Students work on programming assignments and other normal computing tasks, and major applications include MatLab, StarOffice, Netscape, e-mail and compilers. The server is a Sun Enterprise E250 with two 400MHz UltraSPARC-II CPU’s, 2GB of RAM, a 1Gbps



a) University Lab



b) Product Development Team

Figure 12 Day-long plot of aggregate CPU utilization, network bandwidth, and active users for real-world installations of the Sun Ray 1 system.

link to the IF, 13GB of swap space, and the Solaris 2.7 operating system. There are 50 terminals attached to the server, and at the busiest part of the day many are in use (Total Users in the graph). However, far fewer users are actively running jobs on the system (Active Users in the graph). Both processors reach full utilization during peak periods. However, aggregate network load is below 5Mbps, making the 1Gbps uplink massive overkill.

The second graph presents a load profile from our computer product development team. Engineers and other staff use CAD tools for hardware design, text editors and compilers for software development, as well as Netscape, StarOffice, Citrix MetaFrame, e-mail, calendar, etc. for office productivity. The server is a Sun Enterprise E4500 with eight 336MHz UltraSPARC-II CPU’s, 6GB of RAM, a 1Gbps link to the IF, 13GB of swap space., and the Solaris 2.7 operating system. Across two buildings, there are over 100 terminals attached to the server, and many are in constant use (with a smaller fraction in active use). Users in this group leave their sessions active and frequently utilize the mobility feature via the smart card. Again, aggregate network load is below 5Mbps. The server processors are never fully occupied, and there is certainly sufficient headroom to scale down the server. A major

benefit of this system is the reduction of administration cost from the more than 100 workstations previously used by this group to the single, shared server.

7 Multimedia applications

Real-time applications, such as video or 3D-rendered games, represent an important class of applications that place high demands on system resources in order to provide desired performance. As such, these kinds of applications have traditionally been viewed as totally unsuitable for remote display systems; their bandwidth and processing requirements are expected to be prohibitive. In addition, these kinds of applications differ from the ones studied previously because they have updates that occur at high rates, affect many pixels, and are not tied to human interaction. Thus, they represent the worst-case scenario for a SLIM system.

Multimedia applications typically render directly to the frame buffer in order to maximize performance. To achieve the same effect in a SLIM system, the application must utilize the SLIM protocol directly. This means that there is a porting cost and implementation overhead, and as we will see below, the added overhead can be quite high. However, it is fair to point out that reducing the resolution of the media streams and scaling them locally on the SLIM console reduces this extra cost to a manageable level, thereby enabling applications to achieve display rates well beyond human tolerance limits with little or no noticeable degradation.

Also, because SLIM desktop units have so little resources, we must ensure that multimedia applications do not starve other applications for protocol processing service on the console. Therefore, we implemented a simple network bandwidth allocation mechanism on the Sun Ray 1 console. Under this scheme applications (e.g. the X-server or Quake) executing on possibly different servers make bandwidth requests to the display console based on their past needs. These requests are transparent to the application programmer, as they are made by the X-server on behalf of X applications and by the video library (see Section 2.2) on behalf of multimedia programs. The console sorts the requests in ascending order and grants them one at a time until a request exceeds the available bandwidth, at which point all remaining requests are granted a fair share of the unallocated bandwidth. In this way, high-demand multimedia applications can run while other traditional applications still receive good interactive service from the console.

To analyze SLIM's suitability for these applications, we experimented with three multimedia applications: an MPEG-II player, an NTSC video player, and Quake, an interactive 3-D game from id Software. Using the software library mentioned in Section 2.2, we enhanced these applications with the ability to display directly on Sun Ray 1 consoles via the SLIM protocol. The applications transmit synchronized audio, and they have similar real-time quality of service demands. More specifically, humans require a frame rate of at least 24Hz to achieve

smooth display and proper motion rendition. Quake has the additional requirement of timely user interaction.

Although Quake is a 3-D game, it does not use standard 3-D interfaces, such as OpenGL. Thus, we use it more to assess interactive game performance than to analyze rendering performance for 3-D graphics. Since Sun Ray 1 consoles have no hardware acceleration for 3-D operations, 3-D graphics performance is directly related to the speed of the server processor anyway.

Using the test configuration listed in Table 3, we demonstrate that despite substantial resource requirements, even high-demand applications are supported by the SLIM architecture. In particular, the bandwidth and processing capabilities of the consoles are more than sufficient to meet the demands of these applications, and it turns out that server performance is the primary bottleneck. Of course, it is important to note that in all these cases, a 10Mbps interconnection fabric would not be able to provide adequate performance.

7.1 MPEG-II player

Support was added to Sun's ShowMeTV video player to utilize the SLIM protocol, and we use it for stored video playback. In this experiment, we selected an MPEG-II clip with a resolution of 720x480 and used the CSCS command with 6 bits/pixel compression. We extract frames from the MPEG-II codec at the point where they are in YUV format, and the SLIM video library is then used to transmit them to the desktop.

This application nearly consumes an entire CPU. Disk I/O and video decompression on the server are the performance-limiting factors. The displayed frame rate was fairly uniform at 20Hz (roughly 40Mbps), which is quite good, given the resources available on the desktop. Still, full frame rate (30Hz for this clip) can be achieved by sending every other line and scaling at the desktop. Degradation is not noticeable for the most part, and the bandwidth is reduced by half.

7.2 Live NTSC video

To test live video, we used a custom application which obtains JPEG-compressed NTSC fields from a video capture card, decompresses them up to the point where YUV data are available, and transmits them to the desktop via the CSCS command. Since NTSC is interlaced, we can capture only 640x240 fields and scale up to full size (640x480).

The decompression operation fully consumes the processor, and this application is not multi-threaded. Thus, the server CPU is the performance-limiting factor, and the displayed frame rate ranges from 16Hz to 20Hz (roughly 19–23Mbps), depending on characteristics of the video. To create a situation in which protocol processing on the console is the bottleneck, we simulate 4-way application-level parallelism by simultaneously executing four half-size (320x240) players. In this case, we observe a display frame rate of 25–28Hz (59–66Mbps). Thus, if we were to multi-thread this application, it would display

full-size video which is quite watchable (at the cost of four heavily-utilized CPU's).

7.3 Quake

We use Quake as a representative example of 3D-rendered games. Running it under X produces poor results, and the key to increasing performance would be to use the YUV color space directly in the application. However, we only had access to the code which puts pixels on the display, and that would be a costly porting operation anyway. So, the next best option is to add a translation layer which converts frames to a format suitable for use by the SLIM CSCS protocol command.

When the game engine renders a frame, it produces 8-bit, indexed-color pixels. Our translation calculates a YUV lookup table based on the RGB colormap. To display a frame, we convert the 8-bit pixel values to 5-bit YUV data via table lookup and color component subsampling. Then, the frame is transmitted to a Sun Ray 1 console.

When we run Quake at a resolution of 640x480, the display rate ranges from 18Hz to 21Hz (roughly 22–26Mbps), depending on scene complexity. Although this is good performance, we found the interactive experience to be somewhat lacking. However, if we run Quake at 3/4 resolution (480x360), display rate improves to 28–34Hz (roughly 20–24Mbps), which we found to be playable despite occasional hiccups. The performance-limiting factor in these cases was the CPU, due almost exclusively to the high cost of translation. For example, at the 640x480 resolution it took roughly 30ms/frame to do the YUV conversion and 13ms/frame to transmit the data, which means that the upper bound on display rate was only about 23Hz.

To further improve performance, we could parallelize the application. Because we had limited access to source code, we simulated the effect of parallelizing an instance of the game at full resolution (640x480) by running four instances at 320x240 resolution, thereby creating a situation in which the limiting factor was the protocol processing performance of the desktop unit. With this setup we achieved frame rates ranging from 37Hz to 40Hz (roughly 46–50Mbps), which we found to be smooth and responsive with no hiccups or other noticeable effects.

8 Related work

There are a wide range of lightweight computing devices that have been developed, including network computers, JavaStations, the Plan 9 Gnot, and network PC's. These machines all differ from the SLIM desktop unit in a significant manner. Although they are low-resource implementations, they are full-fledged computers executing an operating system and windowing software. In this section, we compare SLIM to other thin-client systems and discuss how they differ.

8.1 X window system

Because we use X-based applications in our studies, we are able to make the most direct comparison with the X window system. The X protocol[13] was designed to be

fairly high-level, with the goal of minimizing bandwidth by expending computing resources locally. SLIM, on the other hand, is designed to minimize local processing, at the cost of a potentially higher bandwidth overhead. Our results in Figure 8, however, show that both approaches are highly competitive. That is, while the SLIM protocol is much simpler and can be serviced by a low-cost processor, it requires roughly the same bandwidth as X for most applications. For higher-bandwidth, image-based applications, SLIM has a significant advantage over X.

In addition, X terminals are not well-suited to running multimedia applications. Under the X protocol, each frame would have to be transmitted using an `XPutImage` command with no compression possible, i.e. a full 24 bits must be transmitted for each pixel. The situation is different for SLIM. In the worst case, it is the same as X, but typically, some form of compression is possible. If the SLIM CSCS command is not being used, there is still opportunity for finding redundancy among pixels which can reduce bandwidth somewhat. Using the CSCS command, at most 16 bits are required per pixel (a 33% bandwidth reduction) and compression up to 5 bits per pixel (an 88% bandwidth reduction) is possible by altering the color-space conversion parameters. If video is scaled to a higher resolution, the server must perform the operation prior to sending the image to an X terminal, whereas the Sun Ray 1 console can do it locally at substantial savings in bandwidth. For example, transmitting a half-size video stream to a SLIM console and scaling it to full-size would require roughly 18Mbps of bandwidth. However, over 105Mbps of bandwidth would be needed under X. Thus, SLIM and X are extremely competitive at low bandwidth, while SLIM provides substantial savings on higher bandwidth operations, where its impact is much more significant on overall performance.

8.2 Windows-based terminals

Windows-based terminals (WinTerms) employ the Citrix ICA protocol[3] or the Microsoft RDP protocol[5][10] to communicate with a server running Windows NT, Terminal Server Edition. These protocols are quite similar in nature to X but are tied to the Windows GUI API. On the other hand, the low-level SLIM protocol has no such bias and can be used by a system with any rendering API. Another difference between them and the SLIM protocol is that they are highly optimized for low-bandwidth connections. This is accomplished via a variety of techniques, including Windows object-specific compression strategies (e.g. run-length coding of bitmaps), caching of Windows objects and state at the terminal, and maintaining backing store at the terminal. Because the resources included in the terminals directly determine the performance and bandwidth savings possible, these types of systems can have expensive terminals which constantly require upgrades to improve performance. In addition, multimedia capabilities are limited, but because the protocols are extendable, extra support could be added.

8.3 VNC

Like SLIM, the Virtual Network Computing[15][16] (VNC) architecture from Olivetti Research uses a protocol which is independent of any operating system, windowing system, and application. The protocol commands are similar to SLIM, but VNC is designed to access the user's desktop environment from any network connection through a variety of viewers, including a web browser over the internet. The Sun Ray 1 implementation, however, is limited to operating in a workgroup setting with a direct connection to the server.

The key difference between the two approaches, though, is the manner in which the display is updated. With the Sun Ray 1, updates are transmitted from the server to the consoles as they occur in response to application activity. VNC, on the other hand, uses a client-demand approach. Depending on available bandwidth, the VNC viewer periodically requests the current state of the frame buffer. The server responds by transmitting all the pixels that have changed since the last request. This helps the system scale to various bandwidth levels, but has the drawback of larger demands on the server in the form of either maintaining complex state or calculating a large delta between frame buffer states. In either case, our experience with the system is that even in low-latency, high-bandwidth environments, VNC is fairly sluggish. In addition, VNC includes no support for multimedia applications but leaves open the possibility of including specialized compression techniques in the future.

8.4 Other remote-display approaches

The MaxStation[9] from MaxSpeed Corporation is a terminal-based system which uses peripheral cards installed in a PC server machine to transmit video directly to attached consoles. Monitors are refreshed over a 64Mbps connection, but the bandwidth limit restricts the maximum resolution that can be supported to 1024x768 8-bit pixels. The use of expansion cards to create dedicated connections is much less flexible and more difficult to maintain than a commodity switched network, like the Sun Ray 1 uses for the IF.

The Desk Area Network[2][8] (DAN) is a multimedia workstation architecture which uses a high-speed ATM network as the internal interconnect. The frame buffer is a network-attached device which reads and displays pixel tiles. The frame buffer is similar to a Sun Ray 1 desktop unit, but the DAN architecture was designed as a dedicated, stand-alone workstation with sufficient internal bandwidth to transmit high-volume multimedia data (such as video streams) between system components.

9 Conclusions and future work

The SLIM system is a new thin-client architecture. The desktop unit is a stateless, low-cost, fixed-function device that is not much more intelligent than a frame buffer. It requires no system administration and no software or hardware updates. Different applications and operating systems can be ported to display on a SLIM console by

simply adding a virtual device driver for the SLIM protocol to their rendering API's. In addition, the communication requirements are modest and can be supported by commodity interconnection technology.

This paper makes three important contributions. First, we developed an evaluation methodology to characterize interactive performance of modern systems. Second, we characterized the I/O properties of real-world, interactive applications executing in a thin-client environment. Finally, via a set of experiments on an actual implementation of the SLIM system, we have shown that it is feasible to create an implementation which provides excellent quality of service. More specifically, we demonstrated the following:

- Using a dedicated network ensures that added round-trip latency (less than 550 μ s) between the desktop and the remote server is so low that it cannot be distinguished from sitting directly at the console local to the server.
- By only transmitting encoded pixel updates, network traffic requirements of common, interactive programs are well within the capabilities of modern 100Mbps technology. In fact, 10Mbps is more than adequate, and even 1Mbps provides reasonable performance.
- The low-level SLIM protocol is surprisingly effective, requiring a factor of 2–10 less bandwidth to encode display updates than sending the raw pixel data. Further, its bandwidth demands are not much different from the X protocol's, i.e. the simplicity of the protocol did not result in the expected increase in bandwidth cost.
- Yardstick applications quantified multi-user interactive performance and showed that substantial amounts of sharing are possible, e.g. the yardstick application and 10 to 36 other active users can share a processor with no noticeable degradation. Network sharing is an order of magnitude larger than processor sharing.
- The consoles are more than adequate to support even high-demand multimedia applications, and server performance turns out to be the main bottleneck. High-resolution streaming video and Quake can be played at rates which offer a high fidelity user experience.

Still, many research topics remain to be addressed before we can realize the full potential of an ultra-thin architecture like SLIM. Further research is necessary to provide interactive performance guarantees in a shared environment. As we move to a model where there are a large number of users sharing a large number of servers, we may need to rethink the design of the operating system to create a truly scalable, secure, heterogeneous system that fully exploits the mobility and resource sharing ability in the architecture.

Acknowledgments

The Sun Ray 1 implementation would not have been possible without the hard work of the prototype development team: Jim Hanko, Jerry Wall, Alan Ruberg, Lawrence Butcher, and Marc Schneider. We would also like to thank the members of the Sun Ray 1 product team. The experiments would not have been possible without a great deal of technical assistance from and helpful discussions with Jim Hanko and Jerry Wall. Thanks are also due to the many people who participated in our user studies. We would like to thank Sunil Marangoly, Kent Peacock, Kevin Pon, Felicia Stanger and our customers for their help with system resource monitoring. Finally, we would like to thank Sun Microsystems Inc. for its financial support of this study.

References

- [1] T. Anderson, D. Culler, and D. Patterson, "A Case for Networks of Workstations: NOW," *IEEE Micro*, 15(1), February 1995, pp. 54–64.
- [2] P. Barham, M. Hayter, D. MacAuley, and I. Pratt, "Devices on the Desk Area Network," *IEEE Journal on Selected Areas in Communications*, 13(4), May 1995, pp. 722–32.
- [3] Boca Research, Inc., "Citrix ICA Technology Brief," *Technical White Paper*, Boca Raton, FL, 1999.
- [4] Compaq Computer Corporation, "Performance and Sizing of Compaq Servers with Microsoft Windows NT Server 4.0, Terminal Server Edition," *Technology Brief*, Houston, TX, June 1998.
- [5] Databeam Corporation, "A Primer on the T.120 Series Standard," *Technical White Paper*, Lexington, KY, May 1997.
- [6] A. Dearle, "Toward Ubiquitous Environments for Mobile Users," *IEEE Internet Computing*, January-February 1998, pp. 22–32.
- [7] Y. Endo, Z. Wang, J. Chen, and M. Seltzer, "Using Latency to Evaluate Interactive System Performance," *Symposium on Operating System Design and Implementation*, October 1996, pp. 185–199.
- [8] M. Hayter and D. McAuley, "The Desk Area Network," *Operating Systems Review*, 25(4), October 1991, pp. 14–21.
- [9] Maxspeed Corporation, "Ultra-Thin Client — Client/Server Architecture: MaxStations under Multiuser Windows 95," *Technical White Paper*, Palo Alto, CA, 1996.
- [10] Microsoft Corporation, "Comparing MS Windows NT Server 4.0, Terminal Server Edition, and UNIX Application Deployment Solutions," *Technical White Paper*, Redmond, WA, 1999.
- [11] J. Nieh and M. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Operating Systems Review*, 31(5), December 1997, pp. 184–97.
- [12] J. D. Northcutt, J. Hanko, G. Wall, A. Ruberg, "Towards a Virtual Display Architecture," *Project Technical Report SMLI 98-0184*, Sun Microsystems Laboratories, 1998.
- [13] A. Nye (Ed.), *X Protocol Reference Manual*, O'Reilly & Associates, Inc., 1992.
- [14] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," *Proceedings of the Summer 1990 UKUUG Conference*, July 1990, pp. 1–9.
- [15] T. Richardson, F. Bennet, G. Mapp, and A. Hopper, "Teleporting in an X Window System Environment," *IEEE Personal Communications*, No. 3, 1994, pp. 6–12.
- [16] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, January–February 1998, pp. 33–38.
- [17] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd edition, Addison-Wesley, Reading, MA, 1992.
- [18] M. Schneider and L. Butcher, "NewT Human Interface Device Mark II Terminal Hardware Specification," *Project Technical Report SMLI 98-0200*, Sun Microsystems Laboratories, Palo Alto, CA, 1998.
- [19] Sun Microsystems Inc., "Java Server Sizing Guide," *Technical White Paper*, Palo Alto, CA, October 1997.
- [20] A. Tamches and B. Miller, "Fine-Grain Dynamic Instrumentation of Commodity Operating System Kernels," *Symposium on Operating Systems Design and Implementation*, February 1999, pp. 117–30.
- [21] Unisys, "Sizing and Performance Analysis of Microsoft Windows NT Server 4.0, Terminal Server Edition, on Unisys Aquanta Servers," *Technical White Paper*, Blue Bell, PA, August 1998.

Linguistic Knowledge Can Improve Information Retrieval

William A. Woods, Lawrence A. Bookman, Ann Houston, Robert J. Kuhns, Paul Martin, and Stephen Green

Introduction by William A. Woods

This article was published in the Proceedings of the Applied Natural Language Processing Conference (ANLP-2000) in Seattle, Washington, May 1-3, 2000. A preliminary version was published as a technical report (TR-99-83) in the Sun Microsystems Laboratories Technical Report Series. The article represents a milestone in an ongoing project aimed at discovering technology to help people find specific information online. We undertook this project with the understanding that if we could significantly improve people's ability to find information online, then this would have wide impact across a variety of computer applications, from finding information in online documentation, to finding needed information on a Web site; from following ideas in research material, to organizing case material for legal proceedings; from supporting telephone call centers, to publishing information with indexes that enable people to efficiently find what they need; and on and on ... This project anticipated the needs that many people have now experienced using search engines on the Web.

We began by looking closely at what goes wrong when people try to find information online and looking for ways to solve the problems that we identified. One problem that immediately became apparent was that people often asked questions using different terminology from that used by the author of the material they needed to find. Sometimes they used different morphological forms of a word, like "compute" versus "computation," and sometimes they used different semantically related words, like "go" versus "move." We've come to call this the "paraphrase problem." Another obvious problem was that people spent an inordinate amount of time looking through the documents returned by a request, trying to determine whether they actually contained the information that was needed (and unfortunately, most of them didn't).

This investigation led to the discovery of the techniques described in this article, in which the indexing system automatically constructs a conceptual taxonomy of all the words and phrases in the indexed material, organized by generality, using an extensive body of linguistic knowledge, including knowledge of semantic relationships among concepts and morphological structure and relationships between words. We call this "conceptual indexing." This taxonomy is then used by a specific passage retrieval algorithm to make connections between what you ask for and what you need to find. The taxonomy also supports very effective deep browsing. The specific passage retrieval system locates not just documents, but specific passages in those documents that are likely to contain the information you need, and it effectively ranks these passages using a scoring method that really does tend to rank the best passages first. The system saves people time spent searching and improves the quality of decisions, using linguistic knowledge to help them find what they really want.

This project is an activity of the Knowledge Technology Group in Sun Microsystems Laboratories, whose goal is to develop and exploit technology for dealing with knowledge — acquiring, organizing, disseminating, retrieving, and browsing it. Current members of the team are: William A. Woods, Stephen Green, Paul Martin, Robert J. Kuhns, Ann Houston, and summer intern, Scott Sanner. Former members are acknowledged in the article. The project has developed a powerful technology that is now used internally at Sun in a number of applications, and some of it has been included in Sun products. Going forward, we continue to look for ways to exploit what has been developed so far and to discover new ways to help people deal with knowledge online.

You can find a brief overview of conceptual indexing technology at:
<http://www.sun.com/research/knowledge/>

You can use the technology to search the Sun Microsystems Laboratories Web site at:
<http://www.research.sun.com/>

REFERENCES:

"Conceptual Indexing: A Better Way to Organize Knowledge," W. A. Woods, Technical Report SMLI TR-97-61, Sun Microsystems Laboratories, Mountain View, CA, April, 1997.
Online at: <http://www.sun.com/techrep/1997/abstract-61.html>

"Finding the Hidden Gems: Search Engines and Beyond," W. A. Woods, Knowledge Management in the Telecoms Industry, First Conferences, Ltd., London, ENGLAND, October 23-24, 1997.

"Knowledge Management Needs Effective Search Technology," W. A. Woods, Sun Journal, March, 1998. Online at: http://www.sun.com/sun-journal/V2N1/03_feat2a.html

"Natural Language Technology in Precision Content Retrieval," J. Ambroziak and W. A. Woods, proceedings of the International Conference on Natural Language Processing and Industrial Applications (NLP+IA 98), August 18-21, 1998, Moncton, New Brunswick, CANADA.
Online at: <http://www.sunlabs.com/techrep/1998/abstract-69.html>

"Linguistic Knowledge can Improve Information Retrieval," William A. Woods, Lawrence A. Bookman, Ann Houston, Robert J. Kuhns, Paul Martin, and Stephen Green, Proceedings of ANLP-2000, Seattle, WA, May 1-3, 2000, (preliminary version: Technical Report SMLI TR-99-83, Sun Microsystems Laboratories, Mountain View, CA, December, 1999.
Online at: <http://www.sun.com/research/techrep/1999/abstract-83.html>

"Aggressive Morphology for Robust Lexical Coverage," William A. Woods, Proceedings of ANLP-2000, Seattle, WA, May 1-3, 2000, (preliminary version: Technical Report SMLI TR-99-82, Sun Microsystems Laboratories, Mountain View, CA, December, 1999.
Online at: <http://www.sun.com/research/techrep/1999/abstract-82.html>

"Conceptual Indexing: Practical Large-Scale AI for Efficient Information Access," William A. Woods, Invited talk at AAAI 2000, Proceedings AAAI 2000, Austin, TX, August 2, 2000.

"Halfway to Question Answering," by W. A. Woods, Stephen Green, Paul Martin, and Ann Houston, Proceedings of the TREC-9 Conference, November, 13-16, 2000.

Linguistic Knowledge can Improve Information Retrieval

William A. Woods Lawrence A. Bookman* Ann Houston

Robert J. Kuhns Paul Martin

Stephen Green

Sun Microsystems Laboratories

1 Network Drive

Burlington, MA 01803

{William.Woods,Ann.Houston,Robert.Kuhns}@east.sun.com

{Paul.Martin,Stephen.Green}@east.sun.com

March 10, 2000

Abstract

This paper describes the results of some experiments using a new approach to information access that combines techniques from natural language processing and knowledge representation with a penalty-based technique for relevance estimation and passage retrieval. Unlike many attempts to combine natural language processing with information retrieval, these results show substantial benefit from using linguistic knowledge.

1 Introduction

An online information seeker often fails to find what is wanted because the words used in the request are different from the words used in the relevant material. Moreover, the searcher usually spends a significant amount of time reading retrieved material in order to determine whether it contains the information sought. To address these problems, a system has been developed at Sun Microsystems Laboratories (Ambroziak and Woods, 1998) that uses techniques from natural language processing and knowledge representation, with a technique for dynamic passage selection and scoring, to significantly improve retrieval performance. This system is able to locate specific passages in the indexed material where the requested information appears to be, and to score those passages with a penalty-based score that is highly correlated with the likelihood that they contain relevant information. This ability, which we call “Precision Content Retrieval” is achieved

*Lawrence Bookman is now at Torrent Systems, Inc.

by combining a system for Conceptual Indexing with an algorithm for Relaxation-Ranking Specific Passage Retrieval.

In this paper, we show how linguistic knowledge is used to improve search effectiveness in this system. This is of particular interest, since many previous attempts to use linguistic knowledge to improve information retrieval have met with little or mixed success (Fagan, 1989; Lewis and Sparck Jones, 1996; Sparck Jones, 1998; Varile and Zampolli, 1997; Voorhees, 1993; Mandala et al., 1999) (but see the latter for some successes as well).

2 Conceptual Indexing

The conceptual indexing and retrieval system used for these experiments automatically extracts words and phrases from unrestricted text and organizes them into a semantic network that integrates syntactic, semantic, and morphological relationships. The resulting conceptual taxonomy (Woods, 1997) is used by a specific passage-retrieval algorithm to deal with many paraphrase relationships and to find specific passages of text where the information sought is likely to occur. It uses a lexicon containing syntactic, semantic, and morphological information about words, word senses, and phrases to provide a base source of semantic and morphological relationships that are used to organize the taxonomy. In addition, it uses an extensive system of knowledge-based morphological rules and functions to analyze words that are not already in its lexicon, in order to construct new lexical entries for previously unknown words (Woods, 2000). In addition to rules for handling derived and inflected forms of known words, the system includes rules for lexical compounds and rules that are capable of making reasonable guesses for totally unknown words.

A pilot version of this indexing and retrieval system, implemented in Lisp, uses a collection of approximately 1200 knowledge-based morphological rules to extend a core lexicon of approximately 39,000 words to give coverage that exceeds that of an English lexicon of more than 80,000 base forms (or 150,000 base plus inflected forms). Later versions of the conceptual indexing and retrieval system, implemented in C++, use a lexicon of approximately 150,000 word forms that is automatically generated by the Lisp-based morphological analysis from its core lexicon and an input word list. The base lexicon is extended further by an extensive name dictionary and by further morphological analysis of unknown words at indexing time. This paper will describe some experiments using several versions of this system. In particular, it will focus on the role that the linguistic knowledge sources play in its operation.

The lexicon used by the conceptual indexing system contains syntactic information that can be used for the analysis of phrases, as well as morphological and semantic information that is used to relate more specific concepts to more general concepts in the conceptual taxonomy. This information is integrated into the conceptual taxonomy by considering base forms of words to subsume their derived and inflected forms (“root subsumption”) and more general terms to subsume more specific terms. The system uses these relationships as the basis for inferring subsumption relationships between more general phrases and more specific phrases according to the intensional subsumption logic of Woods (Woods, 1991).

The largest base lexicon used by this system currently contains semantic subsumption information for something in excess of 15,000 words. This information consists of basic “kind of” and

“instance of” information such as the fact that *book* is a kind of *document* and *washing* is a kind of *cleaning*. The lexicon also records morphological roots and affixes for words that are derived or inflected forms of other words, and information about different word senses and their interrelationships. For example, the conceptual indexing system is able to categorize *becomes black* as a kind of *color change* because *becomes* is an inflected form of *become*, *become* is a kind of *change*, and *black* is a *color*. Similarly, *color disruption* is recognized as a kind of *color change*, because the system recognizes *disruption* as a derived form of *disrupt*, which is known in the lexicon to be a kind of *damage*, which is known to be a kind of *change*.

When using root subsumption as a technique for information retrieval, it is important to have a core lexicon that knows correct morphological analyses for words that the rules would otherwise analyze incorrectly. For example, the following are some examples of words that could be analyzed incorrectly if the correct interpretations were not specified in the lexicon:

delegate (de+leg+ate) take the legs from

caress (car + ess) female car

cashier (cashy + er) more wealthy

daredevil (dared + evil) serious risk

lacerate (lace + rate) speed of tatting

pantry (pant + ry) heavy breathing

pigeon (pig + eon) the age of peccaries

ratify (rat + ify) infest with rodents

infantry (infant + ry) childish behavior

Although they are not always as humorous as the above examples, there are over 3,000 words in the core lexicon of 39,000 English words that would receive false morphological analyses like the above examples, if the words were not already in the lexicon.

3 Relaxation Ranking and Specific Passage Retrieval

The system we are evaluating uses a technique called “relaxation ranking” to find specific passages where as many as possible of the different elements of a query occur near each other, preferably in the same form and word order and preferably closer together. Such passages are ranked by a penalty score that measures the degree of deviation from an exact match of the requested phrase, with smaller penalties being preferred. Differences in morphological form and formal subsumption of index terms by query terms introduce small penalties, while intervening words, unexplained permutations of word order, and crossing sentence boundaries introduce more significant penalties.

Elements of a query that cannot be found nearby introduce substantial penalties that depend on the syntactic categories of the missing words.

When the conceptual indexing system is presented with a query, the relaxation-ranking retrieval algorithm searches through the conceptual taxonomy for appropriately related concepts and uses the positions of those concepts in the indexed material to find specific passages that are likely to address the information needs of the request. This search can find relationships from base forms of words to derived forms and from more general terms to more specific terms, by following paths in the conceptual taxonomy.

For example, the following is a passage retrieved by this system, when applied to the UNIX[®] operating system online documentation (the “man pages”):

Query: print a message from the mail tool

6. -2.84 **print mail mail mailtool**

Print sends copies of all the selected mail items to your default printer. If there are no selected items, mailtool sends copies of those items you are currently...

The indicated passage is ranked 6th in a returned list of found passages, indicated by the 6 in the above display. The number -2.84 is the penalty score assigned to the passage, and the subsequent words *print*, *mail*, *mail*, and *mailtool* indicate the words in the text that are matched to the corresponding content words in the input query. In this case, *print* is matched to *print*, *message* to *mail*, *mail* to *mail*, and *tool* to *mailtool*, respectively. This is followed by the content of the actual passage located. The information provided in these hit displays gives the information seeker a clear idea of why the passage was retrieved and enables the searcher to quickly skip down the hit list with little time spent looking at irrelevant passages. In this case, it was easy to identify that the 6th ranked hit was the best one and contained the relevant information.

The retrieval of this passage involved use of a semantic subsumption relationship to match *message* to *mail*, because the lexical entry for *mail* recorded that it was a kind of *message*. It used a morphological root subsumption to match *tool* to *mailtool* because the morphological analyzer analyzed the unknown word *mailtool* as a compound of *mail* and *tool* and recorded that its root was *tool* and that it was a kind of *tool* modified by *mail*. Taking away the ability to morphologically analyze unknown words would have blocked the retrieval of this passage, as would eliminating the lexical subsumption entry that recorded *mail* as a kind of *message*.

Like other approaches to passage retrieval (Kaszkiel and Zobel, 1997; Salton et al., 1993; Callan, 1994), the relaxation-ranking retrieval algorithm identifies relevant passages rather than simply identifying whole documents. However, unlike approaches that involve segmenting the material into paragraphs or other small passages before indexing, this algorithm dynamically constructs relevant passages in response to requests. When responding to a request, it uses information in the index about positions of concepts in the text to identify relevant passages. In response to a single request, identified passages may range in size from a single word or phrase to several sentences or paragraphs, depending on how much context is required to capture the various elements of the request.

In a user interface to the specific passage retrieval system, retrieved passages are reported to the user in increasing order of penalty, together with the rank number, penalty score, information about which target terms match the corresponding query terms, and the content of the identified passage with some surrounding context as illustrated above. In one version of this technology, results are presented in a hypertext interface that allows the user to click on any of the presented items to see that passage in its entire context in the source document. In addition, the user can be presented with a display of portions of the conceptual taxonomy related to the terms in the request. This frequently reveals useful generalizations of the request that would find additional relevant information, and it also conveys an understanding of what concepts have been found in the material that will be matched by the query terms. For example, in one experiment, searching the online documentation for the Emacs text editor, the request *jump to end of file* resulted in feedback showing that *jump* was classified as a kind of *move* in the conceptual taxonomy. This led to a reformulated request, *move to end of file*, which successfully retrieved the passage *go to end of buffer*.

4 Experimental Evaluation

In order to evaluate the effectiveness of the above techniques, a set of 90 queries was collected from a naive user of the UNIX operating system, 84 of which could be answered from the online documentation known as the man pages. A set of “correct” answers for each of these 84 queries was manually determined by an independent UNIX operating system expert, and a snapshot of the man pages collection was captured and indexed for retrieval. In order to compare this methodology with classical document retrieval techniques, we assign a ranking score to each document equal to the ranking score of the best ranked passage that it contains.

In rating the performance of a given method, we compute average recall and precision values at 10 retrieved documents, and we also compute a “success rate” which is simply the percentage of queries for which an acceptable answer occurs in the top ten hits. The success rate is the principal factor on which we base our evaluations, since for this application, the user is not interested in subsequent answers once an acceptable answer has been found, and finding one answer for each of two requests is a substantially better result than finding two answers to one request and none for another.

These experiments were conducted using an experimental retrieval system that combined a Lisp-based language processing stage with a C++ implementation of a conceptual indexer. The linguistic knowledge sources used in these experiments included a core lexicon of approximately 18,000 words, a substantial set of morphological rules, and specialized morphological algorithms covering inflections, prefixes, suffixes, lexical compounding, and a variety of special forms, including numbers, ordinals, Roman numerals, dates, phone numbers, and acronyms. In addition, they made use of a lexical subsumption taxonomy of approximately 3000 lexical subsumption relations, and a small set of semantic entailment axioms (e.g., *display* entails *see*, but is not a kind of *see*). This

system is described in (Woods, 1997). The database was a snapshot of the local man pages (frozen at the time of the experiment so that it wouldn’t change during the experiment), consisting of approximately 1800 files of varying lengths and constituting a total of approximately 10 megabytes of text.

Table 1: A comparison of different retrieval techniques.

System	Success Rate	Recall (10 docs)	Precision (10 docs)
<i>tf.idf</i>	28.6%	14.8%	2.9%
SearchIt system	44.0%	28.5%	7.4%
Recall II	60.7%	38.6%	7.3%
w/o morph	50.0%	not measured	not measured
w/o knowledge	42.9%	not measured	not measured

Table 1 shows the results of comparing three versions of this technology with a textbook implementation of the standard *tf.idf* algorithm (Salton, 1989) and with the SearchItTM search application developed at Sun Microsystems, Inc., which combines a simple morphological query expansion with a state-of-the-art commercial search engine. In the table, Recall II refers to the full conceptual indexing and search system with all of its knowledge sources and rules. The line labeled “w/o morph” refers to this system with its dynamic morphological rules turned off, and the line labeled “w/o knowledge” refers to this system with all of its knowledge sources and rules turned off. The table presents the success rate and the measured recall and precision values for 10 retrieved documents. We measured recall and precision at the 10 document level because internal studies of searching behavior had shown that users tended to give up if an answer was not found in the first ten ranked hits. We measured success rate, rather than recall and precision, for our ablation studies, because standard recall and precision measures are not sensitive to the distinction between finding multiple answers to a single request versus finding at least one answer for more requests.

5 Discussion

Table 1 shows that for this task, the relaxation-ranking passage retrieval algorithm without its supplementary knowledge sources (Recall II w/o knowledge) is roughly comparable in performance (42.9% versus 44.0% success rate) to a state-of-the-art commercial search engine (SearchIt) at the pure document retrieval task (neglecting the added benefit of locating the specific passages). Adding the knowledge in the core lexicon (which includes morphological relationships, semantic subsumption axioms, and entailment relationships), but without morphological analysis of unknown words (Recall II w/o morph), significantly improves these results (from 42.9% to 50.0%). Further adding the morphological analysis capability that automatically analyzes unknown words (deriving additional morphological relationships and some semantic subsumption relationships)

significantly improves that result (from 50.0% to 60.7%). In contrast, we found that adding the same semantic subsumption relationships to the commercial search engine, using its provided thesaurus capability degraded its results, and results were still degraded when we added only those facts that we knew would help find relevant documents. It turned out that the additional relevant documents found were more than offset by additional irrelevant documents that were also ranked more highly.

6 Anecdotal Evaluation of Specific Passage Retrieval Benefits

As mentioned above, comparing the relaxation-ranking algorithm with document retrieval systems measures only a part of the benefit of the specific passage retrieval methodology. Fully evaluating the quality and ranking of the retrieved passages involves a great many subtleties. However, two informal evaluations have been conducted that shed some light on the benefits.

The first of these was a pilot study of the technology at a telecommunications company. In that study, one user found that she could use a single query to the conceptual indexing system to find both of the items of information necessary to complete a task that formerly required searching two separate databases. The conclusion of that study was that the concept retrieval technology performs well enough to be useful to a person talking live with a customer. It was observed that the returned hits can be compared with one another easily and quickly by eye, and attention is taken directly to the relevant content of a large document. The automatic indexing was considered a plus compared with manual methods of content indexing. It was observed that an area of great potential may be in a form of knowledge management that involves organizing and providing intelligent access to small, unrelated “nuggets” of textual knowledge that are not amenable to conventional database archival or categorization.

A second experiment was conducted by the Human Resources Webmaster of a high-tech company, an experienced user of search engines who used this technology to index his company’s internal HR web site. He then measured the time it took him to process 15 typical HR requests, first using conventional search tools that he had available, and then using the Conceptual Indexing technology. In both cases, he measured the time it took him to either find the answer or to conclude that the answer wasn’t in the indexed material. His measured times for the total suite were 55 minutes using the conventional tools and 11 minutes using the conceptual indexing technology. Of course, this was an uncontrolled experiment, and there is some potential that information learned from searching with the traditional tools (which were apparently used first) might have provided some benefit when using the conceptual indexing technology. However, the fact that he found things with the latter that he did not find with the former and the magnitude of the time difference suggests that there is an effect, albeit perhaps not as great as the measurements. As a result of this experience, he concluded that he would expect many users to take much longer to find materials or give up, when using the traditional tools. He anticipated that after finding some initial materials, more time would be required, as users would end up having to call people for additional information. He estimated that users could spend up to an hour trying to get the infor-

mation they needed...having to call someone, wait to make contact and finally get the information they needed. Using the conceptual indexing search engine, he expected that these times would be at least halved.

7 Conclusion

We have described some experiments using linguistic knowledge in an information retrieval system in which passages within texts are dynamically found in response to a query and are scored and ranked based on a relaxation of constraints. This is a different approach from previous methods of passage retrieval and from previous attempts to use linguistic knowledge in information retrieval. These experiments show that linguistic knowledge can significantly improve information retrieval performance when incorporated into a knowledge-based relaxation-ranking algorithm for specific passage retrieval.

The linguistic knowledge considered here includes the use of morphological relationships between words, taxonomic relationships between concepts, and general semantic entailment relationships between words and concepts. We have shown that the combination of these three knowledge sources can significantly improve performance in finding appropriate answers to specific queries when incorporated into a relaxation-ranking algorithm. It appears that the penalty-based relaxation-ranking algorithm figures crucially in this success, since the addition of such linguistic knowledge to traditional information retrieval models typically degrades retrieval performance rather than improving it, a pattern that was borne out in our own experiments.

Acknowledgments

Many other people have been involved in creating the conceptual indexing and retrieval system described here. These include: Gary Adams, Jacek Ambroziak, Cookie Callahan, Chris Colby, Jim Flowers, Ellen Hays, Patrick Martin, Peter Norvig, Tony Passera, Philip Resnik, Robert Sproull, and Mark Torrance.

Sun, Sun Microsystems, and SearchIt are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. UNIX est une marque enregistree aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

References

- (Ambroziak and Woods, 1998) Jacek Ambroziak and William A. Woods. Natural language technology in precision content retrieval. In *International Conference on Natural Language Processing and Industrial Applications*, Moncton, New Brunswick, Canada, August 1998. www.sun.com/research/techrep/1998/abstract-69.html.

- (Callan, 1994) Jamie P. Callan. Passage-level evidence in document retrieval. *SIGIR*, pages 302–309, 1994.
- (Fagan, 1989) J. L. Fagan. The effectiveness of a nonsyntactic approach to automatic phrase indexing for document retrieval. *Journal of the American Society for Information Science*, **40**(2):115–132, March 1989.
- (Kaszkiel and Zobel, 1997) Marcin Kaszkiel and Justin Zobel. Passage retrieval revisited. *SIGIR*, pages 302–309, 1997.
- (Lewis and Sparck Jones, 1996) David D. Lewis and Karen Sparck Jones. Natural language processing for information retrieval. *CACM*, **39**(1):92–101, 1996.
- (Mandala et al., 1999) Rila Mandala, Takenobu Tokunaga, and Hozumi Tanaka. Combining multiple evidence from different types of thesaurus for query expansion. In *Proceedings on the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM-SIGIR, 1999.
- (Salton et al., 1993) Gerald Salton, James Allan, and Chris Buckley. Approaches to passage retrieval in full text information systems. *SIGIR*, pages 49–58, 1993.
- (Salton, 1989) Gerard Salton. *Automatic Text Processing*. Addison Wesley, Reading, MA, 1989.
- (Sparck Jones, 1998) Karen Sparck Jones. A look back and a look forward. *SIGIR*, pages 13–29, 1998.
- (Varile and Zampolli, 1997) Giovanni Varile and Antonio Zampolli, editors. *Survey of the State of the Art in Human Language Technology*. Cambridge Univ. Press, 1997.
- (Voorhees, 1993) Ellen M. Voorhees. Using wordnet to disambiguate word senses for text retrieval. In *Proceedings of 16th ACM SIGIR Conference*. ACM-SIGIR, 1993.
- (Woods, 1991) William A. Woods. Understanding subsumption and taxonomy: A framework for progress. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 45–94. Morgan Kaufmann, San Mateo, CA, 1991.
- (Woods, 1997) William A. Woods. Conceptual indexing: A better way to organize knowledge. Technical Report SMLI TR-97-61, Sun Microsystems Laboratories, Mountain View, CA, April 1997. www.sun.com/research/techrep/1997/abstract-61.html.
- (Woods, 2000) William A. Woods. Aggressive morphology for robust lexical coverage. In *(these proceedings)*, 2000.

Designing Fast Asynchronous Circuits

Ivan E. Sutherland, Jon K. Lexau

Introduction by Ivan E. Sutherland

I often complained to the late Charlie Molnar about how hard our research project is. He always responded that I should be pleased with our opportunity to get "way ahead." Yes, it's been hard, but after a decade of research we've achieved exciting results. It is now easy to argue that, given a type of transistors, our circuits make them go as fast as they possibly can. This paper, one of four that we presented at the ASYNC 2001 conference, March 2001, in Salt Lake City, tells how we design our circuits; our other conference papers [see references below] describe some of our circuit forms, how to use them, and a novel computer design, called FLEET.

Our group in Sun Microsystems Laboratories works on circuits, called "asynchronous," that use internal timing instead of the more common external "clock." Instead of waiting for the next clock tick, our asynchronous circuits proceed as fast as they can, waiting only for data or for space to put results. Clocked circuits are more common in computers today because they are easier to design. Although asynchronous circuits are harder to design, we've been able to build and demonstrate an impressive collection of them. Our circuits are real; we build and test silicon chips to prove that they work.

To make fast circuits we put very little in them. For the past half decade we've been simplifying our circuits to make them faster. It is a remarkable fact of physics that any embellishment of a circuit must slow it down. The Theory of Logical Effort [book referenced below] relates hardware delay to logical complexity, showing how to compute the minimum delay for a given digital logic function. The paper you are about to read shows how we apply logical effort theory.

The paper describes a chip design process unlike any other. Most chip designs begin with a target speed. Instead, our process starts by making the promise that all logic gates will have a uniform but not yet specified delay. This promise simplifies the design task, but more importantly, it accommodates "lean and mean" logic circuits free of embellishment. The resulting circuits are very fast, not only because they are simple, but also because they use each transistor to its full potential.

Only at the end of the design process do we pick a value for the initially-unspecified delay. We make good on the initial promise of uniform delay by picking the widths of transistors so that every transistor drives a load proportional to its strength. Although the minimum achievable delay depends on the circuit details, it is also possible to lengthen the uniform delay to save power in a lower speed version. Because the choice of speed comes late in our design process, a single logic design can serve either high speed and low power applications by appropriate choice of transistor widths.

At the outset of this project we didn't anticipate finding a unique design process, but good research includes surprises. We set out to build a particular class of circuits, bringing some mathematical tools to bear on the problems we encountered. We built and tested real chips. We "listened to the silicon" to hear what its physics tells about what to design and how to design it. We tried many things that didn't work, learning from initial failures. From our effort has emerged the principal value of research, understanding. This paper, and the other papers and patents from Sun Labs, are attempts to share understanding with other people so that Sun, its engineers, its customers, and our society may benefit.

"Designing Fast Asynchronous Circuits." © 2001 Sun Microsystems, Inc. and IEEE. Reprinted, with permission, from Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA, 11–14 March 2001, pp. 184–93. Also © 2000, Sun Microsystems, Inc.

PUBLICATIONS:

Ivan Sutherland, Bob Sproull and David Harris, "Logical Effort: Designing Fast CMOS Circuits," Morgan and Kaufmann Publishers, 1999. 1st Edition

REFERENCES:

William Coates, Jon Lexau, Ian Jones, Scott Fairbanks and Ivan Sutherland, FLEETzero: An Asynchronous Switching Experiment, Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA, 11–14 March 2001. pp. 173–182. (sml#2000–0768) Copyright 2001 by IEEE. Used by permission. Also Copyright 2000, Sun Microsystems, Inc.

Jo Ebergen, Squaring the FIFO in GasP, Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA, 11–14 March 2001. pp. 194–205. (sml#2000–0755) Copyright 2001 by IEEE. Used by permission. Also Copyright 2000, Sun Microsystems, Inc.

Ivan Sutherland and Jon Lexau, Designing Fast Asynchronous Circuits, Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA, 11–14 March 2001. pp. 184–193. (sml#2000–0759) Copyright 2001 by IEEE. Used by permission. Also Copyright 2000, Sun Microsystems, Inc. Slides. sml2001–0143.

Ivan Sutherland and Scott Fairbanks, GasP: A Minimal FIFO Control, Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA, 11–14 March 2001. pp. 46–53. (sml#2000–0756) Copyright 2001 by IEEE. Used by permission. Also Copyright 2000, Sun Microsystems, Inc.

William Coates, Jo Ebergen, Jon Lexau, Scott Fairbanks, Ian Jones, Alex Ridgway, David Harris and Ivan Sutherland A Counterflow Pipeline Experiment, Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems, Barcelona, Spain. 19–21 April 1999. pp. 161–172. (sml#99–0068) Copyright 1999 by IEEE. Used by permission.

Charles Molnar, Ian Jones, William Coates and Jon Lexau, A FIFO Ring Performance Experiment, Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eindhoven, The Netherlands, 7–10 April 1997. pp. 279–289. (sml#97–0015) Copyright 1997 by IEEE. Used by permission.

Ivan Sutherland, Micropipelines: Turing Award, Communications of the ACM, June 1989. Vol 32, #6, pp. 720–738. sml#94:0412. Copyright 1989 by ACM, Inc. Used by permission. <http://info.acm.org/pubs/toc/CRnotice.html>

Designing Fast Asynchronous Circuits

Ivan E. Sutherland and Jon K. Lexau

Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303-4900

Abstract

A five-step design process for asynchronous circuits helps simplify their logic and speed their operation. First, assume that all logic gates in the control will have nearly uniform delay. Second, use the uniform delay assumption to simplify control logic. Third, lay out the chip to get wire length data. Fourth, choose a specific delay and calculate transistor widths to apply that specific delay uniformly to all logic gates in the control; this paper shows how. Fifth, verify correct operation with standard methods.

The specific gate delay trades off speed, area, and power consumption; postponing its choice takes advantage of asynchrony to accommodate the limitations imposed by layout. The theoretical lower bound for specific delay depends on the logical effort of the most complex loop in the design and remarkably, is independent of wire capacitance, given wide enough transistors, but wire capacitance puts practical bounds on speed. The effect of wire resistance remains unexplored.

0. Introduction

Circuit delays determine system performance. In synchronous systems each chain of logic must meet a pre-defined delay constraint to fit the specified period of the clock. Meeting this constraint requires great care in logic design, electrical design, and layout. In stark contrast, some asynchronous systems, called “delay insensitive,” use logic arrangements that guarantee correct function regardless of circuit delays, paying a price in speed and logic complexity for simplicity of electrical design. We believe that designs somewhere between these two extremes will achieve the best performance. At the register level our systems rely on pre-calculated circuit delays; it is only at higher levels that they are asynchronous. Individual *modules* will wait as long as required for adjacent modules to act. Request and acknowledge signals travel between modules to coordinate their asynchronous action.

We have been using the five-step design method described in the abstract for some time. It was only, however, as a result of complaints about early drafts of this paper that we came to articulate the steps clearly. Our GasP circuits, described in [6], depend for correct operation on nearly uniform delay in all gates. We designed GasP in the belief that we could use logical effort theory [7] to fulfill the uniform delay assumption without, initially, knowing exactly how to do so for complex designs. We discovered that the uniform delay assumption facilitates the use of self-resetting logic gates, called “post charge logic” by Proebsting [3][4][5]. To reduce the logical effort of our modules we made increasing use of self-resetting techniques. We knew that reduced logical effort must reduce delay, but we did not, until recently, realize that the worst loop’s logical effort establishes an irreducible minimum gate delay and therefore a theoretical maximum speed.

Partly in response to the need for uniform delays posed by GasP circuits, we developed the electrical model of transistor widths described here. Using the width model enabled us to find suitable transistor widths for designs too complex for algebraic expression or hand calculation. As we began to use the width model we noticed that for each logic design there was a limit to obtainable speed. For too ambitious a speed goal the width model calculation wouldn’t converge.

The equations presented here show that the logical effort of the worst logic loop of each design limits its maximum speed. Fortunately, even the most complex circuit in the GasP family has very low logical effort; we drove towards that goal merely because it seemed right, and not initially from any deep understanding of its essential importance. It is the low logical effort of the GasP family that makes aggressive choices of transistor widths possible and leads to very fast operation. The Muller C-elements in the Micropipeline control, for example, have too much logical effort to reach equivalent speeds. An example of transistor width calculation for the Micropipeline control appears near the end of this paper.

The bulk of this paper is about step four of the five-step design process, balancing transistor widths to realize nearly

uniform gate delay. As you read the companion paper [6] about GasP, bear in mind how the uniform delay assumption impacts design. As you think about more complex systems such as those described in [1] and [2], recognize that after their logic design is complete we pick suitable transistor widths using the methods described in this paper. It should be equally possible to use narrower transistors for slower operation with reduced power consumption while retaining nearly uniform gate delay.

Although wire capacitance has no impact on maximum theoretical speed, one must use good wire capacitance information to pick good transistor widths. Driving long wires requires wide transistors. Early estimates of wire length and preliminary choices of specific gate delay can lead to wise choices of transistor width prior to layout. Final choice of transistor width should include real wire lengths from the final layout; good initial estimates save later adjustments. Like many design methods for integrated circuits, this one is really iterative; transistor widths depend on wire length which, to some extent, may depend on transistor widths.

1. Step One: Assume Uniform Gate Delays

This is a pretty easy step. The only tricky point is that some gates may contain only half an inverter—a single transistor. The self-resetting gates in GasP, for example, have separate inputs for action and reset, often two series N-type transistors for action and a single P-type transistor for reset. Because each half acts only during part of the cycle, one treats them as entirely separate logic gates, each of which must exhibit the chosen delay.

But how about unequal rising and falling delays? If some gates consist of only a P-type transistor and some of only an N-type transistor, must we balance the delays in each in spite of the difference in their electrical properties? The difference in delay doesn't matter because we avoid races between rising and falling transitions. Rising transitions interact only with rising transitions, and falling with falling. Thus, in principle, one can retain two uniform delay assumptions, one for rising and one for falling transitions. Although we typically use a 2/1 ratio of P- to N-transistor width, we believe that any reasonable ratio will serve equally well.

2. Step Two: Design the Control Logic

The uniform delay assumption is useful in three ways. First, it encourages designs with an equal number of logic gates in every loop. Were it not for the uniform delay

assumption, we might put more gates in some loops than in others, hoping to balance loop delay by subsequent choice of transistor widths in logically mismatched loops, a difficult optimization chore. Seeking fast designs, we generally choose three or five gates per loop; three often prove sufficient. With the same number of logic gates in each loop, each with the same delay, each loop runs at about the same speed, keeping pipeline throughput consistent. Making some stages of a pipeline system faster than others makes little sense, because system throughput is set by the slowest stage. Instead, one should save power and area by allowing the inherently simple parts of the system to run no faster than the complex parts.

Second, the uniform delay assumption is important because it encourages use of logic gates that have very low logical effort. Self-resetting gates, for example, eliminate unnecessary transistors by borrowing suitable reset signals from nodes later in the chain of logic where they enjoy a higher power level. Multiple use of single wires is also possible because opposing drive transistors don't conduct concurrently.

Third, the uniform delay assumption encourages breaking complex loops into simpler parts. This increases throughput not only by letting the separate loops operate concurrently, but also because the logical effort of the most difficult loop sets the theoretical maximum speed. It is often easy to allocate parts of a function into concurrently operating loops. For example, delivering duplicate data items to many output paths concurrently with high throughput fits well into a branching structure. Each small branch retains high throughput, delivering its data items to a small number of successors that will, in turn, deliver data to still more successors, and so on to their ultimate destinations.

3. Step Three: Get Wire Lengths from Layout

Because wire capacitance has a major impact on transistor widths, good estimates of wire capacitance are important. We have repeatedly tried and failed to estimate wire lengths prior to layout; our estimates were uniformly too short. The reason seems to be that wires pass not only between modules, but also deliver information inside the modules in unexpected ways. Nevertheless, first estimates of wire length suffice to guide the choice of transistor widths for initial layout. Confirming and adjusting the chosen transistor widths after layout seems wise.

4. Step Four: Balance Transistor Widths

This is a practical paper about how to match the delays of the different gates in loop circuits. In theory the task is relatively simple, and indeed, this paper offers a simple way to use SPICE to do the computations involved. Before introducing the calculation method we first review the underpinnings of theory. More information on logical effort theory is available in [7]. For a short and dense summary, please see Appendix A.

4.1. The Complexity of Stages

One pipeline control stage can be more complex than other stages for two reasons. First, it may drive a load larger than normal, perhaps because its associated data path is wider than normal and the extra latches present extra load. Indeed, the latch load of wide data paths is the largest component of load faced by many control circuits. Second, a complex control stage may perform extra logic functions. For example, the junction of two pipelines requires an extra AND function to combine corresponding data elements from each. The most complex GasP stages do data conditional branch, data conditional merge and arbitration.

Is the load presented to the control by latches fixed? One might attempt to include latches in the uniform delay assumption, selecting their transistor widths for the specific delay chosen. However, because there are so many latches in most designs, we prefer to treat them as fixed loads like wires. Moreover, the pass transistors in our latches are almost of minimum size, and so the fixed capacitance of the wires carrying control signals to them is a major consideration. We simplify the design task by treating latches as a fixed load. With this treatment, the number of latches driven, like the capacitance of wires, has no impact on the theoretical maximum speed, given wide enough transistors. In practice, the logical effort of GasP circuits is so low that the single latch drive amplifier included in many drawings of GasP circuits suffices to drive the broadest data paths. As we shall see later in an example, latch load may have a major impact on transistor width but has little effect on speed.

4.2. Loop Circuits

It is easy to see that every asynchronous system must contain logical loops. Left on its own and unimpeded by external delays, an asynchronous system runs at its own characteristic speed; its internal loops oscillate. It follows that at least some, if not all, of its internal loops must have

an odd number of logical inversions. Any loop with an odd number of inversions will oscillate at a frequency that reveals much about the delay of its logic gates. Such a loop is called a “ring oscillator” and forms the simplest asynchronous system. In CMOS, three inverters suffice for a ring oscillator; a loop of only one CMOS inverter will not oscillate.

Interesting asynchronous systems contain multiple ring oscillators coordinated with logical AND functions. The coordination function is usually done either by a Muller C-element or by a NAND gate. In the Micropipeline control circuit of Figure 4, for example, each logical loop passes through two Muller C-elements and one inverter; the Muller C-elements knit the individual loops into a pipeline control.

4.3. Electrical Properties of Loop Circuits

A simple loop circuit consists of three inverters driving a load as shown in Figure 1A. Were there no load, the last inverter in the ring would drive only the input of the first inverter, fastest oscillation would result from three identical inverters, and the frequency would be limited only by the inverters themselves. The delay in each inverter would indicate how long that inverter takes to pass enough charge to change the state of its own output terminal and the input terminal of the next identical inverter. It is well known that the frequency of oscillation without external load is almost independent of the transistor width chosen for the identical inverters because increasing their width not only increases their drive capability but also their load in almost equal proportion.

With an external load, the last inverter must drive not only the input of the first inverter, but also the external load. In this case, maximum speed comes from a geometric progression of sizes. Let us use L to represent the external capacitive load and C_j to represent the input capacitance of inverter j . Then h_j , the ratio of the load it drives to its input capacitance, also called its *electrical effort*, is given by $h_j = C_{j+1}/C_j$. Electrical effort is sometimes known as

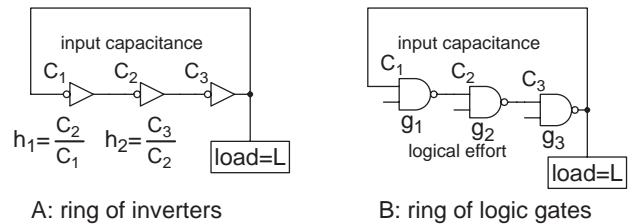


Figure 1. Rings with Load L

“step-up,” but we avoid that term as possibly ambiguous. Working around the loop one writes: $h_1 C_1 = C_2$, $h_2 C_2 = C_3$, $h_3 C_3 = (L + C_1)$, or $h_1 h_2 h_3 C_1 = (L + C_1)$, or, for the general case with n inverters, $C_1 \prod(h_j) = (L + C_1)$. One can also write this result as $C_1 = [L] / [\prod(h_j - 1)]$.

The delay of an inverter depends on the electrical effort, h_j , that it bears. More electrical effort implies more load in proportion to drive capability and hence more delay. Thus reducing the electrical effort will increase speed. However, as the equation for C_1 shows, reducing electrical effort also increases the widths of the transistors. As the combined electrical effort, $\prod(h_j)$, approaches one, the width of the transistors becomes infinite, rendering the load negligible in comparison to the drive capability of the inverters. The maximum possible speed is that of the ring oscillator without external load.

For reasonable values of electrical effort in each inverter, say 2 or more, C_1 is small compared to L , and the input of the inverter string, C_1 , takes relatively little current compared to the load, L . This is the beauty of self-resetting logic; like the loop of Figure 1B, it steals a small current from an already large load, using it to drive logic inputs that would otherwise load down their predecessor logic gates. This is the principle upon which GasP’s performance rests.

Now let us examine the situation of Figure 1B where each part of the ring also performs logic. A logic gate can amplify electrical signals less well than an inverter; how much less well is called its *logical effort* and is represented as g_j . Logical effort acts like electrical load; in a mathematical sense, electrical effort and logical effort are equivalent. Even without the external load, the ring of Figure 1B will oscillate more slowly than that of Figure 1A because of the collective logical effort of its logic gates. Indeed, one way to measure logical effort is from the oscillation frequency of a ring of identical logic gates on a chip.

The delay of a logic gate depends on the product of its logical effort and its electrical effort, a product called simply its *effort*, $f_j = g_j h_j$. A good model of the gate delay is $d_j = f_j + p_j$, where p_j is a *parasitic delay* inherent in the geometry of the gate. By keeping the effort, f_j , constant from logic gate to logic gate we will match at least the variable part of the gate delay, ignoring p_j . Some support for ignoring p_j comes from the similarity of the logic in different loops in GasP, most of which contain a single NAND function and two inverters. Further help

comes from “tuning” widths slightly to compensate for unusually large parasitic delays when they occur.

If the effort, f_j , of each gate is held the same, a gate’s electrical effort, h_j , must vary in inverse proportion to its logical effort, g_j , because $h_j = f_j / g_j$. To have similar delay, gates that do much logic, and therefore have high logical effort, must do correspondingly less electrical amplification; logic gates that do little logic shoulder more of the overall electrical amplification task.

4.4. Effort Expectation

There remain a wide range of satisfactory values for the value of f_j chosen for use throughout the design. The reason is that for any modest value of f_j , C_1 is small compared to L , and the input of the inverter string, C_1 , takes relatively little current compared to the external load, L . In this paper we use the term *effort expectation* and the symbol X , for the particular amount of effort assigned to each logic gate; if all f_j are the same, $X = f_j$ and in any case $X^n = \prod(f_j)$. The Logical Effort book [7] uses the term “stage effort” for a similar value of effort calculated to achieve least delay for a particular circumstance. The term effort expectation not only emphasizes that its value for asynchronous loops is arbitrary, but also avoids confusion with pipeline stages.

Substituting into our earlier equations we find $C_1 \prod(f_j / g_j) = (L + C_1)$. Substituting X^n for $\prod(f_j)$ and multiplying by $\prod(g_j)$ gives $X^n C_1 = (L + C_1) \prod(g_j)$ and $C_1 = [L \prod(g_j)] / [X^n - \prod(g_j)]$. The numerator in this expression, $[L \prod(g_j)]$, which is the effective “load,” is larger than L in proportion to the logical effort of the loop, another example of how logical effort plays the same mathematical role as electrical effort. The denominator, $[X^n - \prod(g_j)]$, shows that there is a minimum value of X , a minimum set only by the combined logical efforts of the logic gates in the loop. At this minimum effort expectation some transistor widths become infinite. Surprisingly, the electrical load, L , plays no part in establishing this minimum.

This analysis considers only one external load and omits the impact of branching paths within a loop. Such branching paths act like other external loads, taking current away from the loop and imposing extra effort on its gates, thereby increasing the minimum possible value of effort expectation. Because the algebra is complex, we use numerical methods to discover how various choices of effort expectation affect transistor widths. In practice we

reduce effort expectation as far as possible while preserving “reasonable” transistor widths.

Speed, area and power consumption are all related to effort expectation. If the designer allows each gate to bear more effort it will have smaller transistors but will be slower, consuming less power both because of reduced speed and because of reduced area. If the designer permits each gate to bear less effort, it will need larger transistors to drive its load, the circuit will run faster, but will consume more power. The minimum possible effort expectation, and hence the maximum theoretical speed of the system, is set only by the logical effort of its most complex loop. Thus the choice of effort expectation controls the choice between speed and area. Because the circuits are asynchronous, the same logic design covers a range of choices.

Although the choice of effort expectation is quite arbitrary, useful values for GasP circuits lie in the range 2 to 5, but we like the values 3 and 4 because they simplify human interpretation. Sometimes we use an effort expectation of $8/3$ to simplify layout. In circuits with P transistors generally twice as wide as N transistors, setting $X = 8/3$ makes the widths of individual N and P transistors in an inverter respectively, $1/8$ and $1/4$ of the total load they drive.

5. Automatic Width Calculation

Instead of doing the calculation by hand, one can use SPICE to calculate transistor widths as well as to simulate circuit behavior. To meet the separate needs of width calculation and logic simulation, each prototype logic gate in the basic library needs two simulation models, different inside but sharing identical terminals. The model for analog simulation contains transistors and capacitive loads. The model for width simulation is much simpler: it contains only DC current sources and resistors. These two models for each prototype should be interchangeable; each pair should have the same number of terminals and the same terminal names. A larger circuit that contains many logic gates and the connections among them should be equally meaningful with either library of prototype gates.

Because the width model is just a DC circuit, SPICE can find width values for even very complex circuits very quickly. Of course, any equation solver could calculate widths equally quickly; indeed we’ve used spreadsheets to good effect. The beauty of using SPICE is that it formulates suitable equations directly from circuit interconnections. Moreover, the very same connections used in width models also serve the analog models from which SPICE predicts performance.

After the width simulation has assigned a width to every transistor—a relatively brief computation—the analog simulation proceeds in the usual way. The width simulation chooses a width for each transistor proportional to its total load, considering also the logical effort of its logic gate. Given such widths, SPICE reports that all logic gates have nearly uniform delay and that all voltage transitions have nearly the same shape. This is hardly surprising, given that SPICE is itself based on linear circuit models. Nodes with large parasitic delays exhibit slower transitions, and in very fast circuits, smaller voltage excursions.

5.1. The Width Model

Most circuit nodes have only one driver but many receivers; the driver is the output terminal of a logic gate, while each receiver is the input terminal of another logic gate. The capacitive load of the node is the total input capacitance of all the input terminals on the node plus the stray capacitance of the wire that forms the node. The width simulation must calculate a suitable width for the transistors in the driver to accommodate this total load, taking into account the logical effort of the driver.

To inform it of its total load, the width simulation must deliver a sum of individual loads to each driver. Remembering that the sum of DC currents injected into a node must also exit the node, let us use simulated DC current to model capacitive load.

Using simulated current to represent load leads to the only non-intuitive part of the whole method. The non-intuitive aspect is that a logic gate’s width model turns its flow backwards: logic *inputs* become current *sources*, logic *outputs* become current *sinks*.

Because the logic inputs to a logic gate are loads, logic *input* terminals, like “a” and “b” in Figure 2A and Figure 2B, become *sources* of simulated DC current in width models, as shown in Figure 2C. The amount of simulated current that a width model forces out of a logic gate’s input terminal is proportional to the capacitive load of the logic gate at that input.

Because the logic outputs of a logic gate must provide drive, logic *output* terminals, like “c” in Figure 2A and Figure 2B, become *sinks* of DC current in its width model, as shown in Figure 2C. The total simulated DC current from all logic inputs on a given node enters the logic output terminal of the logic gate that drives that node. Thus the simulated current coming into the logic output terminal of the driving logic gate represents the total load it must drive.

We find it convenient to scale the simulated DC currents in the width model so that one simulated amp represents

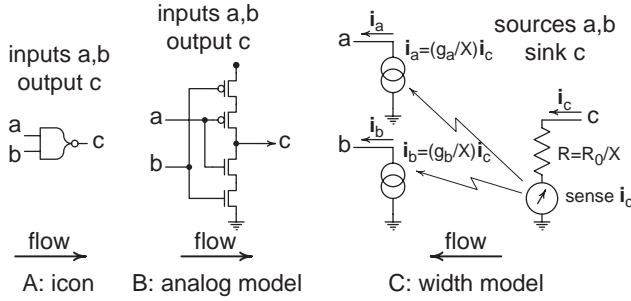


Figure 2. Models for Muller C-element

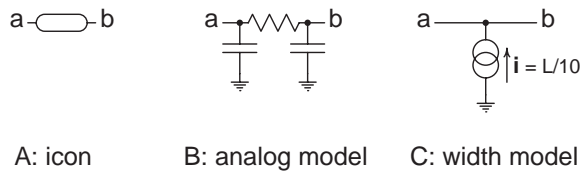


Figure 3. Models for Wire

the capacitive load of one micron of transistor gate width. For example, the model of an inverter with P-type and N-type transistors with fixed widths of 8 microns and 4 microns, respectively, would include a 12 amp simulated current source to model its input node. This choice of scale may cause some wires in the width model to carry hundreds of amps, but never mind; such simulated amps are just floating-point numbers in a computer, and so there's no risk of melting the wire! The vast difference between amps in the width model and micro-amps or milli-amps in the analog model helps to distinguish the models.

The width model also represents the stray capacitive loads of wires as shown in Figure 3. Our analog simulation model of Figure 3B represents each wire as a "II" section with two shunt capacitors and a series resistor between. Our width model, shown in Figure 3C, represents the stray capacitance of the wire as a current source proportional to the length of the wire. These models seem good enough for wires with significant capacitance. Models for wires with significant resistance remain unexplored.

There is also a proportionality constant for the stray capacitance of wire. The parameters provided by MOSIS for their 0.35 micron process indicate that each micron of wire length is between 10 and 15 times easier to drive than a micron of transistor gate width. We have confirmed these numbers using chips with ring oscillators that measure the relative difficulty of driving inverters and driving wire. To be conservative we use a wire model with the more-difficult-to-drive value of 10, delivering one simulated amp for every 10 microns of wire length. Because this model

over-estimates the wire load, it computes some excess of width for transistors that drive long wires.

5.2. Logical Effort Parameters

The proper width for the transistors in a particular logic gate depends not only on the load that it drives, but also on the topology of the logic gate. As the width calculation adjusts the widths of transistors in one logic gate, the load they present to other logic gates changes, requiring adjustment of their transistor widths, and so on. The width model for a logic gate accommodates such multiple adjustments, because the amount of simulated current that the width model forces out of its logic input terminals depends on the amount of simulated current coming in at its logic output terminal.

The width model uses a Current-Controlled Current Source (CCCS) to force simulated current out of each logic input terminal. The gain of the CCCS for each input terminal depends on the topology of the logic gate, expressed as its logical effort, and on the effort expectation, X . Figure 2C shows two CCCS devices, one for each input of the Muller C-element depicted, both of which respond to the measured current as indicated by the jagged arrows. In the figure, g_a and g_b are the logical efforts of inputs a and b respectively. The width models let the user set the effort expectation, X , of each gate separately, but we generally treat it as constant for all logic gates.

5.3. Special Cases

Some nodes have more than one driver. For example, several points of a multiplexor may all drive a common output node, though not concurrently. In the width model for such circuits, which driver should accept the simulated current forced out of the logic inputs of the driven logic gates? Simply dividing the simulated current equally between the several drivers would calculate too small a width for each.

In the width model a node with multiple drivers in parallel delivers its current to a fictitious circuit element instead of to its drivers. The width models for drivers intended to work in parallel, such as multiplexor-points, omit the terminating resistor of Figure 2C. Instead, the fictitious circuit element holds a single resistor that serves all drivers on its node, creating a voltage on the node proportional to the total driven load. Each multiplexor-point observes this voltage on its logic output terminal and uses a Voltage-Controlled Current Source (VCCS) to force simulated current out of its logic input terminals proportional to the voltage on its logic output terminal.

Because the fictitious element appears in the width model prototype library, it requires a companion in the analog simulation prototype library. Although the width model for the fictitious element holds a resistor, the analog simulation model of the fictitious element is empty.

Another special case is the “keeper,” a pair of weak back-to-back inverters placed on a node to retain its state. The main drive transistors for a node can easily overpower its keeper, but their ability to drive other loads is slightly degraded by the presence of the keeper. The width model for a keeper should represent both the load of its input inverter and the current the keeper delivers from its output inverter to oppose change. Both of these currents are loads on the node represented as simulated currents coming out of the keeper. Keepers are an exception to the rule that logic gate outputs are current sinks.

The final special case comes about because of the minimum width permitted for layout of transistors. We often find that a width simulation proposes widths for some transistors narrower than the minimum width allowed by layout rules. To prevent this, the width model of a logic gate insists that each transistor be of at least the minimum width. The CCCS associated with each logic gate input includes a minimum simulated current value that corresponds to the minimum layout width permitted for transistors. The result, of course, is that the width model specifies at least the minimum width for the transistors in such logic gates and produces that minimum amount of simulated current at its inputs. However, very small logic gates may end up with wider transistors than ideally necessary and thus have less delay than expected. Such gates can cause problems in circuits where uniform delays are essential.

6. An Example

Let us apply this method to the control of Figure 4 which represents consecutive stages P, Q, and R of a very long Micropipeline with wire loading omitted. Each stage consists of a Muller C-element and an inverter. Stage Q of this Micropipeline bears a load ten times that of the other stages, perhaps because it drives a broader data path. Although the same control circuit topology appears in every stage, a width model simulation shows that the heavily-loaded stage must have wider transistors than the other stages to drive its greater load with similar delay. The results of such a width simulation appear in Figure 5 and Figure 6.

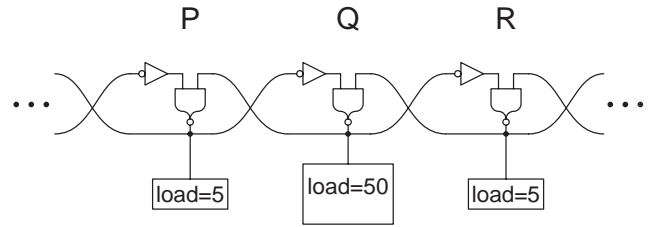


Figure 4. A Micropipeline - stage Q heavily loaded

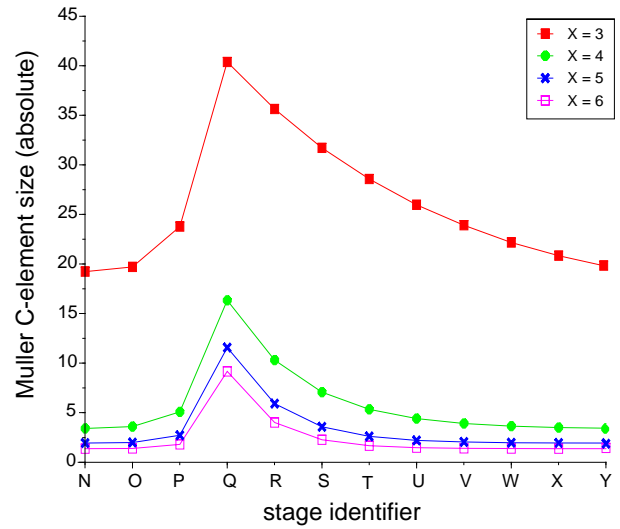


Figure 5. C-Element Size vs Effort Expectation

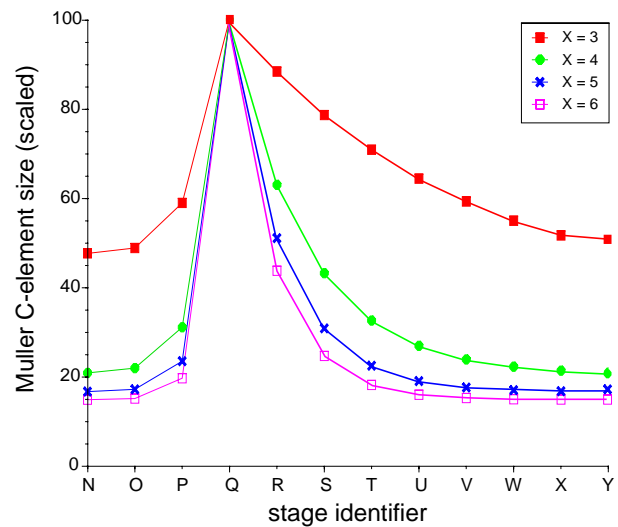


Figure 6. Scaled Size vs Effort Expectation

Both figures illustrate the widths of the Muller C-element transistors for effort expectations of 3, 4, 5 and 6. Figure 5 shows the input capacitance of one input of the Muller C-element in each stage in the same units as those used to measure the loads of 5 and 50. Figure 6 shows the same results normalized to the size of the largest Muller C-element, exposing the relative shapes of the curves. As one might expect, the figures show that the Muller C-element in the heavily-loaded stage Q must have wider transistors than required elsewhere in order to have similar throughput.

It may be less obvious why stages P and R immediately adjacent to stage Q also require enlarged transistors, as do stages O and S next to *them*, and so on. Although the adjacent stages all bear the same data-path load, each must also drive the larger Muller C-element in an adjacent stage. Thus the calculation yields the “tapered” curves seen in the figures.

When we first saw the taper of Figure 5 and Figure 6 its asymmetry came as a complete surprise to us. Why are stages R and S ... *after* the heavily-loaded stage more affected than stages P and O ... *before* it? The reason is that the forward gain per control stage of this Micropipeline control circuit exceeds its reverse gain. The gain differs because both the inverter and the C-element provide gain in the forward direction, but only the Muller C-element provides gain in the reverse direction. Thus it is harder for a stage to drive wide transistors in a predecessor than in a successor stage, and so the widths taper off more slowly towards the right of the graph.

It’s also interesting to note the differences between the curves for different effort expectations. The normalization of Figure 6 makes clear the differences in the shapes of the curves; higher effort expectations imply that the designer is willing to wait longer and so each logic gate can offer more gain. Thus for higher effort expectations the transistor widths taper off more quickly. Notice how reluctantly the top curve changes. With an effort expectation $X = f_j = 3$, and a logical effort, $g_j = 2$, the Muller C-element can support an electrical effort, $h_j = f_j / g_j$; in other words, it can provide a gain of only $3/2 = 1.5$. Therefore the load at its input terminals is $(2/3)L$, which adds to the small external load on stage R. Neglecting the small load of the inverter in stage S, stage R finds the combination, $5 + (2/3)50 = 38.3$, nearly as hard to drive as the load borne by stage Q. The situation is little different for stages S and T and so on. Algebraic analysis shows that, including the load from the inverters, the width calculation for this circuit cannot converge if the effort expectation is less than

$1 + \sqrt{3} = 2.732$. The top curve tapers slowly because the effort expectation is close to that limit.

The ratio of the load size to the Muller C-element size for stage Q is also interesting. The ratio tells, in effect, how much gain the Muller C-element in the heavily-loaded stage actually achieved, considering that it must also drive adjacent stages. The numbers are $50/9.16 = 5.46$ instead of 6, $50/11.6 = 4.33$ instead of 5, $50/16.3 = 3.07$ instead of 4, and $50/40.4 = 1.24$ instead of 3. Pushing the limits of expected effort costs dearly.

7. Conclusion

We offer this paper not only in the hope that it may help others design fast circuits, but also as an explanation of the methods we use to achieve the results reported in other papers at this conference [1][2][6]. Our design method uses careful calculation of transistor widths to give all gates nearly uniform delay. Even though we determine transistor widths *after* completing the logic design and layout, the logic designer can take advantage of the uniform delay assumption to reduce design complexity.

By using a fixed number of stages in each asynchronous loop, the logic designer obtains a system in which all parts cycle at about the same speed. The logic designer is encouraged to simplify the logic in the most complex logic loops, because they will establish the minimum delay possible for all gates.

The task of balancing transistor widths is easy. We have used hand calculations and spreadsheets for the calculations in addition to the simulation model described in this paper. In practice the hardest part is making accurate estimates of wire loading; post-layout knowledge of wire lengths appears to be very important. Capacitive wire loads increase the width of transistors required to drive them but have no bearing on theoretical maximum speed. The theoretical maximum speed is set only by the combined logical effort of the most complex loop in the design. We have no experience with wires with significant resistive delay.

One always wonders how to do the fifth step of the process: verify correct operation. The width simulations described here calculate transistor widths that appear wonderful when used in an analog simulation. But are we fooling ourselves by validating one model with another? How reliable is an analog SPICE simulation? How well does it model real silicon? Our group continues to build a series of relatively simple test chips, each teaching us something more about our design methods. We have tested two substantial chips for which we calculated transistor

widths as outlined here. Both chips use tapered control stages to meet the requirements of data path loading and logic function that vary from stage to stage. If operating speed is a measure, then both chips validate width simulation. Unfortunately, we cannot observe the shape of on-chip signals nor the on-chip delay of individual logic gates, but the high throughput of our chips nevertheless provides assurance that we can indeed avoid excessive delay in pipeline stages that contain complex control logic and in stages that drive heavy loads.

8. Acknowledgments

The authors would like to thank the many reviewers of early versions of this paper. Their comments and suggestions have led to a much improved final product. We would also like especially to thank Wes Clark and David Harris for their helpful feedback.

9. References

- [1] W. S. Coates, J. K. Lexau, I. W. Jones, S. M. Fairbanks and I. E. Sutherland, "FLEETzero: An Asynchronous Switching Experiment," *Proc. of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2001.
- [2] J. Ebergen, "Squaring the FIFO in GasP," *Proc. of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2001.
- [3] V. Narayanan, B. A. Chappell, and B. M. Fleischer, "Static Timing Analysis for Self Resetting Circuits," *Proc. of the International Conference on Computer-Aided Design*, 1996.
- [4] Proebsting, private communication.
- [5] Proebsting, US Patent 5,343,090, "Speed Enhancement Technique for CMOS Circuits," August 30, 1994.
- [6] I. Sutherland and S. Fairbanks, "GasP: A Minimal FIFO Control," *Proc. of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2001.
- [7] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann Publishers, Inc., 1999.

Appendix A

Logical Effort - A Short and Dense Summary

The transistor model used throughout the Logical Effort book [7] as well as in this paper describes the delay of a logic gate as the sum of two components. The first component is directly proportional to the external load on the logic gate and inversely proportional to the width of the transistors in the logic gate. The second component of delay is a parasitic delay due mostly to the stray capacitance of transistor drain nodes. Changing the width of the transistors in the gate has little effect on parasitic delay. The reason is that although wider transistors provide more drive, they also have greater drain capacitance, and the two effects very nearly cancel out.

Given this linear model, a graph of logic gate delay, d , as a function of *electrical effort*, $h = C_{out}/C_{in}$, is the straight line $d = gh + p$, where p is a parasitic fixed delay and g is the *logical effort* of the gate. The logical effort tells how much the delay increases with increasing load. Mathematically, logical effort and electrical effort are interchangeable, indeed, the similar roles of g and h in the delay equation led to the names "logical effort" and "electrical effort." The electrical effort of an inverter is sometimes called its step-up ratio C_{out}/C_{in} , but for more complex logic gates the term "step-up ratio" may have a less precise meaning, and so we avoid using the term. The product of logical effort and electrical effort $f = gh$ is called simply the *effort* of the gate, and in suitable units it is a measure of the variable part of the gate delay $d = f + p$.

One can use data from SPICE simulation to plot delay against electrical effort for a simulated logic gate. The slope of the resulting curve reveals the logical effort of the gate. Test chips with ring oscillators made from identical gates with identical loading on each also give frequency data that reveals logical effort.

Logical effort teaches that an inverter is a better electrical amplifier than a logic gate, or to put it the other way around, a logic gate is less good as an amplifier than an inverter is. A logic gate's electrical gain is less than an inverter's either because its topology places transistors in series that conduct less well than would a single transistor, or because its topology requires extra transistors in parallel that contribute extra input capacitance to its inputs, or for both reasons. The logical effort of a logic gate describes how much worse it is as an amplifier than an inverter is. The logical effort of a logic gate is normalized so that an inverter has logical effort of one.

Logical effort is a useful concept because it depends only on the topology of the logic gate and the relative widths of its transistors. The logical effort of a logic gate remains constant if all its transistors change width in equal proportion. The logical effort of different inputs of a logic gate may differ, depending on their particular circuit topology. Logical effort is also useful because the logical effort of a string of gates is the product of their individual logical efforts.

The logical effort of a logic gate describes, in terms of gain, the cost of its logic function. In one family of ordinary static CMOS, the logical effort of a two-input NAND gate is $4/3$; of a two-input NOR is $5/3$; of a two-input Muller C-element is 2, of the data inputs of a multiplexor of any width is 2; and of a single logic gate two-input XOR circuit is 4. A NOR gate is more costly than a NAND gate because the P-type transistors it places in series are less effective than the N-type series transistors in the NAND. An XOR gate is very costly because it uses parallel strings of series transistors to make its logic output change for any change at either input.

Using the linear model of delay, one can write an expression for the total delay in a string of logic gates by adding up their individual delays. The delay in each logic gate is, of course, a function of both its transistor width and its loading, a substantial part of which depends on the width of the transistors that it drives. Setting the partial derivative of total delay with respect to each transistor width equal to zero gives a set of equations whose solution minimizes the total delay.

Study of such equations shows that: A) if the transistors in a particular logic gate are too wide, that logic gate will operate quickly but will put more load on the logic gates that drive it, retarding their action; B) if the transistors in a logic gate are too narrow, that logic gate will operate too slowly even though the logic gates that drive it act more quickly; and C) transistors of proper width balance the drive capability of each logic gate and the load it imposes on its predecessor so as to equalize the effort, f , of each logic gate and therefore the variable part of each logic gate's delay. Taking the derivative of delay removes the parasitic delay from consideration because it remains the same for all transistor widths.

Experienced designers know that driving a large electrical load requires a series of amplifying inverters

whose transistor widths form a "horn" from narrow to wide. The shape of the horn is exponential so that each logic gate drives some multiple of the load it imposes on the previous logic gate, holding h constant. The gain of the amplifiers overcomes electrical effort.

Although it is widely known how to use the gain of inverters to overcome the electrical effort of driving large capacitive loads, it is less widely known that gain is also necessary to overcome logical effort. It proves useful to define the *total effort* of a string of logic gates as the product of their logical efforts multiplied by the electrical effort, $H = C_{out}/C_{in}$, imposed on the entire string; the upper case H indicates the total value for the string rather than for an individual gate. Thus $F = H \prod (g_j)$. Because logical effort and electrical effort are mathematically interchangeable, only the total effort of a string of logic gates matters. The logical effort book [7] explains how to find the best number of logic gates to use for a particular total effort. The best number of logic gates balances parasitic delay and effort delay. Moreover, according to the linear model, the string of logic operates fastest if its total effort is allocated uniformly to its logic gates. Where n is the number of logic gates, f_j is the effort of logic gate j , and F is the total effort, $f_j = F^{1/n}$.

Distributing the total effort uniformly throughout the string of logic gates makes sense even without the mathematics. Combinational logic gates, being poor amplifiers, should bear correspondingly less of the electrical effort, whereas inverters, being good amplifiers and free of combinational logic, should bear more of the electrical effort. A string of logic gates achieves least overall delay if the width of their transistors forms a horn, but the horn's shape must accommodate the logical effort of each gate. The horn's shape should hold constant the effort on each gate, where effort is the product of logical and electrical effort, $f = gh$.

A gratifying result from logical effort calculations is that the order of gates in a logic string affects neither the total effort of the string nor the minimum achievable delay for the string. Of course, the order may affect layout area or, because of logical inversions, the logic function performed by the string.

The Export of Cryptography in the 20th Century and the 21st

Whitfield Diffie and Susan Landau

Introduction by Whitfield Diffie and Susan Landau

Our efforts have focussed on two parallel tracks. The first has been to remove the government restrictions on the deployment of cryptography. Simultaneously we have worked on new issues affecting security, including new threats, new policy problems, and new technical directions.

This paper does not report the results of research in the usual sense. It reports the outcome of seven years work whose results are measured not in increased knowledge but in an improved environment for Sun development and Sun marketing. The lifespan of Sun Microsystems coincides closely with the era of globalization. To compete in the global economy, Sun, like other large companies, must have access to markets all over the world. One important limitation on this access has been US export-control regulations. The report describes the results of a decade-long battle to reconcile the export regulations with the realities of modern business and technology. Success was achieved in the year 2000 with sweeping improvements in the regulations. This environment will lead to new products and improved functioning of existing products both in Sun's core business areas and in developing areas such as wireless communications.

For most of the era of electronic communication, encryption -- protecting communications by scrambling them -- was largely the province of the government. Before modern electronics, encryption was too expensive for widespread business use. Most development was done by the government and kept secret for exclusive government use. Cryptography was treated as a weapon under the export-control laws and encryption systems could not be exported for commercial purposes, even to close allies and trading partners.

During the 1980s and 1990s cryptography emerged from its role as an obscure technology used by the government to protect its communications to a necessary underpinning of Internet commerce. The rise of the personal computer and the Internet changed cryptography from an exotic military-only technology into a critical technology of Internet commerce. Despite this, the government was slow to accept the new reality. Industry efforts to develop and use cryptography were thwarted by government export-control regulations, which emerged as the dominant government influence on the manufacture and use of encryption technology. Making repeated attempts to continue its domination of the field, by the late 1990s the US government held a stance that was barely tenable in the rest of the world. Influences varying from the rise of open-source software to European indignation at evidence of US communications intelligence came together to force a change.

PAPERS, BOOKS, REPORTS, OP-EDS:

S. Landau, S. Kent, C. Brooks, S. Charney, D. Denning, W. Diffie, A. Lauck, D. Miller, P. Neumann and D. Sobel, "Codes, Keys and Conflicts: Issues in U.S. Crypto Policy," ACM Press, 1994.
http://info.acm.org/reports/acm_crypto_study.html

S. Landau, S. Kent, C. Brooks, S. Charney, D. Denning, W. Diffie, A. Lauck, D. Miller, P. Neumann and D. Sobel, "Crypto Policy Perspectives," Communications of the ACM, Vol. 37 (August 1994), pp. 115–121.

Susan Landau and Whitfield Diffie, "Cryptography Control: FBI Wants It, but Why?," Christian Science Monitor, October 6, 1997.
<http://www.csmonitor.com/durable/1997/10/06/opin/opin.2.html>

Whitfield Diffie and Susan Landau, Privacy on the Line: the Politics of Wiretapping and Encryption, MIT Press, 1998.
<http://mitpress.mit.edu/book-home.tcl?isbn=0262041677>

"Standing the Test of Time: The Data Encryption Standard," Notices of the American Mathematical Society, March 2000, pp. 341–349.
<http://www.ams.org/notices/200003/fea-landau.pdf>
Reprinted, in translation, in "Surveys in Applied and Industrial Mathematics," TVP Publishers (Moscow), Vol. 7, No. 2 (2000), pp. 240–258.

"Communications Security for the Twenty-First Century: the Advanced Encryption Standard," Notices of the American Mathematical Society, April 2000, pp. 450–459.
<http://www.ams.org/notices/200004/fea-landau.pdf>
Reprinted, in translation, in "Surveys in Applied and Industrial Mathematics," TVP Publishers (Moscow), Vol. 7, No. 2 (2000), pp. 259–281.

"Advanced Encryption Standard Choice is Rijndael," Notices of the American Mathematical Society, January 2001, p. 38.
<http://www.ams.org/notices/200101/200101-toc.html>

The Export of Cryptography in the 20th Century and the 21st

**Whitfield Diffie and Susan Landau
Sun Microsystems Laboratories
Palo Alto, California**

November 2000

On the 14th of January 2000, the Bureau of Export Administration issued long-awaited revisions to the rules on exporting cryptographic hardware and software. The new regulations, which grew out of a protracted tug of war between the computer industry and the U.S. Government, are seen by industry as a victory. Their appearance, which has been attended by both excitement and relief, presents an appropriate occasion for examining the evolution of export control in the cryptographic area and considering its impact on the deployment of privacy-protecting technologies within the United States. In response to European Union export liberalizations, on October 19, 2000, the U.S. regulations were further eased, but the more significant modifications occurred in the January changes.

Before the electronic age, all ‘‘real-time’’ interaction between people had to take place in person. Privacy in such interactions could be taken for granted. No more than reasonable care was required to assure yourself that only the people you were addressing – people who had to be right there with you – could hear you. Telecommunications have changed this. The people with whom you interact no longer have to be in your immediate vicinity; they can be on the other side of the world, making what was once impossible spontaneous and inexpensive. Telecommunication, on the other hand, makes protecting yourself from eavesdropping more difficult. Some other security mechanism is required to replace looking around to see that no one is close enough to overhear: that mechanism is cryptography, the only security mechanism that directly protects information passing out of the physical control of the sender and receiver.

At the turn of the 20th century, cryptography was a labor-intensive, error-prone process incapable of more than transforming a small amount of written material into an encoded *ciphertext* form. At the turn of the 21st, it can be done quickly, reliably, and inexpensively by computers at rates approaching a billion bits a second. This progress is commensurate with that of communications in general yet the fraction of the world’s communications protected by cryptography today is still minuscule. In part this is due to the technical difficulty of integrating cryptography into communication systems so as to achieve security, in part to an associated marketing problem. Proper implementation of cryptosecurity requires substantial up-front expenditure on infrastructure whereas most of the benefit is unavailable until there is nearly ubiquitous coverage, a combination that deters investment. These factors result in a lack of robustness of the market that makes it prey to a third factor: political opposition.

As telecommunication has improved in quality and gained in importance, police and intelligence organizations have made ever more extensive use of the possibilities for electronic eavesdropping. These same agencies now fear that the growth of cryptography in the commercial world will deprive them of sources of information on which they have come to rely. The result has been a struggle between the business community, which

needs cryptography to protect electronic commerce and elements of government that fear the loss of their surveillance capabilities. Export control has emerged as an important battleground in this struggle.

Background

In the 1970s, after many years as the virtually exclusive property of the military, cryptography appeared in public with a dual thrust. First came the work of Horst Feistel and others at IBM that produced the U.S. Data Encryption Standard. DES, which was adopted in 1977 as Federal Information Processing Standard 46, was mandated for the protection of all government information legally requiring protection but not covered under the provisions for protecting classified information – a category later called “unclassified sensitive.”

The second development was the work of several academics that was to lead to *public-key cryptography*, the technology underlying the security of internet commerce today. Public-key cryptography makes it possible for two people, without having arranged a secret key in advance, to communicate securely over an insecure channel. Public-key cryptography also provides a digital signature mechanism remarkably similar in function to a written signature.¹ The effect of new developments in distinct areas of cryptography was to ignite a storm of interest in the field, leading to an explosion of papers, books, and conferences.

The government response was to try to acquire the same sort of “born classified” legal control over cryptography that the Department of Energy claimed² in the area of atomic energy. The effort was a dramatic failure. NSA hoped an American Council on Education committee set up to study the problem would recommend legal restraints on cryptographic research and publication. Instead, it proposed only that authors voluntarily submit papers to NSA for its opinion on the possible national-security implications of their publication. (Landau83)

It did not take the government long to realize that even if control of research and publication were beyond its grasp, control of deployment was not. Although laws directly regulating the use of cryptography in the U.S. appeared out of reach – and no serious effort was ever made to get Congress to adopt any – adroit use of export control proved remarkably effective in diminishing the use of cryptography, not only outside the United States but inside as well.

Export Control

The export control laws in force today are rooted in the growth of the Cold War that followed World War II. In the immediate post-war years the U.S. accounted for a little more than half of the world’s economy. Furthermore, the country was just coming

1 In recognition of the increasing importance of electronic commerce, in June 2000, President Clinton signed into law the Millennium Digital Commerce Act, Public Law 106–229, which establishes the legal validity of “electronic signatures.”

2 The courts have never ruled on the constitutionality of this provision of the Atomic Energy Act. In 1997, the Progressive magazine challenged it by proposing to publish an article by Robert Morland entitled “The H-bomb Secret, how we got it, why we’re telling it.” After the appearance of an independent and far less competent article on how h-bombs work, the government succeeded in having the case mooted and leaving the impression in the popular mind that it would have won. In fact, the virtual certainty that it would have lost is undoubtedly why it acted as it did.

off a war footing, with its machinery of production controls, rationing, censorship, and economic warfare. The U.S. thus had not only the economic power to make export control an effective element of foreign policy but the inclination and the regulatory machinery to do so.

The system that grew out of this environment had not one export control regime but two. Primary legal authority for regulating exports was given to the Department of State, with the objective of protecting national security. Although the goods to be regulated are described as *munitions*, the law does not limit itself to the common meaning of that word and includes many things that are neither explosive nor dangerous. The affected items are determined by the Department of State acting, through the *Office of Defense Trade Controls*,³ on the advice of other elements of the executive branch, especially, in the case of cryptography, the National Security Agency.

Exports that are deemed to have civilian as well as military uses are regulated by the Department of Commerce. Such items are termed *dual-use* and present a wholly different problem from “munitions.” A broad range of goods – vehicles, aircraft, clothing, copying machines – are vital to military functioning just as they are to civilian. If the sale of such goods was routinely blocked merely because they might benefit the military of an unfriendly country, there would be little left of international trade. Control of the export of dual-use articles therefore balances considerations of military application with considerations of foreign availability – the existence of sources of supply prepared to fill any vacuum left by U.S. export bans.

The munitions controls are far more severe than the dual-use controls, requiring individually approved export licenses specifying the product and the actual customer as opposed to broad restrictions by product category and national destination. Legal authority to decide which regime is to be applied lies with the Department of State, which can authorize the transfer of jurisdiction to the Department of Commerce, a process called *commodities jurisdiction*.

Assessing whether a product is military or civilian is not always straightforward. Once we leave the domain of the clearly military (such as fighter aircraft), we immediately encounter products that either have both military and civilian uses or products that can be converted from one to the other without difficulty. The Boeing 707, a civilian airplane, was a mainstay of the world’s airlines during the 1960s and 1970s. Its military derivatives, the C-135 (cargo, including passengers), the KC-135 (tanker), and RC-135 (intelligence platform) have been mainstays of Western military aviation. Recognizing that civilian aircraft might be put to military use and thus bypass export control, the U.S. government nonetheless permitted their export as a business necessity. The allowability of exports was judged on the basis of how dual-use goods were configured and who was to be the customer. Generally speaking a commercial technology that is not explicitly adapted to a uniquely military function can be sold to a non-military customer without excessive paperwork.

The application of export controls naturally depends heavily on the destination for which goods are bound. Applications for export to U.S. allies, such as the countries of Western Europe, are more likely to be approved than applications for exports to neutral, let alone hostile, nations. Clearly the effectiveness of export controls will be vastly magnified by coordination of the export policies of allied nations. During the

3 The ODTIC was originally called the Munitions Control Board and has had various names over the years.

Cold War, the major vehicle for such cooperation among the U.S. and its allies was *COCOM*, the *Coordinating Committee on Multilateral Export Controls*, whose membership combined Australia, New Zealand, and Japan with the U.S. and most western European countries.⁴ Although COCOM existed primarily to prevent militarily significant exports to non-COCOM countries, that did not mean that the COCOM countries exported freely among themselves. Many products that would not be permitted out of COCOM could be sold to other COCOM countries but still required a burdensome export approval process.

Export Status of Cryptography

In the post-WWII period, cryptography was, like nuclear energy, an almost entirely military technology. It is stretching the point only a little to say that insecure analog voice scramblers or hand-authentication techniques that might be found in civilian uses were no more closely related to high-grade military encryption equipment than glowing watch dials or x-ray machines were related to atomic bombs. Not surprisingly, all cryptography, regardless of functioning or intended application was placed in the category of munitions. As the information revolution progressed – particularly as computers began to “talk” more and more to other computers – the argument for dual-use status slowly improved. Telecommunications between humans can be authenticated by combinations of more or less informal mechanisms: voice recognition, dial-back, request to know the last check written on an account, etc. To achieve high security in communication between computers without human intervention, cryptography is indispensable. Nonetheless, cryptography remained in the “munition” category long after this seemed reasonable to most observers.

The importance of the munition/dual-use distinction lies in a difference in licensing procedures and a difference in the criteria for export approval. As munitions, cryptographic devices required individually approved export licenses. Two factors combine to make such licenses antagonistic to commercial use of cryptography. One is time: the weeks or months required to get approval often exceed the time commercial organizations allocate to procurement of even major systems. The other is the requirement to identify the end customer. In much of commerce, manufacturers deal with one or more layers of resellers who may either be unaware of the identities of buyers or unwilling to share their information with their suppliers. Munitions are not only more cumbersome to export but more likely to be denied approval outright. The law regulating military exports makes no provision for the probable effectiveness of export policy. If an export is judged militarily imprudent, it is barred regardless of the likelihood that this action will actually prevent the would-be purchaser from obtaining equipment of the type desired.

Even after the business necessity and thus the dual-use character of cryptography had become clear, the problem of distinguishing military from civilian cryptosystems remained elusive. Some cases were straightforward. Systems specially adapted to work with military communication protocols – such as the MK XII IFF⁵ devices that identify aircraft to military radars – or those whose implementations were ruggedized for field use or satisfied arcane military specifications against radiation leakage could safely be classified as military. But what about cryptosystems running in ordinary commercial

⁴ For some reason, Iceland was not included.

⁵ Identification Friend or Foe

computing equipment in ordinary office environments? Such equipment performs very similarly whether in a general's office or in a banker's.

The challenge of export control is to develop a policy that interferes as little as possible with international trade while limiting the ability of other countries to develop military capabilities that threaten U.S. interests. This requires setting rules that distinguish military uses of technology from civilian ones. In the case of cryptography, the initial attempt was to classify cryptosystems as military or civilian by strength, much as guns might be classified by caliber. Small arms have civilian applications — from hunting and target shooting to personal protection and public safety — whereas artillery is purely military. The distinction, however, proved far harder to make in the case of cryptography than of firearms. A cryptographic system adequate to protect a billion-dollar electronic funds transfer is indistinguishable from one adequate to protect a top-secret message.

The Impact of Export Control on Cryptography

As the U.S. share of the world's economy has declined over the past five decades, export controls have become less effective as a mechanism of U.S. foreign policy. Worldwide growth of manufacturing capacity, particularly in military technology, has made many more products available from non-U.S. sources, while the associated growth of markets outside the U.S. has meant that the cost to U.S. businesses of export controls is far greater. In 1950, it cost U.S. companies little to be prevented from exporting something for which there were few foreign customers. Today, with a majority of potential customers outside the U.S., a product's exportability can make the difference between success and failure.

This change in impact of export controls has changed their role. Export controls on cryptography have come to be used at least as much for their effect on the domestic market as the foreign one. Three factors have made this possible:

- The export market in computer hardware and software is huge. The typical American computer company makes more than half its sales abroad and must manufacture exportable products to be competitive.
- Security is always a *supporting feature*; no system exists for the primary purpose of being secure. To be usable and effective security must be integrated from scratch with the features it supports. Even when it is feasible, adding cryptography to a finished system is undesirable.
- Making two versions of a product is complicated and expensive, particularly when, as is typically the case, domestic and foreign products must interoperate. Making a more secure product for domestic use, furthermore, points out to foreign customers that you have given them less than your best. These costs would be borne were the domestic demand for security great enough but so far it has not been.

The result of U.S. export controls has thus been to limit the availability of strong cryptography, not merely abroad but at home.

These policies, which put the interests of intelligence and law-enforcement agencies ahead of other national concerns, were made possible by the dominant, though

far from invincible, position of U.S. companies in the world market for computer hardware and software. Security, though a small component of most computer systems, is often essential. By forbidding the export of systems with good security, the U.S. risks losing the business of security-conscious customers to foreign competition, thereby accelerating the development of the computer industries outside the U.S. The fast-growing computer industry in both Europe and Asia have been only too happy to challenge the U.S. position and, as the growth of the world wide web and electronic commerce made the commercial importance of cryptography more obvious, the U.S. government came under more and more pressure to amend its regulations.

Declining Influence of the Cold War

The export controls on cryptography began to soften in the late 1980s with the transfer to commerce of technologies that were not used to protect long range (and thus interceptable) communications. Change was accelerated by the end of the Cold War at the beginning of the 1990s. A major move in industry's direction was a deal struck in 1992 between the National Security Agency, the Department of Commerce, and computer industry interests. It provided for streamlined export approval for products using selected algorithms with keys no longer than 40 bits.⁶ Initially, two algorithms, both trade secrets of RSA Data Security, a leading maker of cryptographic software, were approved; others were added later.

The problem of keylength is not an issue that lends itself well to compromise and the strength represented by 40-bit keys could hardly have pleased either side. In 1992, a message encrypted using a 40-bit key could be cracked by a personal computer using the crudest techniques in a month or so – hardly sufficient for the lifetime of product plans, let alone personnel records. On the other hand, had such systems been applied to even a few percent of the world's communications they would have created a formidable barrier to signals intelligence. Intercept devices must determine in a fraction of a second whether a message is worth recording. Encryption, broadly applied, seriously interferes with this selection process. If a small enough fraction of messages are encrypted, then being encrypted marks a message as interesting and the message will be recorded. Too many encrypted messages, even weakly encrypted messages, will glut the interceptor's disks and frustrate the collection effort.

Government attempts to control cryptography were not limited to its export strategy. In parallel with the keylength-based formula – which it presumably saw as an interim measure – the U.S. government tried to change the rules to give itself a permanent advantage. In early 1993, it moved to replace the fifteen-year-old, 56-bit, Data Encryption Standard with an 80-bit algorithm that provided a special *trap door* for government access.⁷ Although the standard was adopted, it found few takers and was generally counted as a failure.

Looking back over the 1990s, it is hard to judge whether the Clipper program set

6 If the encryption algorithm is properly designed, then the difficulty of unauthorized decryption is determined by the number of bits in the key; an increase of one bit doubles the cost to the intruder. A good encryption algorithm with a 56-bit key is thus 2^{16} or 65,000 times more difficult to crack than one with a 40-bit key. It is often taken for granted that cryptosystems are as strong as their keys suggest and thus it is common to speak of 40-bit cryptography, meaning both that the keys are 40 bits long and that breaking the system takes approximately a trillion encryptions.

7 This was the infamous *Clipper* system, in which the keys were split and escrowed with Federal agencies. (USDoC, 1994)

the stage for the sequence of confrontations and compromises that followed or whether all were merely consequences of the same technological and market forces. The government made several attempts to establish the principle that it had the right to control cryptographic technology in order to guarantee its power to read intercepted messages.⁸ Over the same period it restructured the export–control bureaucracy and relaxed the regulations.

While cryptography was classified as a munition, a would–be exporter was required either to seek an export license from “State” or request a transfer of jurisdiction to “Commerce.” In 1996 the Department of Commerce Bureau of Export Administration was given direct authority over most cryptographic exports.⁹ In the process, however, the personnel to carry out the new role were transferred from the State Department to the Commerce Department, creating a sense that there was likely to be more change of form than substance.

It is during the reorganization of the export–control machinery that Department of Justice personnel were first introduced into the process. In tune with this introduction, though somewhat ahead of it in time, was a move to shape the terms of debate by talking about signals intelligence in terms that were drawn more from law enforcement and less from the military. It was true then and is true now that most U.S. interception of communications is targeted not against criminals (no matter how loosely this term is used) but against other countries – largely countries we recognize and with many of which we are on friendly terms. Spying on your “friends” is and has always been an uncomfortable activity but much of the discomfort is mitigated by secrecy. A matter never spoken about creates few awkward pauses in conversation but to engage in a public debate one must have something to say. In the debate about encryption it was necessary for the government to say why it was seeking to expand its powers of interception. The answer was to point to an unholy trinity: terrorists, drug dealers, and paedophiles. Entirely lacking in popular support, these groups were in no position to step forth and speak out against being spied on.¹⁰

A rationale has its costs. Giving a law–enforcement rationale made it hard to maintain the intelligence criteria and as the decade wore on, the government’s proposals moved toward the needs of police – individualized court ordered surveillance, perhaps requiring the cooperation of a foreign judicial system – and away from the invisible broad spectrum surveillance that the intelligence community desired. The predictable consequence was that the intelligence agencies, realizing that their needs were not being met, became less vociferous in their support of crypto–control proposals.

In the summer of 1996 the National Research Council released its 18–month study on cryptography policy, *Cryptography’s Role in Securing the Information Society* (the *CRISIS* report), conceived at the time of the key–escrow proposal. Acting on a mandate from Congress, the NRC convened a panel of sixteen experts from government, industry, and science, thirteen of whom received security clearances. The panel was heavily weighted towards former members of the government – the chair, Kenneth Dam,

8 In a related move, the government scored a major victory. The Communications Assistance for Law Enforcement Act of 1994 gave it the power to require communications carriers to build wiretapping into their networks.

9 This was done by adopting Department of State regulations authorizing shippers to go directly to the Department of Commerce for certain categories of goods, rather than submitting their applications first to the Department of State.

10 Whether wiretapping actually plays a significant, let alone indispensable, role in combating any of these phenomena is hard to assess. (Diffie)

for example, had been Under Secretary of State during the Reagan administration – and many opponents of the government’s policies anticipated that the NRC report would support the Clinton administration’s cryptography policy. It did not.

The report concluded that “on balance, the advantages of more widespread use of cryptography outweigh the disadvantages,” and that current US policy was inadequate for the security requirements of an information society (Dam, pp. 300–301). Observing that existing export policy hampered the domestic use of strong cryptosystems, the panel recommended loosening export controls and said that products containing DES “should be easily exportable.” (Dam, pp. 312)

This was not a message the Clinton administration wanted to hear and no immediate effect on policy was discernible. In the fall of 1996 the government announced that a window of opportunity for export would run for the two years 1997 and 1998. During this window, manufacturers would be allowed to export Data Encryption Standard products quite freely if they had entered into memoranda of understanding with the government promising to develop systems with *key recovery*¹¹ during the open-window period. This approach did not even survive its own window. In September 1998, the rules were relaxed to permit freer export of products containing DES or other cryptosystems with keys no longer than 56-bits.

It was a classic example of “too little, too late.” Users around the world had come to feel that cryptographic keys should be 128 bits long. Technical arguments to the effect that there was no point in making the cryptography stronger than the surrounding security system cut little ice with customers. Very strong cryptosystems seem to cost no more to build or run than weaker ones so why not have the strong ones.

The year 1996 also saw the start of Congressional interest in cryptography export. The absurdity of US export controls and the danger that they would have a devastating impact on the growing electronic economy led various members of Congress to introduce bills that would have diminished executive discretion in controlling cryptographic exports. None of the bills – which in their later forms were called SAFE for Security and Freedom through Encryption – was close to having enough votes to override a promised presidential veto. Nonetheless, Congressional support for the liberalization of cryptographic export policy was to grow over the next few years, a policy in keeping with previous Congressional decisions. A decade earlier, contrary to the desires of the Reagan administration, the Computer Security Act placed civilian computer security researchs and standards under the control of the National Institute of Standards and Technology, rather than NSA. (Diffie98, pp. 68–69)

America’s International Strategy

The end of the Cold War, realigned the world and made the “east versus west” structure of COCOM inappropriate. The organization, which had existed since 1949, was replaced by a new coalition, the Wassenaar Arrangement, that included former enemies from the Soviet Union and the Warsaw Pact. The expanded organization, comprising 33 nations, is less unified than its predecessor and its procedures are less formal. Although member nations agree on a common control list, each country performs its own review. In behind-the-scenes negotiations in 1998 the Clinton administration scored a coup: Wassenaar agreed that “mass-market” cryptography using a key length not exceeding

11 The term “key escrow” had acquired a bad name.

64 bits would not be controlled.¹² The implication was that anything else would be but the Wassenaar Arrangement is subject to “national discretion,” and various nations in the agreement had not previously restricted the export of cryptography. Would they now? The Clinton administration believed so. It looked as if export restrictions would stay. Then evidence surfaced suggesting that the U.S. might be using Cold–War intelligence agreements for commercial spying.

A U.S. signals intelligence network called *ECHELON* that had been in existence for at least twenty years came embarrassingly to light. The Echelon system is a product of the UK–USA agreement, an intelligence association of the English speaking nations dominated by the United States. According to a report prepared for the European Parliament (Campbell) Echelon targets major commercial communication channels, particularly satellite systems. Many in Europe drew the inference that the purpose of the system was commercial espionage, and indeed, former CIA Director James Woolsey acknowledged that was at least a partial purpose of the system. Commercial communications play a large and growing role in government communications (both military and nonmilitary) and are thus a “legitimate” target of traditional national intelligence collection. It is the position of the U.S. that it does not provide covert intelligence information to U.S. companies.¹³ The potential targets of such spying could hardly be expected to regard U.S. policy as adequate protection under the circumstances. Consternation replaced cooperation in the European community. Nations whose policies had previously ranged from the no–controls stance of Denmark to the relatively strict internal controls of France, were now united on the need to protect their communications from the uninvited ear of U.S. intelligence and cryptography was key to any solution. European policies began to diverge from American ones.

The Rules Change

In 1999, a SAFE bill passed the five committees with jurisdiction and was headed to the floor of the House, when it was announced that the regulations would be revised to similar effect. The administration capitulated but avoided the loss of control that a change in the law would have produced.

On 16 September 1999, U.S. Vice President, and Presidential candidate, Albert Gore Jr.¹⁴ announced that the government would capitulate. Beginning with regulations announced for December – and actually promulgated on 14 January 2000 – keylength would no longer be a major factor in determining the exportability of cryptographic products.

In its attempt to make a viable military/civilian distinction, the new regulations take several factors into consideration:

1. They define a concept of *retail* products, seemingly intended to replace the *mass market* products defined in the Wassenaar Arrangement, but different in some important respects.

12 The 64–bit limit was for symmetric, or private–key, cryptography. This translates to approximately 650 bits for public–key cryptography. (Public key is typically used for key exchange; then the communication is encrypted via a private–key algorithm using the key just negotiated.)

13 The U.S. government says that it uses intelligence information to assist U.S. business in countering foreign corrupt practices. (Woolsey)

14 The Administration’s anti–cryptography policy was inimical to Silicon Valley, whose support was seen as crucial for the Vice President’s bid for President.

2. They distinguish sharply between commercial and government customers.
3. They make special provision for software distributed in source code.

In the view of export control, an item is retail if it is:

- Made freely available to a wide range of customers and preferably sold in large volume.
- Not customized for each individual user, and not extensively supported after sale.
- Not intended explicitly for communications infrastructure protection.

The definition is not entirely in accord with the everyday meaning of “retail” since many retail items are configured for each customer and some, such as custom tailored clothing, have no wholesale stage.

Retail items are largely free of control. They must be submitted for a “one-time review” that the government is supposed to complete within thirty days. If the would-be exporter has not heard anything within that time, it is free to ship its product. The government can demand additional information or even demand more time because the “review is not proceeding in an appropriate fashion” (USDoC2000, paragraph 4g) but the rule is some improvement over the previous versions which required the exporting organization to wait until it received an export license from the government before shipping.

Items that are not retail are regulated primarily on the basis of the customer. For many items, commercial sales are acceptable but government sales are not. The distinction between government and the private sector is not always clear and will surely be a continuing source of friction.

One interesting feature of the regulations is their application to software that may be distributed freely in source form. Anyone who publishes software electronically, particularly by posting it on the World Wide Web, is required to notify the Bureau of Export Administration no later than the “time of export.” If the software requires a license for commercial use, the BXA must be kept informed of foreign licensees and provided with non-proprietary descriptions of the products in which these licensees use the software.

The new rules go a long way toward achieving the objective enunciated earlier. They are a clever compromise between the needs of business and the needs of the intelligence community. Products employed by individual users, small groups, or small companies are fairly freely exportable. Products intended for protecting large communications infrastructures – and it is national communication systems that are the primary target of American communications intelligence – are explicitly exempted from retail status.

European Decontrol

In June 2000 the European Council of Ministers announced the end of cryptographic export controls within the European Union and its “close trading and security partners” called the *EU+10*. Aside from the EU, this group contains Australia,

Canada, the Czech Republic, Hungary, Japan, New Zealand, Norway, Poland, Switzerland, and the United States. The liberalized export regulations of January 14 will no longer provide the level playing field the U.S. Administration has sought.

On July 17, 2000, in response to the European liberalizations, the Clinton administration adopted similar ones: export licenses would no longer be required for export of cryptographic products to the fifteen EU members and the same additional countries. In addition, there would no longer be a distinction between governments and other customers in the European Union plus ten group of countries. Furthermore, although companies would have to provide one-time technical reviews to the U.S. government prior to export, they would be able export products immediately.

Why Did it Happen?

What forces drove the U.S. Government from complete intransigence to virtually complete capitulation in under a decade? Most conspicuous is the Internet, which created a demand for cryptography that could not be ignored and at the same time made it more difficult than ever to control the movement of information but more subtle forces were also at play. One of these was the *open-source* movement.

Ever since software became a big business, most software companies have distributed object code and treated the source code as a trade secret. For many years, the open-source approach to software development – freely sharing the source code with the users – was limited to hobbyists, some researchers, and a small movement of true believers. In the mid-1990s, however, some businesses found that an open-source operating system gave them more confidence and better reliability due to rapid bug fixes and the convenience of customization. Others discovered that they could make good money maintaining open-source software. The fact that sufficiently skillful and dedicated users could get free source code from the Web, compile it, configure it, install it, and maintain it did not mean that there were not other users willing to pay for the same services.

Open-source software has taken its place as a major element in the software marketplace. The consequence is a general decrease in the controllability of software. In particular, a serious threat to effectiveness of the government efforts to stop the export of software containing strong cryptography. A policy predicated on the concept of software as a finished, packaged product, one that was developed and controlled by an identifiably and accountable manufacturer foundered when confronted with programs produced by loose associations of programmers/users scattered around the world.

The problem is not merely one of enforcement. The government has always maintained that it could control the export of information but that view is hard to reconcile with the First Amendment and has never been thoroughly tested. A curious but widely accepted convention has grown up under which information of sufficiently limited circulation is not treated as having First Amendment protection. The maintenance manual for an aircraft may be a book but it is treated more like a component of the aircraft than a publication. Proprietary source code was treated in the same way.

By comparison open-source software was widely distributed – arguably published – on web sites. The Bureau of Export Administration might take the view that publishers of some programs required licenses but the legal basis of their position was doubtful and compliance was low. If a program, such as an operating system, leaves the U.S. without cryptography, foreign programmers can add cryptographic components

immeasurably more easily than they could with a proprietary source operating system. U.S. export controls have little influence on this process.

To make that matter more arcane, the government has stopped short of claiming that source code published on paper lacks First Amendment protection, maintaining that only source code in electronic form is subject to export control.

In 1996, Daniel Bernstein, a graduate student at the University of California in Berkeley decided that rather than ignore the law, as most researchers had, he would assert a free-speech right to publish the code of a new cryptographic algorithm electronically. Bernstein did not apply for an export license, maintaining that export control was a constitutionally impermissible infringement of his First Amendment rights. Instead, he sought injunctive relief from the federal courts. Bernstein won in both the district court (Bernstein96) and the Appeals Court for the Ninth Circuit. (Bernstein99) Unfortunately for the free-speech viewpoint the opinion of the a appeals court was withdrawn in preparation for an *en banc* review by a larger panel of Ninth-Circuit judges, an *en banc* review that never took place. The appearance of new regulations provided the government with an opportunity to ask the court to declare the case moot. To the government's delight, the court obliged, indefinitely postponing what the government perceived as the danger that the Supreme Court would strike down export controls on cryptographic source code as an illegal prior restraint of speech.

A final adverse influence on export control came from the government's role as a major software customer and the military desire to stretch its budget by using more *commercial off-the-shelf* software and hardware. If export regulations discouraged the computer industry from producing products that met the government's security use, the government would have to continue the expensive practice of producing custom products for its own needs. This was uneconomical to the point of infeasible; the only way to induce the manufacturers to include sufficiently-strong encryption in domestic products was to loosen export controls.

Conclusion

For fifty years the United States used export controls to prevent the widespread deployment of cryptography. This policy succeeded for forty of those years but changes in computing and communications in the last decade of the 20th century increased the the private-sector need for security and reduced the policy it to a Cold War relic. Its demise opens the way for securing the civilian communications infrastructure on which all of society will depend in the 21st century.

Recommendations

Although the new export regulations in the area of cryptography are a substantial improvement on earlier ones they still leave much to be desired.

- The regulations remain complex. The amendments, exclusive of surrounding procedural and explanatory material, amount to some dozen pages and the material they amend is several hundred.
- Although the burden of timeliness has on its face shifted from the exporter to the government, the conditions that permit the government to require more time are vague and appear to admit of discriminatory application. The use of these

extensions should be precisely spelled out.

- The definition of retail is at some variance with the ordinary English use of that term. The regulations should perhaps return to the Wassenaar Arrangement's concept of "mass market."
- The notification requirements for open-source programs, although considerably less onerous than the earlier licensing requirements may still constitute an unconstitutional prior restraint on publication. Considering that they, like all of cryptographic export control, serve the interests of the U.S. signals intelligence organizations, and that those organizations presumably watch the Web anyway, the notification requirements seem to serve little purpose.
- Although understandable from a U.S.-intelligence perspective, the restriction on infrastructure protection products may not be compatible with the U.S. desire to protect the critical infrastructure of the industrialized world from terrorist attack. This issue is fundamentally the same as those faced by the National Research Council CRISIS panel. We remind the readers of their conclusion, "On balance, the advantages of more widespread use of cryptography outweigh the disadvantages." (Dam96, p. 6). We believe the same holds true for infrastructure protection. On balance, the advantages of more widespread use of cryptography for infrastructure protection products outweigh the disadvantages.

The shortcomings of export law in the cryptographic area are typical of the shortcomings of our export laws in general. Cryptography may therefore point the way toward a fairer export-control regime that balances the broad spectrum of United States interests rather than focusing on military security, which is not currently a major vulnerability. Such a regime, recognizing the importance of international commerce in the post-Cold War world would shift the much of the burden from exporters to the government. Foreign availability tests would be more broadly applied; exporters would be entitled to timely responses; a broader range of export decisions would be appealable to the federal courts; and the effectiveness of export policy would be subject to periodic review.

Bibliography

AES *Advanced Encryption Standard*: <http://csrc.nist.gov/encryption/aes/>

Bernstein96 *Daniel Bernstein v. U.S. Department of State*, 922 F. Supp. 1426, 1428–30 (N.D. Cal. 1996)

Bernstein99 *Bernstein v. U.S. Department of State* 176 F. 3d 1132, 1141, rehearing en banc granted, opinion withdrawn, 192 F. 3d 1308 (9th Cir. 1999)

Campbell *Duncan Campbell, "Interception 2000: Development of Surveillance Technology and Risk of Abuse of Economic Information," Report to the Director General for Research of the European Parliament, Luxembourg, April 1999.*

Dam *Kenneth Dam and Herbert Lin, “Cryptography’s Role in Securing the Information Society,” National Academy Press, 1996.*

Diffie *Whitfield Diffie and Susan Landau, “Privacy on the Line: the Politics of Wiretapping and Encryption,” MIT Press, 1998.*

Hager *Nicky Hager, Secret Power, Craig Potton Publishing, New Zealand, 1996.*

Kahn *David Kahn, “The Codebreakers,” Scribners, 1996.*

Landau 1983 *Susan Landau, “Primes, Codes and the National Security Agency,” Notices of the American Mathematical Society, [Special Article Series], Vol. 30, No. 1 (1983), pp. 7–10.*

USDoC 1977 *United States Department of Commerce, National Bureau of Standards (1977), “Data Encryption Standard,” Federal Information Processing Standard Publication 46.*

USDoC 1994 *United States Department of Commerce, National Institute of Standards and Technology (1994), “Approval of Federal Information Processing Standards Publication 185, Escrowed Encryption Standard,” Federal Register, Vol. 59, No. 27, February 9, 1994.*

USDoC 2000 *Department of Commerce, Bureau of Export Administration: 15 CFR Parts 734, 740, 742, 770, 772, and 774, Docket No. RIN: 0694–AC11, Revisions to Encryption Items. Effective January 14, 2000.*

USHR 1999 *U.S. House of Representatives, Select Committee on U.S. National Security, Final Report of the Select Committee on U.S. National Security and Military/Commercial Concerns with the Peoples Republic of China, 1999.*

Woolsey *James R. Woolsey, “Why We Spy on Our Allies” The Wall Street Journal, March 17, 2000.*

High-performance, Space-efficient, Automated Object Locking

Laurent Daynes and Grzegorz Czajkowski

(17th IEEE International Conference on Data Engineering, Heidelberg, Germany)

Introduction by Laurent Daynes and Grzegorz Czajkowski

The technology described in the paper published in the 17th IEEE International Conference on Data Engineering in April 2001 was developed in the context of the Forest project. The two authors were then working on supplementing a virtual machine that implemented Orthogonal Persistence for the Java™ platform (OPJ) with the ability to safely execute simultaneously multiple applications. The approach chosen was dictated by the requirement to enable safe sharing and direct access to objects by multiple applications. Guaranteeing isolation between applications while allowing them to safely share data requires mechanisms akin to those used in database systems. To this end, the OPJ virtual machine was augmented with very light-weight transactional mechanisms that automatically enforces "serializability", thus giving each application the illusion of running stand-alone while actually sharing, potentially concurrently, data with other applications. The principles for this approach were actually elaborated with the very first design of OPJ [1]. The main obstacle to realizing this transaction-based approach to application isolation was the absence of compelling technology to support automated concurrency with performance acceptable for the runtime of a modern programming language [2]. The paper describes in detail a patented solution that address this problem.

Although the work on efficient automated object locking has enjoyed academic success (the paper received the Best Paper Award), the development of a first prototype of transaction-based protection in the context of the Java platform has exposed many problems that made clear that much more research was needed before the approach reached maturity. The pressing need for scalable execution of many applications has resulted in departing from safe sharing and focusing on the simpler case of strict isolation between applications. Under the lead of G. Czajkowski, the two authors are now pursuing this direction to design scalable multi-tasking virtual machines (MVM) for the Java programming language.

The work on MVM grew out of an initial prototype based on bytecode editing that was developed as a proof of concept to lay out the basic principle of a new approach to scalable application isolation [3]. Since then, a more ambitious prototype based on the Java HotSpot™ Virtual Machine has been developed [4]. At the same time, solutions to isolate a JVM™ from unwanted interactions with, and errors from user-defined native methods have also been developed [5]. Both efforts have played a key role in the proposal of the JSR 121, <http://jcp.org/jsr/detail/121.jsp>.

"High-Performance, Space-efficient, Automated Object Locking." © 2001 Sun Microsystems, Inc. and IEEE. Reprinted, with permission, from Seventeenth IEEE International Conference on Data Engineering, Heidelberg, Germany, 2–6 April 2001.

PUBLICATIONS:

1. M. Atkinson, M. Jordan, L. Daynes and S. Spence, "Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system", Proceedings of the 7th International Workshop on Persistent Objects Systems, Cape May, NJ, May 1996.
2. L. Daynes, "Implementation of Automated Fine-Granularity Locking in a Persistent Programming Language", Software – Practice and Experience, 30:1–37, 2000.
3. G. Czajkowski, "Application Isolation in the Java Virtual Machine", Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Minneapolis, MN, October 2000.
4. G. Czajkowski, L. Daynes, "Multi-tasking without compromise: a Virtual Machine Approach", Proceedings of Object-Oriented Programming, Systems, Languages and Applications, October 2001, Tampa, FL.
5. G. Czajkowski, L. Daynes, M. Wolczko, "Automated and Portable Native Code Isolation", Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering, Hong-Kong, November 2001.

High-Performance, Space-efficient, Automated Object Locking

Laurent Daynès Grzegorz Czajkowski
Sun Microsystems Laboratories
901 San Antonio Road, Palo Alto CA 94303
{firstname.lastname}@eng.sun.com

Abstract

The paper studies the impact of several lock manager designs on the overhead imposed to a persistent programming language by automated object locking. Our study reveals that a lock management method based on lock state sharing outperforms more traditional lock management designs. Lock state sharing is a novel lock management method that represents all lock data structures with equal values with a single shared data structure. Sharing the value of locks has numerous benefits: (i) it makes the space consumed by the lock manager small and independent of the number of locks acquired by transactions, (ii) it eliminates the need for expensive book-keeping of locks by transactions, and (iii) it enables the use of memoization techniques for whole locking operations. These advantages add up to make the release of locks practically free, and the processing of over 99% of lock requests between 8 to 14 RISC instructions.

1. Introduction

The PJama Virtual Machine (PJVM) [1] is an extension of the JavaTM Virtual Machine (JVM) [2] with orthogonal persistence [3]. From the outset, the PJama project has aimed at a flexible and integrated transaction (FIT) environment, where every computation runs in the context of a transaction, and all objects, irrespective of their type and lifetime are under transactional control [4]. The FIT combination of type safety, systematic confinement of computation in transactions, and automatic enforcement of transactional properties, offers a robust environment that makes it unnecessary to use separate virtual machines (and therefore, separate operating system processes) to defend persistent applications against each other's errors.

One of the main obstacles to applying the FIT approach to PJama is the absence of sound technology to efficiently support automated concurrency control. We favored locking over other concurrency control techniques for two main reasons: its impact on PJama's already sophisticated mem-

ory management is minimal; and, it can be easily extended to support advanced concurrency control semantics [4], which will be supported in future versions of PJama.

PJama differs from most multi-user persistent programming languages (PPLs) and object-oriented databases (ODBMS) in that all transactions execute within the same address space, and can directly access both transient and memory-resident persistent objects. This architecture requires sophisticated lock management, such as that used in centralized relational database systems, to mediate the direct accesses to objects by transactions.

1.1. Problem Characterization

Programming systems that tightly couple a transaction processing engine with a high-level programming language usually include in their runtime a component that automates the requesting of locks on behalf of executing programs. Automated locking both simplifies the programmer's work and avoids depending on the programmer to always formulate *well-formed* transactions, that is, transactions that execute an operation on an object only when they own the lock of that object in the lock mode corresponding to that operation. In a PPL, automated locking is accomplished by transparently augmenting programs with small sequences of instruction, called a *lock barriers*, that issue lock request to the lock manager (LM) when it is appropriate to do so.

Ideally, a transaction needs to request the lock of an object it accesses only once, before its first access to that object. Identifying ahead of time the first access to an object by an arbitrary program is, in general, impossible. A pragmatic solution is to precede every object access with a lock barrier, and rely on compiler analysis to identify and remove as many *redundant* lock barriers as possible.

Implementing automated locking at the granularity of individual objects in PJama is challenging. First, the size of objects is small, typically between 16 to 42 bytes [5], while transactions can be very large. For instance, the OO7 benchmark [6], which is believed to reproduce the behavior of a typical PPL application, defines elementary operations

that access 60,000 objects for small-size databases. This requires space-efficient lock management that scales well with the number of locks. Second, the characteristics of the JavaTM programming language complicate the elimination of redundant lock barriers. Dynamic class loading prevents the use of global optimization, the costs of which cannot be afforded at runtime anyway. The absence of effective methods for removing unnecessary lock barriers results in frequent unnecessary lock requests (99% as indicated by our measurements). Lastly, persistent objects typically reside for a substantial amount of time in main memory so that the use of *swizzling* techniques [7] to translate them into a main-memory format suitable for direct manipulation pays off. Thus, accessing a persistent object often costs as little as a main-memory access. Hence, each instruction added by a lock barrier to an object access has dramatic performance consequences.

1.2. Related Work

We aren't aware of any PPLs or ODBMSs that use an architecture similar to that of PJama, which combines direct access by concurrent transactions to a shared heap of objects with automated fine-granularity locking. PPLs that offer transactions (e.g., [8, 9]) leave the responsibility for concurrency control to the programmers, who have to manually set locks. Most ODBMSs have adopted a client-server architecture, where clients supply each transaction with a private buffer of objects [10, 11, 12, 13], or a private buffer of pages [14]. Some ODBMSs (e.g., [15]) combines direct access by multiple transactions to a shared page buffer pool with page locking in order to use virtual memory protection to automate the acquisition of locks. This approach eliminates the need for efficient software-only read and write lock barriers. However, it constrains the granularity of locks to be a multiple of the page size, it requires a separate address space per transaction, and it prevents objects from moving from their original virtual page.

Enabling direct access by concurrent transactions to a shared heap requires a sophisticated lock manager, such as those used in centralized relational database systems. The implementation techniques of lock managers (LMs) have not evolved significantly during the last twenty years [16], and all database systems seem to use some minor variation of the System R lock manager [17]. The most complete and recent detailed description of a LM can be found in [18]. In short, each lock is represented by a data structure called a lock control block (LCB). An LCB is the head of a list of lock request control blocks (LRB). Each LRB represents a granted or pending request of one transaction for the lock. Transactions keep track of their locks by chaining all their LRBs together. LCBs are maintained in a hash table keyed by resource identifier. Processing a lock request consists

of looking up the lock hash table for an existing LCB, and allocating one if none was found. Then, the LRB chain is searched to find the LRB of the requester. If none is found, an LRB is created for the requester to represent either a pending or a granted request, depending on what the conflict detection diagnosed.

Rather than revisiting this design, database implementors have focused on reducing the number of calls to the LM and the overall number of locks used by taking advantage of the physical organization of data into container hierarchies, the limited number of access paths to the data, the semantics of operations on data, and the query-oriented nature of accesses to the data (range-locking techniques [19, 20] illustrate this well).

The introduction of main-memory database systems has changed the trade-off between tuple access and lock management costs, making the database community look closer at the performance of lock operations. The main novelty is the replacement of the hash table that maps locks to the resource they protect, by direct pointers from resources (e.g., tuple) to locks [16], since locked resources are always memory resident. To reduce the high cost both in terms of space and processing overhead of record locking, lock de-escalation techniques [16] and cheap-first locks [21] were suggested. The former relies on the premises of a query processing programming interface, built-in indexes, and a small fixed number of paths to data, and is therefore inappropriate for PPLs. We used a variant of the later in one of the implementations studied in this paper.

1.3. The Focus of the Paper

Our previous work revisited the underlying principle of lock management to eliminate the components of a LM that do not scale with the number of locking units. The result was a novel approach which we called *lock state sharing* [22]. Lock state sharing was shown to dramatically reduce the space overhead of object locking.

This paper focuses on the processing overhead of automated locking. Our previous work was rather inconclusive with respect to this dimension, mostly because of the poor performance of the version of the PJVM used then – an interpreter-only JVM based on the JDKTM version 1.1.7. The PJVM we used this time is based on a state-of-the-art, handle-less, JVM, that includes better garbage collection support, faster synchronization, native thread implementation, and a well-tuned just-in-time (JIT) compiler. It runs 10 times faster than the previous one [24], and adds about 20% of overhead to programs when compared with the original JVM it is derived from¹. This dramatic improvement to the

¹The execution times for OO7's traversals executed on a transient database is 21% slower than with a non-persistent JVM. The SPECjvm98 *db* and *javac* programs are slowed down by 14% and 18% respectively.

performance of the PJVM makes it much more sensitive to the choice of a particular combination of lock barrier and LM to implement automated locking.

The paper is organized as follows. Sections 2 to 5 briefly describe the LM designs studied in the paper. Section 6 presents how the PJVM was modified to support automated locking. Section 7 describes the experiments we conducted to evaluate the impact of the various lock management techniques on PJama’s performance, and Section 8 analyzes their results. Section 9 reports our conclusions.

2. “Traditional” Lock Management

The LM used as the starting point of this study derives from our previous work on PPL-friendly LMs [25], because it was shown to outperform more traditional implementations (e.g., [18]). However, this LM still shares many of the general principles of all the LMs we are aware of, so we will refer to it as *TRAD*, for “*traditional*” implementation. Since all the other LMs studied in the paper result from modifications to *TRAD*, we review its most salient features below.

2.1. Fast access to locks

Fast access to the lock of an object is obtained by storing in its main-memory representation a direct pointer to its lock. In contrast to memory-resident database systems [16], we also face the problem of maintaining the object-to-lock association across possible movements of the object between main memory and disk.

The PJVM, like most persistent object system implementations, maintains in main-memory an associative table, called the ROT (Resident Object Table), that maps persistent object identifiers to addresses in main memory [10, 11, 24]. When the memory manager evicts an object, it first determines whether the object is locked, and if so, replace its address in main-memory with that of its lock. The lock pointer thus remembered is re-installed in the object it protects the next time this one is faulted-in. ROT entries of unlocked non-resident objects are recycled.

2.2. Compact static representation of locks

Figure 1 shows the overall design of the *TRAD* LM. A fixed-size bitmap representation of locks was favored over the classic linked list of lock LRBs. Each active transaction is given a *locking context*, which the LM identifies by a unique bit number. The bit number also indexes a table of locking contexts. All bitmaps use the same mapping from locking contexts to bit numbers, i.e., the i^{th} bit always identifies the same locking context, and therefore, the same transaction. A bit set to 1 in a bitmap indicates that the transaction

corresponding to this bit location belongs to the set of transactions represented by that bitmap.

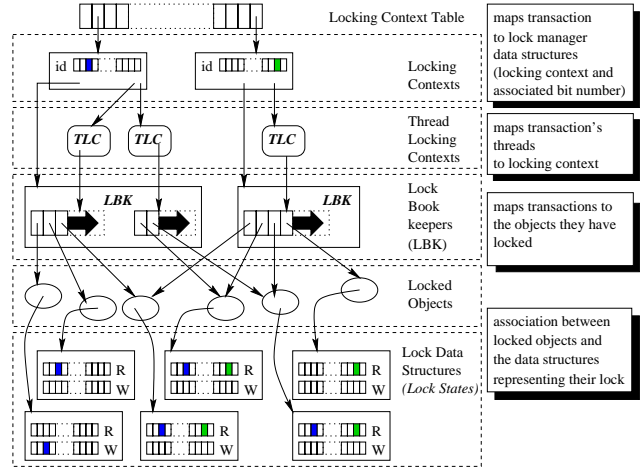


Figure 1. The *TRAD* lock manager design.

The locking context of a transaction comprises its bit number, a list of *thread locking contexts*, and a lock book-keeper. Thread locking contexts are a thread-private data structure used by the LM to service a thread’s lock requests with minimal synchronization between threads of the same transaction. Similarly, lock book-keepers maintain a pool of thread-private growable stacks of references to locked objects.

Locks are implemented as *lock states*. A lock state consists of a *lock type*, book-keeping data, and an array of bitmaps. Each bitmap represents an *owner set* for a given lock mode. A transaction T has been granted a lock L in a mode M if T belongs to the owner sets of L associated with M .

This design has several advantages: (i) lock data structures being of fixed size, at most one transaction pays a memory management operation per lock; (ii) the space overhead is quickly amortized as the working sets of concurrent transactions overlap, whereas, with an LRB list, it increases with the number of transactions [25]; and (iii), operations such as testing and updating lock ownerships translate into fast logical operations on bitmaps. The size of bitmaps is chosen to match the maximum number of concurrent transactions supported.

2.3. Cheap-First Lock

In order to reduce both the space and processing overhead of locking, the allocation of a lock structure is postponed until a second transaction requests a lock. This idea, referred to as “cheap-first lock”, is credited to IBM’s IMS Fast Path [16, 21]. Cheap-first locking marks objects locked by

a single transaction with a tag that identifies both the lock's owner and mode. A lock data structure is allocated only when a second transaction requests a lock on that object.

The *TRAD* LM implements cheap-first locks as follows. All unlocked memory-resident objects point to an *immutable* lock data structure set with the unlocked value, and shared among all unlocked objects. The locking context of each transaction is augmented with two immutable lock data structures that represent the value of a lock owned by that transaction only in, respectively, read and write mode. These lock values are called, respectively, the SRO (single read owner) and SWO (single write owner) of the transaction. The pointers to the SRO and SWO are the tags used to mark objects locked by that transaction only. Thus, the lock pointer of all objects locked by a single transaction in a given mode points to the same lock data structure.

When a transaction requests a lock against an unlocked object, it atomically exchanges the lock pointer to the unlocked lock state with the lock pointer to either its SRO or SWO, depending on the locking mode requested, and adds the reference of the object to its lock book-keeper. When a transaction requests a lock on an object already associated with either an SRO or an SWO lock value for another transaction, it allocates a new lock data structure, sets its new value, and atomically exchanges the current lock pointer with that of the allocated lock structure before recording the locked object. Inversely, when a lock-release operation results in a single-owner lock value, the lock pointer of the locked object can be replaced with the corresponding SRO or SWO pointer, so that the space of the previous lock data structure can be reclaimed.

2.4. Fast-paths

Automated locking can generate an overwhelming number of redundant lock requests, i.e., requests for locks already granted. Because rigorous 2PL is enforced, once granted, a lock is not released until a transaction that owns the lock completes. To reduce the overhead of redundant lock requests, a transaction can take an inconsistent view of the state of a lock and test if it owns it. Such inconsistent tests can be small enough to be inlined in the LM's caller. In the following, we call such small sequences of inlined code a *fast-path*, for it can bypass a call to the LM.

The simplest fast-path is a test of the lock pointer of the requested object. If it is equal to one of the cheap-first lock values of the requester, the LM is bypassed. This fast-path takes only 5 instructions, but can fail often because it covers only a few cases of redundant lock requests.

A more general fast-path consists of testing the membership of the requester in the owner set corresponding to the requested lock mode. This fast-path takes between 8 to 10 instructions, depending on whether the size of bitmaps im-

plementing owner sets is larger than the size of a register.

3. Lock State Sharing

The *TRAD* implementation suffers two major drawbacks that are the consequence of the underlying principles that are used in all of the LMs of which we are aware: (i) *there is one lock data structure per unit of locking*, and (ii), *the lock manager keeps track of the locks of each transaction in order to automatically release them when a transaction terminates*. Delaying lock data structure allocations using cheap-first locks only helps when transactions rarely overlap their working sets.

The *lock state sharing (LSS)* technique introduced in [22] to correct the first problem relies on two observations: the number of running transactions is very small when compared to the number of objects they access; and the number of combinations of sharing between transactions is likely to be small. In other words, one can expect many lock data structures to have identical values.

Therefore, it is advantageous to make two objects with locks of *equal value* point to the same *shared immutable* lock state, and have lock operations change the *association* between an object and the shared lock state representing the value of its lock, rather than updating a private lock data structure. To guarantee that only one shared lock state is used to represent a given lock value, the LM maintains an associative table of immutable shared lock states (*TILS*) keyed on lock values. Note that the TILS does not hold all possible lock values: it is initially empty, and shared lock states are added to it as needed, when a TILS lookup fails. Garbage collection techniques determine unused shared lock states and remove them from the TILS.

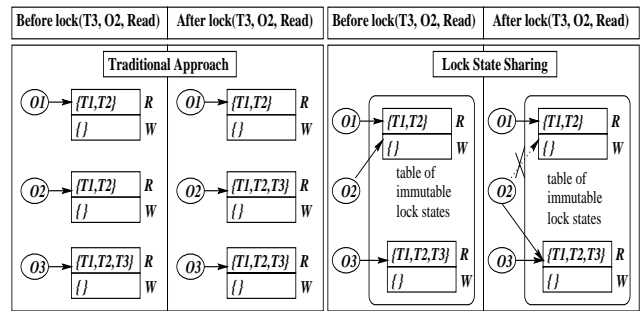


Figure 2. Locking with *TRAD* vs *LSS*.

Figure 2 compares *LSS* (right) with the traditional approach (left) using a simple example. It shows a scenario where two objects, O_1 and O_2 , have been locked in read mode by two transactions T_1 and T_2 , and a third object O_3 has been locked in read mode by transactions T_1 , T_2 and T_3 . The state of each lock and the association between objects and

their lock is shown before and after the acquisition of a read lock on O_2 by T_3 .

In the traditional approach, each object is associated with a private lock data structure. T_3 's request for O_2 's lock is processed by updating O_2 's private lock data structures to reflect the lock's new value (i.e., lock owned in read mode by T_1 , T_2 , and T_3). In *LSS*, O_1 and O_2 share the same lock state which holds the value that the two private data structures representing their respective locks would have. Instead of updating the lock state data structure associated with O_2 to process T_3 's request, the LM searches the TILS for a shared immutable lock state with O_2 's new lock value, and atomically exchanges O_2 's current lock pointer with the pointer returned by the TILS. It is crucial to understand that in both approaches, every object is a unit of locking: the sharing of lock states of equal value must not be confused with protecting several objects using the same lock.

LSS is obtained with minimum changes to the *TRAD* implementation: lock data structures remain unchanged, except that they now represent shared immutable lock states instead of mutable private locks. A hash table of lock data structures keyed on their value implements the TILS. Locking operations work by first building a temporary lock-state value equal to the new value the object's lock must have, and then by using it to probe the TILS for an equivalent immutable shared lock state. The object's current lock pointer is then atomically exchanged with the pointer to the shared lock state returned by the TILS.

To summarize, *LSS* allows transactions to arbitrarily overlap their working sets without dramatic space consumption due to locking. *LSS* is much less sensitive to the maximum size of owner sets, and can afford the use of owner sets containing several hundred elements without noticeable space overhead. Lastly, transitions from one lock state value to another one consist of a single pointer exchange. On most stock hardware this can be accomplished with a single atomic compare-and-swap instruction, such as the SPARCTM platform V9 `cas` or Intel486's `cmpxchg`. This promotes the use of non-blocking synchronizations to implement locking operations, which avoids expensive latching and reduces contention.

4. Elimination of Lock Book-Keeping

All the LMs presented so far keep track of the objects locked by each transaction in order to automatically unlock them when the transaction completes. This stems from the original database systems' assumptions that (i) the total number of lock data structures is much larger than the number of locks acquired by one transaction, and (ii), transactions acquire a relatively small number of locks². Under

²Database LMs have recourse to adjustable locking granularity to keep the number of locks below a few thousand [18].

these conditions, keeping track of a transaction's locks is efficient.

LSS invalidates the above assumptions. Indeed, the total number of shared immutable lock states maintained by the LMs is expected to be several orders of magnitude smaller than the number of objects locked by a single transaction. Hence, to release the locks of a transaction, it is beneficial to simply scan all the shared lock states to find those that represent locks of that transaction, and update their values to reflect the effect of a lock release. This relaxing of the immutability of shared lock states raises two issues. First, it might generate duplicates of existing shared lock states. Such duplicates do not harm the correctness of the locking logic, but increase space consumption. Second, synchronizing exchanges of pointers to shared lock states with updates to shared lock states can increase substantially the costs of lock acquisitions.

The *NLSS* LM is derived from the *LSS* LM by eliminating lock book-keeping data structures and related code. When a transaction T releases all of its locks, the *NLSS* LM first scans the TILS to find all the shared lock states that represent values of locks held by T . For each such lock state, T is removed from all the owner sets. This might turn the shared-lock state into a duplicate of another existing shared lock state of the TILS. If that is the case, the modified lock state is removed from the TILS and added to a *list of duplicates*. Otherwise, it is re-hashed into the TILS.

Once the TILS is processed, the LM must take care of duplicates that might represent values of T 's locks. Such duplicates may have been created by transactions that were running concurrently with T but completed before T , and whose working sets overlapped with that of T . The LM scans the list of duplicates and updates those that hold lock values representing T 's locks. Modifying a duplicate of a TILS's shared lock state always results in a duplicate of another TILS's shared lock state. Therefore, a duplicate can never re-enter the TILS.

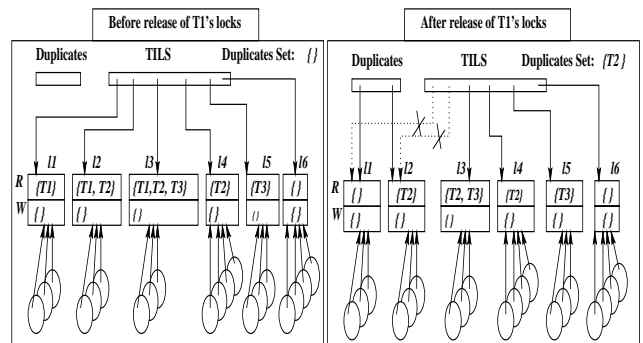


Figure 3. Releasing of locks with *NLSS*.

Figure 3 illustrates how *NLSS* works. It depicts the data

structure of the LM before and after the release of the locks of transaction T_1 . Before T_1 's end, three shared lock states represent values of lock owned by T_1 (l_1 , l_2 and l_3). When T_1 completes, it is removed from the owner sets of these lock states. However, the new updated values of two of these lock states (namely, l_1 and l_2) are already represented in the TILS (by, respectively, l_6 and l_4), so they are removed from the TILS and added to the list of duplicates. When T_2 terminates and releases its locks, it will generate two additional duplicates of the "unlocked" lock state l_6 (l_2 and l_4) and a duplicate of l_5 (l_3). Note that *no associations* between locked objects (circles in Figure 3) and shared lock states are modified when a transaction releases its locks.

The elimination of lock book-keeping from *LSS* seems attractive because it saves both computational and space costs. Furthermore, the costs of releasing all of the locks of a transaction is in terms of the number of shared lock states instead of the number of objects locked by that transaction. On the other hand, the absence of lock book-keepers generates duplicates that increase space consumption, and requires potentially costly sophisticated synchronizations.

5. Elimination of TILS Lookups

The most expensive part of a locking operation for the *NLSS* lock manager is the probing of the TILS to obtain the shared lock state that represents the new value that the object's lock must have. If that shared lock state were already known, the locking operation would just consist of an atomic exchange of the object's current lock pointer with that lock state. Knowing the lock state that represents the result of a lock request in advance can be achieved by applying *memoization* - a technique used in functional programming languages [23].

More specifically, let T be a transaction, op a locking operation, l_i and l_f two shared lock states such that l_f holds the lock state value equal to the result of T executing op with the lock value held by l_i (i.e., $l_f = op(T, l_i)$). Instead of computing the value of $op(T, l_i)$ upon every call to op and probing the TILS with that value to obtain the equivalent shared lock state l_f , the LM maintains l_f in a cache keyed by op , l_i and T . If the parameters passed to a call to op match l_i and T , then l_i can be immediately exchanged with l_f . Otherwise, $op(T, l_i)$ is computed, the TILS probed, and the cache updated before atomically changing the shared lock state pointers.

Adding memoization to *NLSS* is straightforward: each thread locking context is augmented with a memoization cache. Each cache line consists of two shared lock state pointers. The cache has a *single line* per locking operation (e.g., read lock acquisition, write lock acquisition, lock release), that is, only one result per operation is memoized. Probing the cache and subsequently exchanging the current

lock pointer with the probe's result upon a cache hit takes 6 instructions (one arithmetic computation of the lock pointer address, two loads, one atomic compare and swap, a comparison and a branch). This variant of *NLSS* is called *m-NLSS*.

6. Automated Object Locking

Both the interpreter and the JIT compiler of the PJVM have been modified to precede every object access with a lock barrier. Lock barriers typically consist of a call to the LM wrapped in an inlined fastpath (if one is defined by the particular lock manager used). Some of the optimizations already performed by the PJVM JIT compiler, such as common sub-expression elimination, naturally eliminates some redundant lock barriers, but only in marginal proportion, and with almost no visible effect on performance.

One negative effect of this strategy is to pair most accesses to *new objects*, i.e., objects allocated by transactions in progress, with a lock barrier, although locking new objects is unnecessary to enforce strict isolation. New objects fall into two categories: objects reachable only from the stack of the thread that created them, and objects whose reference has been stored by their creator into objects reachable from other transactions. Objects of the first kind are unreachable from other transactions. All paths to objects of the second category are protected by an exclusive lock of their creator, since in order to store a reference in an object O , the automated locking system first obtains a write lock for O . In both cases, access to new objects by transactions different from their creator is already prohibited, therefore locking is unnecessary for new objects.

One simple strategy to reduce the overhead of lock barriers against new objects is to make them look as if they are already locked by their creator. Under this approach, new objects are associated with the SWO of their creator at allocation-time, without calling the LM. Upon transaction completion, the shared SWO lock value is atomically turned into a shared unlocked lock value. All of the created objects look like they are unlocked and become subject to normal automated locking rules. This strategy turns all requests to new object's locks into redundant lock requests, and guarantees that new objects escape to lock book-keeping.

7. Performance Experiments

A total of 15 different PJVM versions were evaluated. The versions differ from one another by two parameters: the LM and the lock barrier used. Table 1 lists the various versions and their characteristics. The LMs share most of their code, except for lock operations whose implementation is specific to each LM. The number of concurrent transactions

was limited to 64 for all LMs so that transaction set operations could be implemented with single 64-bit register instructions. This case favors *TRAD* because both the processing of bitmap operations and the space consumed for a lock state is minimal.

lock barrier			
direct LM call	single owner	single write owner	owner set membership
<i>TRAD</i>	<i>so-TRAD</i>	<i>swo-TRAD</i>	<i>osm-TRAD</i>
<i>LSS</i>	<i>so-LSS</i>	<i>swo-LSS</i>	<i>osm-LSS</i>
<i>NLSS</i>	<i>so-NLSS</i>	<i>swo-NLSS</i>	<i>osm-NLSS</i>
<i>m-NLSS</i>		<i>swo-m-NLSS</i>	<i>osm-m-NLSS</i>

Table 1. Lock Barrier Implementations.

The performance experiments were done using SPECjvm98 [26], and OO7 [6]. SPECjvm98 is a standard suite of benchmark programs destined to evaluate the performance of JVMs. Due to space limitations, we only report measurements for db (multiple database functions on a memory-resident database), and javac (the Java compiler from the JDKTM version 1.0.2).

The OO7 benchmark synthesizes applications managing complex data structures such as CASE or CAD/CAM systems. It defines a database organized in modules. Each module has a manual and a seven level deep hierarchy of assembly objects. Assemblies recursively reference three other assemblies. Leaf assemblies have three bi-directional links toward three composite parts, each of them consisting of a graph of atomic parts inter-connected to three other atomic parts of the same graph. Inter-connections are themselves objects. The benchmark defines three database configurations which varies the number of atomic parts per composite part, and the size of the database. Our experiments with OO7 focus on read-only traversal operations, namely T1 and T6. as we study the overhead of locking operations, and not throughput. Using read-only operations allows for experimenting with various degrees of working set overlap between transactions without interference due to conflicts. Traversal operations navigate through the assembly hierarchy and visit each composite part of each base assembly. For each visited part, T1 performs a depth-first traversal on its graph of atomic parts, whereas T6 visits the root of the graph only.

Measurements were done on a Sun EnterpriseTM 420, with 4 UltraSPARCTM-IIi processors clocked at 450Mhz, a system clock frequency of 113Mhz, 1 Gb of main memory, running the SolarisTM 2.7 operating environment.

8. Performance Analysis

All measurements are compared with measurements of the original PJVM, which does not support transactions. The data reported includes the count of lock requests issued for each benchmark in order to evaluate the fraction of requests that result in calls to the LM. In all our measurements, calls originating from the interpreter accounted for less than 0.05%, so we will not discuss them.

8.1. SPECjvm98

The left part of figure 4 reports the performance of two of the SPECjvm98 programs (the others behave similarly.) These programs were run unchanged. In that case, the static main method is wrapped by a *main* transaction automatically started by the transaction manager, and the whole computation takes place in that transaction. All the objects manipulated by SPECjvm98 programs are created by the main transaction only. As seen in Section 6, the solution to compensate for the lack of program analysis to identify new objects is to make them look as if they have already been locked. Consequently, all lock requests issued for these objects by the automated locking system are redundant. Performance then depends on how well the lock barrier used can filter unnecessary lock requests.

Two histograms are shown for each program: the right-most one shows the overhead relative to a PJVM without automated locking, the one next to it shows the number of lock barriers that resulted in a call to the LM, compared with the total number of lock barriers executed.

The three lock managers *TRAD*, *LSS*, *NLSS* process lock requests for a lock owned *only* by the requester in exactly the same way. Hence there is little difference between systems in which lock barriers call directly the LM (*TRAD*, *LSS*, *NLSS*). Although owner-set membership (OSM) tests take more instructions than a single-owner (SO) tests, they filter *all* redundant lock requests. SO tests often fail because the majority of lock requests are read lock request for new objects protected by write locks. Hence the heuristic “test lock pointer against the requester’s SRO lock pointer” use for read lock requests systematically fails. As a result, automated locking systems that use SO tests perform worse than their counterparts that call the LM directly, because the extra instructions of the test never pay off. As an experiment, we changed all the SO tests so that the current lock pointer is systematically tested against single write-ownership (SWO) by the requester (which covers both read and write lock ownership). This time, most lock barriers succeed and the locking overhead is about the same as with the OSM tests.

Memoization alone also fails to filter all of the calls to the LM, because of the mix of lock request which decreases the

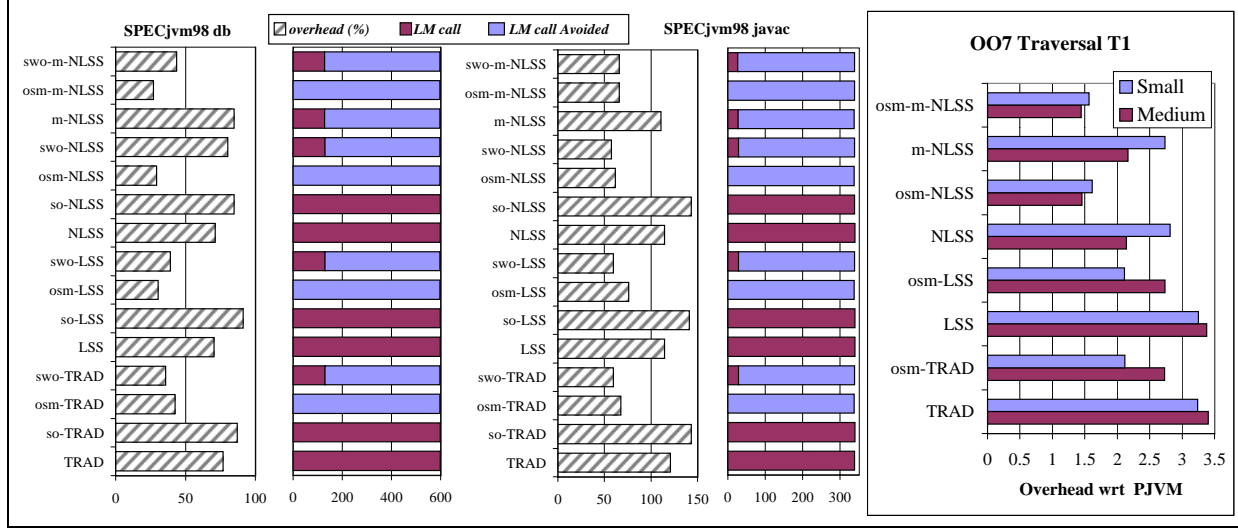


Figure 4. Overhead with respect to PJVM: left, SPECjvm98 programs db and javac, right OO7 traversal T1, small and medium database.

efficiency of the memoization cache. In this case, preceding the use of the memoization cache with an OSM or a SWO test also pays off.

Overall, for non-transactional workloads, automated locking imposes a minimum overhead of between 26% (SPECjvm98 db) and 70% (SPECjvm98 jack, a parser generator).

8.2. OO7

For the OO7 benchmark the individual traversal operations are run as transactions such that the measured transaction entirely overlaps its working set with a number of other transactions. This 100% overlap among transaction working sets defines the upper bounds for the time and space overheads of locking. To measure this, the first $n - 1$ transactions are started, invoked to run the traversal once, and suspended just before commit. The n^{th} transaction is then executed to completion and measured. Figure 5 reports the measurements of that n^{th} transaction for various degrees of overlap ($n = 0$ if there is no overlap) for two of the read-only traversals of OO7, namely T1 (on the left) and T6 (middle), executed against the **medium** size database. We report only measurements of versions that bring additional insight about the costs of automated object locking with respect to Section 8.1. Therefore, we will not discuss versions that use lock barriers based on SO tests (they consistently underperform direct call to the LM), and lock barriers based on SWO tests (they perform up to 34% worse than OSM tests).

The right part of figure 4 compare the overhead with re-

spect to the PJVM of a traversal T1 of the small and medium databases. The histograms on the rightmost part of Figure 5 show how many calls to the LM are avoided by each of the four type of lock barrier we experimented with (namely, OSM test combined with memoization, memoization alone, OSM test alone and direct call to the LM). The bars reporting “filtered” reads and writes show the amount of read and write lock requests that were successfully processed by the lock barrier without entering the LM.

In contrast to the non-transactional workloads of SPECjvm98, the choice of a lock management method now matters because of the number of lock acquisition and release operations: a traversal T1 of a medium database acquires about 625,000 locks.

In the absence of overlapping transactions, redundant lock requests are still the dominant source of locking overhead (they account for 99.2% of the lock requests issued.) Therefore, the use of OSM tests reduce the locking overhead substantially. Lock acquisition and release are fairly cheap in the absence of overlapping transactions since, for all LMs, they merely consist of an atomic exchange of pointers to immutable shared lock state (lock acquisition swaps pointers to the unlocked shared lock state to the requester’s SRO, and the other way around for lock release). In particular, neither *LSS* not its variants need to probe the TILS to acquire lock. Avoiding lock book-keeping pays off, as illustrated by the gap between *osm-TRAD* and *osm-LSS* on one hand, and *osm-NLSS* and *osm-m-NLSS* on the other hand.

When a transaction overlaps with another one, locking

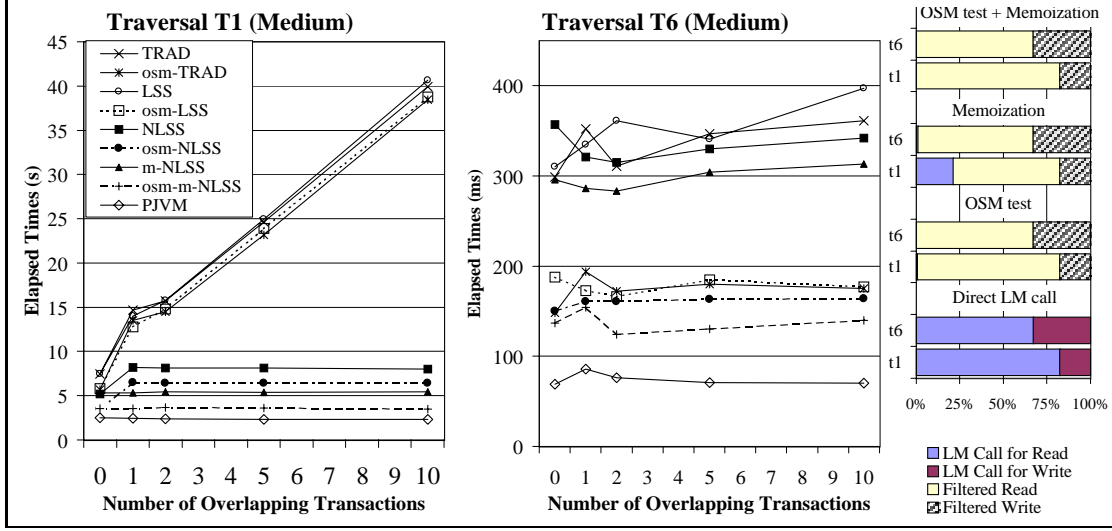


Figure 5. Measurements for OO7 Traversals (100% of overlap between transactions).

operations become more expensive. In particular, for *TRAD* and its variants, the second transaction pays the price of dynamic memory management to allocate/free private lock data structures for each traversed object, according to the cheap-first lock strategy. In contrast, lock state sharing implementations allocate only one new lock data structure to represent the new lock value. On the other hand, the costs of probing the TILS kicks in when the degree of overlapping is greater than one. The resulting degradation in performance is roughly the same, and is illustrated by the sharp increase in overhead for *LSS*, *osm-LSS*, *TRAD* and *osm-TRAD*.

When the degree of overlap is greater than one, the transaction does not pay for allocation anymore, as the second transaction has already paid for it. This reduces the overhead for *TRAD* and *osm-TRAD*. As the degree of overlapping increases, the performance of LMs that keep track of locked objects (*TRAD*, *osm-TRAD*, *LSS*, *osm-LSS*) degrades significantly. In contrast, LMs that do not use lock book-keeping (i.e., all the variants of *NLSS*) scale better.

Again, memoization alone does not perform well because of the frequent invalidation of its cache due to redundant lock requests. This is particularly striking for T1, where 20% of lock requests result in a cache miss, and therefore, a call to the LM. Preceding the memoization cache probe with an OSM-test considerably improves the memoization cache efficiency, reducing cache misses to practically nothing. This makes *osm-m-NLSS* insensitive to the degree of overlapping because most lock acquisitions are serviced without calling the LM.

To summarize, *osm-m-NLSS* outperforms all other LMs for the following reasons: (i) OSM tests filter out *all* of the redundant lock requests, which avoids unnecessary calls to

the LM, and, most importantly, improves the efficiency of memoization caches; (ii) memoization reduces the number of calls to the LM for lock acquisition by 99.5%; and, (iii) lock state sharing without lock book-keeping makes lock release independent of the number of acquired locks. These properties reduce the locking overhead to under 70% for overlapping working sets.

9. Conclusions

This paper has studied the impact of several combinations of lock management method and lock barrier implementation on the processing overhead of automated locking in persistent programming languages using the PJama Virtual Machine as a case study. Our experiments indicate that the best performance are obtained with the *NLSS* variant of lock state sharing, a novel lock management method. *NLSS* dramatically reduces the memory consumption of locking by sharing lock data structures with identical values among locked objects, and avoiding the tracking of locked resource used in other lock management method to automatically release locks at transaction completion. The best performance of automated locking are obtained by combining *NLSS* with a two-stage lock barriers, which consists of an inconsistent lock ownership test to filter out all unnecessary and redundant lock requests, followed by, if the first stage fails, a memoization to avoid calling the lock manager. The resulting lock barrier cost between 8 (first stage only) to 14 (two stages) RISC instructions, and avoids calling the lock manager for 99% of the lock request issued by automated locking. This combination results in at least 30% performance improvement over other lock management methods, and makes automated locking scaling well with both the

size of the working set of a transaction and the degree of overlapping between transaction's working sets.

There is still room for improvement, as automated object locking imposes a considerable processing overhead: between 26% to 67% for an automated locking system based on the best variant of lock state sharing. This overhead is essentially due to the overwhelming number of unnecessary lock barriers. This clearly indicates the direction of future work toward fast and efficient program analysis methods to identify and eliminate unnecessary lock barriers during JIT compilation.

Trademarks Sun, Sun Microsystems, Sun Enterprise, Java, JDK and Solaris, are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. SPARC and UltraSPARC are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries.

References

- [1] Atkinson MP, Daynès L, Jordan MJ, Printezis T, Spence S. An orthogonally persistent JavaTM. *SIGMOD Record*, **25**(4), December 1996.
- [2] Lindholm T, Yellin F. *The JavaTM Virtual Machine Specification, Second Edition*. The JavaTM Series. Addison Wesley, April 1999.
- [3] Atkinson MP, Morrison R. Orthogonal persistent object systems. *VLDB Journal*, **4**(3), 1995.
- [4] Daynès L, Atkinson MP, Valduriez P. Customizable Concurrency Control for Persistent Java. Chapter 7 in *Advanced Transaction Models and Architectures*, Jajodia S and Kerschberg L (ed), Data Management Systems. Kluwer Academic Publishers, Boston, 1997.
- [5] Dieckmann S, Hölzle U. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In Proc. of 13th European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, June 1999, Springer Verlag.
- [6] Carey M, DeWitt D, Naughton J. The OO7 benchmark. *ACM SIGMOD Int. Conf. on Management of Data*, Washington D.C, May 1993.
- [7] Moss JEB. Working With Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, **18**(8):657–673, August 1992.
- [8] Haines N, Kindred D, Morrisett J, Nettles SM, Wing JM. Composing first-class transaction. *ACM Transactions on Programming Languages and Systems*, **16**(6):1719–1736, November 1994.
- [9] Garthwaite A, Nettles S. Transactions for Java. *First International Workshop on Persistence and Java*, Drymen, Scotland, September 1996.
- [10] Kim W, Ballou N, Chou H, Garza JF, Woelk D. Integrating an object-oriented programming system with a database system. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 142–152, September 1988.
- [11] White SJ, DeWitt D. A performance study of alternative object faulting in pointer swizzling strategies. *18th Int. Conf. on Very Large Database*, 419–431, Vancouver, British Columbia, Canada, 1992.
- [12] Carey M, DeWitt D, Franklin M, Hall N, McAuliffe M, Naughton J, Schuh D, Solomon M, Tan C, Tsatalos O, White S, and Zwilling M. Shoring Up Persistent Applications. In *ACM SIGMOD Int. Conf. on Management of Data*, 383–394, 1994.
- [13] Versant. *Multi-thread Database Access*, May 1998. A Versant Application Note.
- [14] White SJ, DeWitt D. QuickStore: a high performance mapped object store. *VLDB Journal*, **4**(4), 629–673, 1995.
- [15] Biliris A and Panagos E. A High performance configurable storage manager. In *11th Int. Conf. on Data Engineering*, 35–43, Taipei, Taiwan, March 1995.
- [16] Lehman T, Gottenmukkala V. The design and performance evaluation of a lock manager for memory-resident database systems. In [27], 406–428.
- [17] Gray J. Notes on data base operating systems. Volume 60 of *Lectures Notes in Computing Sciences*, 393–481. Springer-verlag, 1978.
- [18] Gray J, Reuter A. *Transaction Processing : Concept and Techniques*. Morgan-Kaufman, 1993.
- [19] Lomet D. Key range locking strategies for improved concurrency. *19th Int. Conf. on Very Large Database*, 655–664, Dublin, Ireland, 1993.
- [20] Mohan C. Concurrency control and recovery method for B⁺-tree indexes: ARIES/KVL and ARIES/IM. In [27], 248–306.
- [21] Garcia-Molina H, Salem K. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, **4**(6):509–516, December 1992.
- [22] Daynès L. Implementation of Automated Fine-Granularity Locking in a Persistent Programming Language. In *Software – Practice and Experience* **30**:1-37, 2000.
- [23] Michie D. 'Memo' Functions and Machine Learning. *Nature*, (218):19–22, April 1968.
- [24] Lewis B, Mathiske B, Gafter N. Architecture of the PEVM: A High-Performance Orthogonally Persistent JavaTM Virtual Machine. *9th International Workshop on Persistent Object Systems*, Lillehammer, Norway, September 2000.
- [25] Daynès L, Gruber O, Valduriez P. Locking in OODBMS clients supporting nested transactions. *11th Int. Conf. on Data Engineering*, 316–323, Taipei, Taiwan, March 1995.
- [26] The Standard Performance Evaluation Corporation (SPEC). *SPEC JVM98 Benchmarks* <http://www.spec.org/osg/jvm98>.
- [27] *Concurrency Control Mechanism in Centralized Database Systems* Kumar V (ed), Prentice Hall, 1996.

LOCKSS: A Permanent Web Publishing and Access System

Vicky Reich and David S. H. Rosenthal

Introduction by David S. H. Rosenthal

LOCKSS (Lots Of Copies Keep Stuff Safe) is a joint project of Sun Labs and Stanford University Libraries. This paper appeared in the June 2001 issue of Lib, a premier online journal for the digital library research community. See <http://www.dlib.org/dlib/june01/reich/06reich.html>. The paper provides an overview of the project's goals and achievements to date.

Stanford University Libraries' HighWire Press began in early 1995 with the online production of the weekly Journal of Biological Chemistry (JBC), the most highly cited (and second largest) peer reviewed journal. Stanford's goal was to use first mover advantage with web publishing technology to set business and technology standards for online access to literature and to preserve and strengthen the viability of society publishers and their publications.

Stanford's HighWire Press now publishes the online editions of nearly 300 highly cited journals.

Librarians are key customers for these journals; they purchase institutional subscriptions on behalf of their community members. Librarians are reluctant to transition from print versions to online versions even though often, the online version contains more peer reviewed content and is more valuable for the user than the paper.

When librarians purchase a paper copy, they own that copy forever. When they purchase the online version of the content, they are renting access to someone else's copy (on a Web server). They are tasked with preserving their institution's long-term access to this material, and they have little confidence in the technical or business methods being proposed for this purpose.

LOCKSS is demonstrating that it is possible to use the techniques that librarians have developed over the centuries to preserve long-term access to published materials in the new Internet environment. Access to paper journals is preserved by making lots of copies, scattering them around the world at many independent sites, making it easy for a reader to find a copy, making it easy for a library that discovers its copy to be missing or damaged to find a replacement, and making it hard to find all the copies to modify or destroy them.

LOCKSS allows librarians to keep the purchase model by running independent web caches that collect the journals to which the library subscribes. The caches crawl the journal web sites, and deliver the journals to their readers if the journals are not available from the original publisher.

The caches at various libraries cooperate to detect and repair malicious or deliberate damage. The research interest for Sun Labs lies in the way this cooperation happens – the caches form a highly fault tolerant peer-to-peer system of a novel kind that addresses some unusual questions:

- What can we gain from having far more replicas than needed for conventional fault tolerance?
- How can we maintain a system's integrity against attacks by the bad guy if it is not possible to keep secrets?

The technical details were published in a paper at the 2000 Usenet conference, (<http://lockss.stanford.edu/freenix2000/freenix2000.html>) which contains a fairly extensive bibliography.

The LOCKSS project is unusual for Sun Labs in that it is a Stanford project, supported by grants from the National Science Foundation and the Andrew W. Mellon Foundation and by a grant and staff time from Sun Labs. Hosting the project at Stanford Library has made it possible to recruit the 45 libraries and 35 publishers who are supporting the current system test.

The LOCKSS article is reprinted from D-Lib Magazine, June 2001, Volume 7 Number 6, ISSN 1082-9873, with permission from Stanford University Libraries.

LOCKSS: A Permanent Web Publishing and Access System

**Vicky Reich, Stanford University Libraries
David S. H. Rosenthal, Sun Microsystems Laboratories**

vreich@stanford.edu, dave.rosenthal@sun.com

Abstract: LOCKSS (Lots Of Copies Keep Stuff Safe) is a tool designed for libraries to use to ensure their community's continued access to web-published scientific journals. LOCKSS allows libraries to take custody of the material to which they subscribe, in the same way they do for paper, and to preserve it. By preserving it they ensure that, for their community, links and searches continue to resolve to the published material even if it is no longer available from the publisher. Think of it as the digital equivalent of stacks where an authoritative copy of material is always available rather than the digital equivalent of an archive.

LOCKSS allows libraries to run web caches for specific journals. These caches collect content as it is published and are never flushed. They cooperate in a peer-to-peer network to detect and repair damaged or missing pages. The caches run on generic PC hardware using open-source software and require almost no skilled administration, making the cost of preserving a journal manageable.

LOCKSS is currently being tested at 40+ libraries worldwide with the support of 30+ publishers.

The Problem

The web is an effective publishing medium (data sets, dynamic lists of citing papers, e-mail notification of citing papers, hyperlinks, searching). Increasingly, web editions are the "version of record" and paper editions of the same titles are merely a subset of the peer reviewed scholarly discourse. Scientists, librarians, and publishers are concerned that this important digital material, the record of science, will prove as evanescent as the rest of the web. In addition, each of these communities has specific needs:

- Future generations of scientists need access to this literature for research, teaching, and learning.
- Current and future librarians need an inexpensive, robust mechanism, which they control, to ensure their communities maintain long-term access to this essential literature.

Current and future publishers need assurances that their journals' editorial values and brands will be available only to authorized and authenticated readers.

Therefore, the problem is to preserve an authorized reader's access to the web editions of scientific journals while staying within libraries' budgets and yet respecting publishers' rights.

Requirements

Technically, any solution must satisfy three requirements:

- The content must be preserved as bits;
- Access to the bits must be preserved;
- The ability to parse and understand the bits must be preserved.

There is no single solution to this problem, and furthermore, having only a single solution would mean that materials would still be vulnerable to loss or destruction. Diversity is essential to successful preservation. By proposing LOCKSS, we are not discounting other digital preservation solutions; other solutions must also be developed and deployed.

In particular, we believe that there are needs in this area for both centralized "archives" and a distributed library system like LOCKSS. For digital materials, the terms archiving, library, and preservation tend to be used interchangeably. For our purposes, we follow the dictionary in defining an archive to be a place where public records are stored, and a library to be a place where people read and/or study materials. Preservation is the action of keeping materials from injury or destruction. Both library and archive materials need preservation.

Models of Repositories

There are two models of digital preservation and archiving repositories: centralized and decentralized. A key question to ask of each model is "what are the costs of preserving different types of materials and on whom do these costs fall?"

The centralized model envisages a small number of tightly controlled repositories. Each repository a) does the entire job of preserving content, b) requires expensive hardware, and c) requires sophisticated technical staff. To establish a centralized system, publishers and librarians must take legal and data management actions cooperatively. Preservation costs are borne by a few. Centralized systems are focused on preserving bits and pay less attention to ensuring access. Indeed many of these systems are explicitly "dark" archives, in which content will not be accessed until some "trigger" event (migration, publisher failure, etc.) occurs.

The decentralized model envisages a large number of loosely controlled repositories. Each repository or node in the system a) does some but not the whole job of preserving content, b) uses relatively inexpensive hardware, and c) needs relatively little technical expertise to maintain the hardware and software. The content at each repository is in constant use, under constant scrutiny, and undergoing continual repair. In a decentralized system, publishers take little or no action to preserve the content they publish; librarians take action to preserve

access for their local communities. While libraries bear the costs of digital preservation, each participating library bears only a small fraction of the total cost, in proportion to its resources and priorities. The benefits accrue only to the participating library's communities. Decentralized systems are focused on preserving access rather than just preserving the bits. These systems count on the redundancy inherent in distributed systems to keep the bits safe.

Overview of LOCKSS

LOCKSS is a digital preservation Internet appliance, not an archive. Archives exist to maintain hard-to-replicate materials, and access is sacrificed to ensure preservation. LOCKSS is more akin to a global library system. Libraries hold fairly common materials in "general collections" with access as the primary goal. A key difference between LOCKSS and "general library collections" is that the action of preserving material in the collection is intertwined with the provision of access to the end user.



Librarians retain paper publications to ensure long-term accessibility. One could visualize all the libraries in the world as parts of a system — a very informal, highly decentralized, highly replicated system. The primary goal of this system is to provide access to material, but providing access in this way also ensures that documents are not lost as a result of publisher takeovers, malicious actions, natural disasters, or official edicts. Generally, someone at a local library will find it easy to access the paper copy of a particular book or journal. Once a book or journal has been published it is hard to "unpublish" it by finding and destroying all copies.

LOCKSS is modeled on this paper system. With the LOCKSS model, libraries run persistent web caches. Their readers can use these caches to access the journals to which the library subscribes, whether or not the journals are still available from the publisher.

The system makes it easy to find a copy of an article but hard to find all the copies of the article, thus making it hard to "unpublish" it. Very slowly, the LOCKSS caches "talk to each other" to detect missing or damaged content. If a cache needs to repair content, it can get a replacement from the publisher or from one of the other caches via "inter-library loan."

How the Data Flows

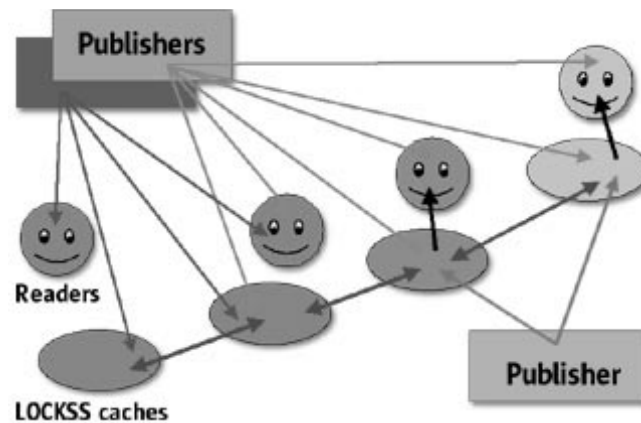


Figure 1: In this example, each LOCKSS cache (oval) collects journal content from the publisher's web site as it is published. Readers (circles) can get content from the publisher site. When the publisher's web site is not available (gray) to a local community, readers from that community get content from their local institution's cache. The caches "talk" to each other to maintain the content's integrity over time.

Reader's Perspective

LOCKSS' key goal is to preserve a reader's access to content published on the web. Readers expect two kinds of access. They expect that:

- When they click on a link to it, or type in a URL, the relevant page will be delivered with minimal delay and no further interaction.
- When they enter terms into a search engine that should match the relevant page, it will be among the returned matches.

Readers who use the web are learning that if a link doesn't resolve to a page, or a search engine can't find a page, further attempts to find the information the page carries are unlikely to be worth the effort. This poses problems for those who use preservation techniques that concentrate on preserving bits; the bits may be preserved yet the reader may not know how to access them, or even that the preserved bits exist.

In contrast, LOCKSS focuses on preserving the service of having links resolve to, or searches to find, the relevant content. An institution using LOCKSS to preserve access to a journal in effect runs a web cache devoted to that journal. Readers use the cache as a proxy in the normal way. At intervals the cache crawls the journal publisher's web site and pre-loads itself with newly published (but not yet read) content. Just as other types of caches are invisible to their users, so is LOCKSS. The LOCKSS cache transparently supplies pages it is preserving even if those pages are no longer available from the original publisher's web site.

An institution can include the contents of the cache among the pages indexed by its local search engine and provide its readers with searching across all the journals to which it subscribes. At present, readers typically have to search individual collections of journals separately.

Librarian's Perspective

Librarians subscribe to journals on behalf of their readers in order to provide both immediate and long-term access. With the advent of the web, for the most part, libraries are forced to lease rather than own the web-based content. Leasing provides immediate access but carries no guarantee of long-term access. Some journals provide their peer reviewed content through off line storage media (tape, CD-ROM, paper), but then links don't resolve and searching is harder to accomplish. A major flaw with web publishing is that there has been no mechanism to implement the traditional purchase and own library model.

A major flaw with web publishing is that there has been no mechanism to implement the traditional purchase-and-own library model. LOCKSS is changing this by demonstrating that it is both easy and affordable to operate a purchase model for web journals. The subscribing library bears costs analogous to the costs of putting paper copies on shelves, keeping track of them and lending or copying them as needed. A library using LOCKSS to preserve access to a collection of journals pays for the equipment and staff time to run and manage a cache containing the full content of the journals. Unlike normal caches, the LOCKSS cache is never flushed, and over the long term, the full content remains accessible.

Because individual libraries must pay for the preservation of the content to which they subscribe, it is essential that the price they pay be as low as possible. LOCKSS is free, open-source software designed to run on inexpensive hardware. The machines the LOCKSS team is using for the beta test cost less than \$800 each, and each machine is capable of storing the content of 5 years worth of a major journal's issues. Running LOCKSS requires so little staff time that one alpha test site complained they learned nothing about the system over the course of 10 months while running it. LOCKSS' low cost and democratic structure (each copy is as valuable as any other) empowers smaller institutions to take part in the process of digital preservation.

In normal operation, an ordinary cache will only act as a proxy for, and thus supply content to, the host institution's own readers. But in a rough analog of inter-library loan, LOCKSS caches cooperate to detect and repair damage. If damage to a page is detected, the LOCKSS cache fetches a copy of the page from the publisher or from another cache. A LOCKSS cache will only supply a page to another LOCKSS cache if the requesting cache at some time in the past proved that it had the requested page. In this way, LOCKSS prevents freeloading. Those who contribute to the preservation of the journal are rewarded with continued access; those who do not contribute to the journal's preservation are not provided with replacement pages.

Publisher's Perspective

Publishers want to maintain journal brand and image. They want material available for future society members and other subscribers. Most publishers will save money and serve their readers better if the transition to electronic-only journals can be completed. They want to encourage libraries to purchase and/or activate online versions of journals. One major obstacle to libraries purchasing online journals is resistance to the rental model with its lack of credible assurance of long-term access. Publishers are unhappy with a purchase model for electronic journals because:

- They fear the journal content will be illegally replicated, or leaked, on a massive scale once copies are in the custody of others; they want their access control methods enforced.
- They want to retain access to reader usage data and have access to the record of the reader's interactions with their site.

LOCKSS solves the reader's and the librarian's problems. It enables librarians to collaborate to preserve readers' access to the content to which they subscribe, but it also addresses the publisher's concerns.

- Because content is provided to other caches only to repair damage to content they previously held, no new leakage paths are introduced.
- Because the reader is supplied preferentially from the publisher, with the cache only as a fallback, the publisher sees the same interactions they would have seen without LOCKSS.

LOCKSS has other advantages from the publisher's perspective:

- It returns the responsibility for long-term preservation, and the corresponding costs, to the librarians. Although publishers have an interest in long-term preservation, they cannot do a credible job of it themselves. Failures or changes in policy by publishers are the events librarians are most interested in surviving.
- Publishers could run LOCKSS caches for their own journals and, by doing so, over time could audit the other caches of their journals. A non-subscriber cache would eventually reveal itself by taking part in the damage detection and repair protocol. The mere possibility of detection should deter non-subscribers from taking part in LOCKSS. Just as the publisher cannot be sure he has found all the caches, the caches cannot be sure none of the other caches belongs to the publisher.

How Does LOCKSS Preserve Content?

LOCKSS has two tasks in preserving content:

- It needs to detect, and if possible, repair, any damage that occurs through hardware failure, carelessness, or hostile action.
- It must also detect, and if possible, render ineffective, any attacks.

Detecting and Repairing Damage

Damage to contents is a normal part of the long-term operation of a digital storage system. Disks fail, software has bugs, and humans make mistakes. To detect damage, the caches holding a given part of a journal's site vote at intervals on its content. They do so by calculating digital hashes of the content and running polls on the values of these hashes:

- In the absence of damage, the hashes will agree.
- If they disagree, one of the losers calls a sequence of polls to walk down the tree of directories to locate the damaged files.
- When a damaged file is located, a new copy is fetched to replace it.
- If the file is not available from the publisher, it will be requested from one of the winning caches.
- If a cache receives a request for a page from another cache, it examines its memory of agreeing votes to see if the requester once agreed with it about the page in question. If the requester did, a new copy will be supplied.

LOCKSS polls are not like the elections of conventional fault-tolerant systems in which voting is mandatory. They are like opinion polls, in which only a sample of the potential electors takes part. Only a sample of the LOCKSS caches holding given content vote in any one poll. Note that even caches that don't vote hear the votes of those that do. They can decide whether they agree or disagree with the majority in the poll. Normally there will be no damage, and each of the polls will be unanimous. If there is a small amount of random, uncoordinated damage, each poll will be a landslide, with only a few disagreeing votes. In normal landslide polls, the majority of systems will be reassured that their copy is good without voting.

Details of this system were presented to the 2000 Usenix conference (<http://lockss.stanford.edu/freenix2000/freenix2000.html>).

Hampering the "Bad Guy"

If there is ever a poll whose result is close, it is highly likely that a "bad guy" is trying to subvert the system. Attempts at subversion are endemic in peer-to-peer systems like LOCKSS. (See <http://www.wired.com/news/infostructure/0,1377,41838,00.html>.) A "bad

guy's" goal might be to change the consensus about some content in the system without being detected.

- A "bad guy" who infiltrated only a few caches and made matching changes to each of their contents would appear to be random damage. The other caches would not change their contents to match.
- If the "bad guy" infiltrated a substantial number of caches, even a small majority, he would cause polls whose results were close. In the event that the material was no longer available from the publisher, the caches which had not been subverted might replace their good copies with the "bad guy's" modified ones. However, the close results of the polls would alert the system's operators that something was wrong.
- Only if the "bad guy" infiltrated the overwhelming majority of caches would his change be both effective and undetected. The remaining "good" caches would appear to have been damaged and would fetch the "bad guy's" versions.

Another obstacle for the "bad guy" is that LOCKSS is designed to run extremely slowly. A single poll may take days. It takes many polls to have an effect. By preventing the system from doing anything quickly, we prevent it doing what the "bad guy" wants. Because it is difficult for the "bad guy" to operate without raising an alarm, it is likely that the administrators of the system would notice and react to attempts at subversion while those attempts were underway. The inter-cache communication can run very slowly because it does not delay readers' accesses to the journals, which are purely local operations.

The difficulty in infiltrating an overwhelming majority of caches lies in being sure you have found enough of them. If there are a lot of caches, it will be a long time between votes from any one of them. The set of caches holding the material to be attacked will be changing over a somewhat slower time-scale as caches die and are born. Because the underlying communication protocol is inherently unreliable, some caches will not hear some votes. These uncertainties work against the "bad guy".

Reputation System

To make life even harder for the "bad guy", each LOCKSS cache maintains a record of the behavior of the other caches in the form of a reputation. As caches are observed taking good actions, their reputation is enhanced; as they are observed taking bad actions, their reputation is degraded. When a cache tallies a poll, it will take action only if the average reputation of those voting on the winning side is high enough. This has three effects:

- A "bad guy" can only have an impact by behaving well for long enough to build up a good reputation. If the "bad guy" has to spend most of his time acting as a "good guy" the system may get sufficient benefit from the good actions to outweigh the bad ones.

- The "bad guy" must achieve his nefarious aims quickly, before his reputation is eroded far enough to render him ineffective. But the system is designed to run very slowly.
- The "bad guy" might find ways to skew the sample to include his co-conspirators and exclude the "good guys". However, unless the conspiracy overwhelms the "good guys" very quickly, the appearance of "bad guys" acting in concert will damage their reputations even faster than if they were acting alone.

More detail on the struggle between good and evil in the LOCKSS context can be found at <http://lockss.stanford.edu/lockssssecurity.html>.

Running LOCKSS

The current LOCKSS version runs on generic PCs. At current prices, a suitable machine with a 60GB disk in a 1U rack-mount case should cost about \$750. The system is distributed as a bootable floppy disk. The system boots and runs Linux from this floppy; there is no operating system installed on the hard disk. The first time the system boots it asks a few questions, then writes the resulting configuration to the floppy, which is then write-locked. At any time, the system can be returned to a known-good state by rebooting it from this write-locked disk.

Each time the system is booted, it downloads, verifies and installs the necessary application software, including the daemon that manages the LOCKSS cache and the Java virtual machine needed to run it. The system then runs the daemon and starts the HTTP servers that provide the user interface web pages. The cache's administrator can use these pages to specify the journal volumes to cache and monitor the system's behavior.

Project Status

Up-to-date project status is available at <http://lockss.stanford.edu/projectstatus.htm>.

Alpha Test

Design and development of LOCKSS started in 1999. Testing started on 6 old 100MHz PCs late that year. An "alpha" version of the software, without a user interface or any precautions against the "bad guy", ran from May 2000 through March 2001 with around 15 caches. The test content was about 160MB representing three months of AAAS Science Online. Alpha sites were Stanford University, the University of California, Berkeley, the Los Alamos National Laboratory (LANL), the University of Tennessee, Harvard University, and Columbia University. This test established that the basic mechanisms worked. The system was able to collect the test content and repair both deliberate and accidental damage to it. The system survived a fire at LANL, network disruptions at Stanford, relocation of the machine at Berkeley, and flaky hardware at Columbia.

Beta Test

The worldwide "beta" test began in April 2001, using an almost complete implementation of the system. Approximately 35 publishers are endorsing the test. Over 40 libraries, with about 60 widely distributed and varyingly configured caches, have signed on to the project. They include major institutions, such as the Library of Congress and the British Library, and smaller institutions, such as the University of Otago in New Zealand.

Beta will test the usability and performance of LOCKSS, measure its impact on network and web-server traffic, and provide some estimates of the costs of running an individual cache and the system as a whole. The test content is a total of about 15 GB from BMJ, JBC, PNAS and Science Online. The test content is provided to the caches by shadow servers, which partially mirror the publishers' websites. They isolate the LOCKSS data streams and allow us to simulate journal failures. We hope in the later stages of beta to add other publishers' journals, and other types of content such as government documents.

Acknowledgements

The Stanford University Libraries LOCKSS team members are: Vicky Reich, Tom Robertson (HighWire Press), David Rosenthal (Sun Microsystems), and Mark Seiden (Consultant).

The National Science Foundation, Sun Microsystems Laboratories, and Stanford University Libraries funded development and alpha testing of LOCKSS. The worldwide "beta" test in 2001 is made possible through a grant from the Andrew W. Mellon Foundation, equipment donated by and support from Sun Microsystems Laboratories, and support from Stanford University Libraries.

We are grateful to the contributors at our alpha sites: Dale Flecker and Stephen Abrams, Rick Luce and Mariella DiGiacomo, David Millman and Ariel Glenn, Bernie Hurley and Janet Garey, Chris Hodges and Hal Clyde King, and Jerry Persons. Special thanks are due to Michael Lesk, Michael Keller, Bob Sproull, and Neil Wilhelm.

The Road to Brazil

aka Design and Architecture of the Brazil Web Application Framework

Stephen A. Uhler

Introduction by Stephen A. Uhler

The Brazil project defines a new concept in web service architecture. It views the web, not just as a sea of browsers talking to large content sites, but instead as a network of services and proxies where content passes through a sequence of processing steps throughout the network, is filtered, combined with other content and modified along the way. This architecture encourages small content sources to become "web enabled", such as devices with attached sensors and actuators. The content is not intended to be consumed directly by user agents, but combined with content from other sources by powerful "meta servers".

The Brazil project has only been around for a short time; however many of the concepts embodied by Brazil have been cultivated from seeds, sown as part of other research projects by the members of the Brazil project team. Somewhere along the way, while trying to convince the IT organization of an important customer that, yes, it is possible to build enterprise applications using a web browser as the only user interface, we were reminded of a scene from a movie involving the Ministry of Information, and the name, Brazil, like the movie, stuck.

The Brazil project is a sample implementation of this architecture that provides the infrastructure to create three classes of systems. The first class of systems, "micro-servers", enable web-based access to small content sources by defining a UPI, or URL Programming Interface that maps device capabilities into HTTP URLs. "Meta-servers", the next class of Brazil systems, combine and filter content from other content sources. This allows bits of data and control instructions from devices, via "micro-servers", to be integrated seamlessly into the content of traditional web sites. Finally, The Brazil project may be used to create traditional web sites, but with the flexibility to add dynamic filtering, content aggregation and personal preferences.

The Brazil project has advanced the state of the art in web services in three key areas:

- the separation of presentation from content
- the architectural reformulation from a monolithic client-server style system to a dynamic data-flow component architecture
- the retrieval and semantic extraction of content from existing and legacy systems, and its subsequent integration to create new presentation

As the variety and sophistication of web users and services increases, so does the coupling between the data presentation and the programming logic required to extract and generate the raw data. The Brazil project implementation defines a novel way to separate content from presentation, using an XML-based scripting language, that allows the same content to be reformatted to suit a variety of user needs, without requiring any modification of the content generation code. As an example, the same content can be dynamically reformatted for use by a desktop browser, a web enabled cellphone, or another Web service that requires XML.

Motivated by the wide range of computing systems that participate on the web, from \$50 web-enabled sensors to \$10M+ servers, the Brazil project defines a scalable component model of web services. These services are created by combining simple components in consistent ways, thus eliminating the complex API's and Interfaces that are typical of today's monolithic systems. Only

the required components need to be included in any particular web service, allowing deployment to small environments, while retaining the richness of capabilities for large ones.

While the Brazil project architecture looks toward the future of web services, it provides facilities to web-enable existing and legacy applications now, thus extending their useful lifetimes. This is accomplished by adding Web service front ends to the existing systems, and using the dynamic reformatting capability mentioned above to convert the operations of existing systems into web services.

The Brazil Project has already made significant contributions to Sun's products. The prototype version of sun.net, especially the reverse proxy – a key component of the sun.net system – was built using the Brazil project component architecture. The initial sun.net applications, such as the calendar, demonstrated the feasibility of creating as web services, applications previously considered only viable as desktop applications.

As the web services infrastructure matures, we expect the Brazil project to continue to contribute to Sun's success, by providing a model by which the existing commercial systems can evolve, both to provide better separation of content from presentation, more data integration and aggregation from different sources, and methodologies that will permit these systems to scale down as well as up.

A sample website built using Brazil technology, demonstrating some of its features, including complete documentation, and containing both runnable code and Java™ language source code is available at: <http://www.experimentalstuff.com>.

Design and Architecture of the Brazil Web Application Framework

Stephen A. Uhler

SMLI TR-2001-99

September 2001

Abstract:

Brazil is a new architecture for building Web applications that is simple, modular and scalable. It embraces a new vision of the web that consists not simply of clients talking to content-servers, but instead pictures a network of portals and content aggregators interposed between the providers and consumers to create a personalized web of customized content, dynamically synthesized by integrating information from a wide range of sources.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
stephen.uhler@sun.com

Design and Architecture of the Brazil Web Application Framework

Stephen A. Uhler

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, California 94303

Introduction

Web servers and their more impressively named cousins, "Web Application Frameworks", constitute the single most important component of the network content delivery system we know as "The Web". The first Web Servers started to appear in 1994 on UNIX® systems on the Internet. The design of those early systems reflects their UNIX heritage. URLs (Uniform Resource Locators) are equivalent to UNIX file names. Each URL, when requested by a client program, typically a "Web Browser", is mapped to the UNIX file of the same name, wrapped in HTTP (HyperText Transport Protocol), and delivered to the client. In those cases where content cannot be represented as a static file and needs to be dynamically generated, the URL names the program that is run to generate the content. This capability, known as CGI (Common Gateway Interface), stems from the traditional UNIX practice of making everything look like a file. Thus each URL represents a file that either contains the content, or contains the program that is used to generate the content.

Background

Over the next five years the WEB saw explosive growth, and the architecture of the original Web Servers, though simple and elegant, was beginning to strain. Static content was still delivered effectively by mapping URLs into files, but dynamic content was becoming problematic. The notion of programs as files, as well as the mechanisms for identifying, launching, managing, and communicating with CGI programs is very specific to the UNIX operating system, which makes porting web servers and their corresponding content to non UNIX systems difficult. In addition, as content management techniques required more of the content to be generated dynamically, even if simply to paste together several static files in response to a single URL, the CGI programs rapidly became the bottleneck. Each dynamic page requires a separate program to be launched and executed by the operating system, only to be terminated each time a request is completed. In addition, the communication between the Web Server and the CGI program is very limited.

Only the URL and its corresponding HTTP envelope information is made available to the CGI program, which can only return content; the ability to pass meta-information back to the server is almost nonexistent.

The next state in the evolution of Web servers focused on eliminating the CGI bottleneck, specifically the program creation and execution step required for each URL requested. Generally, three different approaches have been taken: keeping the basic CGI interface, only making it faster; building web server specific APIs, often by requiring the dynamic code generating portions to be bound into the same process as the web server; or defining language-specific APIs whose implementations don't require the overhead implied by the CGI model.

The *FastCgi* interface tries to improve the performance of the CGI specification by eliminating the process creation and execution step at every request, yet maintaining backward compatibility wherever possible. The *FastCgi* interface, represented by the file that maps from the URL, is created and started once when the web server starts. Multiple requests for the same URL are sent to the same *FastCgi* process by defining a request packet protocol than can accommodate multiple requests and responses for each *FastCgi* process. *FastCgi* has the advantage of preserving a separate execution context for dynamic content generation, while eliminating the bulk of the process creation overhead of

traditional CGI programs. Consequently *FastCgi* programs are easily ported to work with many different web servers.

The second approach to eliminating the CGI bottleneck is to move the dynamic content generation into the same execution context as the server, by expressing dynamic content generation in terms of APIs that are specific to a particular web server. This approach eliminates the process creation and execution overhead of CGI programs entirely, but at the expense of close coupling to a particular web server. Most major web servers provide such API definitions. However, dynamic content generation using these APIs is rarely portable to a different server. In addition, by having the dynamic content generation in the same execution context as the server, a bug in a dynamic content generation module can negatively impact the entire web server, including URL requests that have nothing to do with the bug-containing module.

The third approach used to eliminate the CGI bottleneck is to create a set of language-specific APIs that can be logically bound into the execution context of the web server, yet be defined in a web server independent way. Servlets are the leading example of this approach. A servlet is a Java™ programming language that conforms to a defined set of Java APIs, which can be (and have been) implemented to provide dynamic content for many different web servers. Thus servlets combine the advantages of *FastCgi* -- portability to

different web servers -- with the close coupling of server specific extensions.

The issues

Although all three approaches reduce the performance problems associated with the CGI interface, they still fundamentally retain the notion of a one-to-one mapping between URLs and files, where URLs are unrelated to each other.

As the web has grown, the notion that every URL request and its associated file is independent of any other request has become a serious architectural roadblock. It is now common for a single Web "form" to be spread over multiple pages (URLs), or for a single user to have unique state associated with a sequence of requests that may span days or even years. Finally, as the sheer volume of content on the web has mushroomed, it is no longer appropriate to assume, as is implicit in the CGI one-file-per URL model, that the content resides on the server machine at all. The software architecture that was designed to deliver individual pages in response to URL requests is now being used to build sophisticated applications, whose content happens to be wrapped in HTTP. Somewhere in the switch from delivering static files as URLs to creating full-blown applications, web servers became web application development frameworks.

As the need for more sophisticated features has grown, so too have the capabilities of the web servers used to implement them. However, they are still

based on the original one file per request architecture that was seemingly elegant in the old days, but now just gets in the way. To support these added capabilities, the size and complexity of the APIs has grown. The descendants of the CGI architecture are stressed to provide functionality that isn't a good fit for their designs. As an example, a recent Servlet API (2.0) needs over two dozen classes and almost ten times that many methods to describe its interface.

To be fair, the entire reason for the explosion of interface complexity isn't totally due to the complexity of the interactions required by implementors of the interface. As web servers have become web application frameworks, the notion that the same pile of content can be delivered by any server has persisted. Somehow the "content" is viewed as separable from the server used to deliver it. Consequently, every new web server that arrives on the scene feels obliged to incorporate every nuance and capability of every previously deployed server, to insure that pre-existing content can be delivered with the new software without change. This "feature-bloat" adds significantly to the size of the system, while providing only a small increase in capability.

A new vision for the web

As the web matures, we see a transition away from the current client-server paradigm, where browsers are clicking at particular web sites, whose servers deliver all the content on that site. Instead, a more distributed model will

emerge. In this new model, both the traditional browsers and servers will still exist, but the content received by the client for a particular page is likely to have been retrieved from several traditional back-end servers, and modified to suit the requirements and desires of the particular client. This is akin to a traditional *workflow* business model, where the content passes through various stages, and is transformed at each stage.

Early versions of these intermediate stages, we'll call them *meta-servers*, are already starting to appear on the web. Some of the *meta-servers* are called "portals", and others are known as "content aggregators". In our view, portals and content-aggregators are one in the same. It looks like a portal when viewed from the perspective of the client, and a content aggregator from the perspective of the traditional server (we'll dub *content-server*).

As these *meta-servers* begin to play a more prominent role in the infrastructure, they will have a profound impact on the way in which traditional content-servers are constructed. No longer will the content-server produce both the content and its presentation (look and feel). Meta-servers will transform the content after it leaves the content-server, allowing content-servers to be simpler. Today's content-servers not only provide the content, but manage the presentation, user preferences, and browser differences as well. In the future, content-servers can be simpler, providing just the content. The integration with other content, as well as

the shaping of the look and feel for a particular browser will be added in stages by various *meta-servers* as the content flows toward the ultimate consumer.

Many types of content that are not traditionally located on a web server will become available. This new content, not able to stand on its own in the traditional web world, will be consumed by *meta-servers* which will integrate it with information from other content and meta-servers. Devices, sensors, and actuators will be accessible over the web, and will have their information integrated into the web fabric created by the network of content-servers, devices, and *meta-servers*.

Brazil

Brazil is a new architecture and sample implementation for building both content-servers and *meta-servers*. In the content-server context it permits the attachment of simple devices to the web with the barest of capabilities, squeezing into the tiniest places - a *micro-server*. In the *meta-server* context, it provides rich and flexible mechanisms for synthesizing, transforming, and integrating content: content retrieved both from traditional content-servers as well as the new breed of *micro-servers*. Finally, the architecture provides capabilities to integrate with traditional N-tier applications, providing the bridge between the current client-server web into the future.

To achieve this two part goal, a two part strategy is taken. The existing notion of mapping URLs to UNIX files is abandoned. Modern URLs are too fluid to have a fixed binding to underlying files. Indeed, many small devices have no notion of file systems at all. Specific mechanisms used to implement existing content-server capabilities are discarded. For example, most traditional content-servers use `.htaccess` files to provide password protection for content. The file based nature of the `.htaccess` mechanism is inappropriate for the Brazil architecture, so `.htaccess` support is not built-in. Password protected URLs are still available, albeit via a different mechanism.

The second part of the strategy is based on defining a series of abstract capabilities for Brazil that support the entire range of applications, from the tiniest *micro-server* to a more traditional *content-server* to a sophisticated *meta-server*. This determines an architecture that starts with the small core and simple interface for adding functionality required for a *micro-server* implementation, and adds to it a set of composable, interchangeable modules that can operate together in a scalable way. With modules for manipulating traditional file-based content, the traditional content-server capability can be obtained. By adding modules that can string together arbitrary relationships between users and pages, and combining them with modules that can obtain and manipulate foreign content, sophisticated *meta-servers* are possible.

The Brazil architecture

Four key components and the inter-relationships between them define the Brazil architecture. These components are described in the context of the prototype implementation, written in the Java programming language. Java objects represent two of the components, called `Server` and `Request`. The third component, a Java interface definition called a `Handler`, is the mechanism by which functionality is added into Brazil. The final Brazil component is the data structure for managing the information flow between the other parts, called the Brazil *properties*, named after the Java base class used in the prototype implementation. The *properties* are the name/value pairs that represent the current state of a URL request, along with methods for managing both the lexical and temporal scope of the data.

Building *micro-servers* with Brazil

As content management capabilities are shifted from traditional web servers to meta-servers, the traditional web server can focus entirely on the content it needs to deliver. At the extreme, it becomes a micro-server, delivering domain specific content in a bare bones way. These smaller, simpler servers can now be attached to sources of content that previously would be considered too small or unimportant to justify their own web servers. Examples include a digital thermometer whose content consists of the temperature of something, or a light switch, whose content is either *on* or *off*.

Although a web server whose content consists entirely of "on" or "off" might not make it into the website "top ten" list, when used in conjunction with meta-servers that can aggregate content from this and hundreds or millions of other similar servers, the content suddenly becomes quite interesting.

We use the term UPI, which stands for URL Programming Interface, to talk about the capabilities of these micro-servers. A UPI is just like an API, or Application Programming Interface, traditionally described in terms of specific programming language bindings, only UPIs are described in terms of URLs. Taken in this light, a URL no longer represents a file, instead it represents a set of programmable interfaces or remote procedure calls, that happen to be accessible via HTTP.

Using Brazil as a micro-server becomes defining a UPI for the desired functionality, using the built-in HTTP protocol stack as the transport mechanism, and writing the code to adapt the existing applications functionality to the UPI.

The `Server` object is the simplest of the four Brazil components. It represents the information relevant for the life of the Brazil server. This includes the port number the server is contacted on, the name of the `handler` (described below) that will turn a URL request into content, and an initial set of `properties`, used by the `handler` (or `handlers`) to satisfy an HTTP request. A Brazil application may have one or more active

`Servers`, which usually operate independently.

When a URL request arrives at the `server`, it creates a `Request` object. The `Request` contains all of the information that pertains to client's URL request as well as methods that encapsulate the HTTP protocol. Then the `Server` arranges for all information pertinent to this URL request to be added to the `properties` object. Finally, the `handler` is called to produce the content.

A `handler` is the interface that defines how URLs get mapped into content. It consists of two methods, `init` and `respond`. When the `Server` starts, it creates an instance of the `handler`, and calls its `init` method, providing it with a reference to the `Server` object. Each time a `Request` object is created, in response to an HTTP request, the `respond` method is called, and supplied the `Request` object as a reference. The `Handler` examines the request, and by using the methods in the `Request` object, formulates an HTTP response. Once the request has been satisfied, the `Request` object is discarded.

If any parameters are required to configure the `handler`, they are placed in the "properties" when the server is started. The `handler` can find its configuration information either in the `Server` object passed to the `init` method, or in the `Request` object provided with each request.

The setup described so far is ideal for "micro-server" applications. The `Server` and `Request` objects provide the framework for encapsulating and managing HTTP requests, and the handler maps the URLs onto device specific functionality. There is little unused infrastructure, and implementations can be made quite small. Configuration information required by the handler is provided to the server at startup time, and passed to the handler when its methods are called.

Building *meta-servers* with Brazil

The creation of *meta-servers*, that operate both as portals and content aggregators, use the same framework, and the identical interfaces as the micro-server. However, instead of building a system from a single handler that would need to be modified or rewritten for each new *meta-server* application, the meta-server is constructed as a cooperating collection of handlers, whose arrangement and configuration can be modified to provide a wide range of capabilities.

Because the handler interface is so small, it is easy to create a handler that functions both as a consumer of the handler interface, as in the micro-server example above, and as a provider of the handler interface. This insight lets us create a handler that calls other handlers, permitting multiple handlers to participate in the processing of each HTTP request. By combining these "interior node" handlers with the simple, or "leaf" handlers, a directed graph of

handlers can be created. This permits the construction of *meta-servers* by combining small bits of reusable functionality together.

A simple yet powerful use of "interior node" handlers in Brazil can be illustrated by the Brazil `ChainHandler`, which *chains* together a list of other handlers (possibly including other `ChainHandlers`), forming the basic mechanism for creating handler trees.

As indicated above, the data used to configure the handler is placed into the "properties" when the server is started. As long as there is only one handler, this scheme works fine. However, when multiple handlers are used in the same server, configuration collisions can result either from different handlers choosing the same name for a configuration parameter, or the same handler instantiated multiple times with different configurations.

To overcome this limitation, configuration properties for handlers are statically scoped within the `properties` to allow handlers that use the same configuration property names to have different values. Each "interior node" handler is responsible for creating the set of handlers that use it as the containing side of the handler interface. When each of the handlers is created, it is assigned a name which it uses to identify its configuration parameters, thus avoiding any possibility of name collisions.

For a handler used in the "micro-server", any request that is not dealt with

explicitly results in the server sending the requester an HTTP Not Found response. In the *meta-server* case, where multiple handlers have the opportunity to examine and respond to an HTTP request, a handler may alter the state of the current HTTP request without providing content to the requester. This alteration can take the form of modifying the configuration parameters of other handlers by changing the appropriate values in the "properties" object.

Because the "properties" is a stack, and handlers typically retrieve their configuration properties from the top of the stack, the duration over which a handler's configuration is altered is controllable by manipulating the "properties" stack. In the common case, changes one handler makes to another's properties will only be in effect for the duration of the current request.

A simple *meta-server* example

A common feature of many web servers is to allow users on a timesharing system to have their own private directory of files that are delivered as URLs. URLs that begin with `/~joe` would be delivered from *joe's* private directory of files instead of the main server directory. While most servers have this special capability built-in, the same effect is easy to provide in Brazil with a pair of handlers that co-operate with each other. The `FileHandler` is used to convert URLs into UNIX file names, and deliver the content of the files to the client. It is configured with a *document*

root, the directory in the file system that acts as the root of the URL space.

To manage user's files, a "user-file" Handler is run *before* the `FileHandler`. If the URL starts with `/~` the handler modifies the request by changing the URL by removing the user name portion, and setting the `FileHandler's` document root parameter in the "properties" to the proper user's home directory.

When the `FileHandler` gets the request, it delivers the proper file from the user's directory, based on the new configuration parameters placed in the "properties". When the next request comes in, the new configuration information will have been popped from the properties stack, and be unaffected by the previous modification. The same file handling code is reused in a different context.

Just as the "user-file" handler permits reuse of existing capabilities, by changing the "properties" to reconfigure the server "on-the-fly" in response to a particular request, other "handlers" use the same technique to provide password protection, session management, URL mapping, and a host of other services.

Important components

Just having a mechanism for composing handlers is not sufficient for creating a full featured meta-server. That requires many handlers, each performing a different task, but working together to

create a powerful content manipulation environment. In this section we'll visit some of the more important handlers and describe how they work together to create the Brazil *meta-servers* environment.

The first class of handlers, of which the `FileHandler` described above is an example, can be used together to provide the functionality of a traditional web server, including delivering files, running CGI scripts, providing password protected pages, and interfacing to other, non HTTP protocols such as LDAP or JDBC. Sometimes the meta-server needs to act as a traditional web server too.

The next class of handlers performs the content aggregation capability required by a meta-server. These handlers act as HTTP clients and retrieve content from a different server. The core of this capability is a fast proxy that implements the client side of the HTTP protocol. The `ProxyHandler` causes entire web sites to appear as if the content were stored locally in files. As each URL is retrieved from a content-server, the contents are examined, and every URL that points back to the content-server is rewritten so as to appear locally. When used in conjunction with the `FileHandler`, the `ProxyHandler` provides the illusion of a single UNIX filesystem, where arbitrary sub-directories are actually retrieved dynamically from other servers. This capability, called "web mount", provides an analogous set of semantics as the "filesystem mount"

facility does for ordinary files in the UNIX operating system.

Another interesting content aggregation handler is designed for use with micro-servers (or traditional web servers), whose simple content needs to be integrated with additional data in order to be presented to the user in a meaningful way. Content retrieved from this handler is converted into a set of name/value pairs and placed into the "properties" for further processing. The property values are then used by other handlers to formulate the final response. The content may be extracted from other web sites synchronously each time a request arrives, or in the background, updated on a periodic basis. Using the background method, the client doesn't need to wait for the data to come from the other server; the most recently obtained values are used instead. An interesting variation on the use of this handler is the ability to provide micro-servers that dynamically affect the operation of the main server, by returning values that represent configuration parameters of one or more handlers in the main server.

The third category of handlers is used to manipulate content once it has been obtained. These handlers come in two flavors, content extraction and content integration. The content extraction handlers use the HTML and XML processing capabilities provided by Brazil to analyze and decompose content, and convert it into name/value

pairs that are stored in the "properties". It doesn't matter whether the content was obtained locally, from a file, or remotely from a remote content-server.

The content integration handlers also use the HTML and XML processing capabilities, but this time to insert the previously extracted content into XML templates for final delivery to the requester. The "properties" are used as the rendezvous location, not only to characterize both the HTTP request and the server configuration, but to hold extracted content as well. Handlers can access and manipulate all three kinds of data in a uniform manner.

Handlers in the final category, unlike the other handlers described so far, don't participate directly in generating or manipulating requests or responses. Instead, they are used to insert alternate implementations for key data structures used by the server. For example, there is an implementation of "properties" that may be installed, with a handler, that causes portions of the name/value pairs to be stored and retrieved by a database, providing horizontal scalability and persistence for demanding applications.

Simply by rearranging handlers, and changing the way they interact with each other, a wide variety of web services can be created, often without the need to create new handlers. The following two examples are typical of services that are easily crafted using the current Brazil implementation. The first Brazil micro-server defined a UPI for accessing smartcards. We were able to add

smartcard based identity, authentication, and payment to several existing web-sites, with only minor changes to the existing sites. As is often the case with Brazil applications, we were able to reuse the smartcard UPI in a totally different context: combining it with the Brazil public key Certificate Authority handler to enable smartcard-authenticated delivery of public key certificates.

We built a micro-server that extracts real-time sensor data from home appliances, and a corresponding meta-server that inserts the sensor data into pages of an existing website, while not requiring a single modification to the original web site. From the perspective of the user, the appliance sensor data appears to be seamlessly integrated into the original web site.

Summary

By using a simple interface, in conjunction with powerful, reusable components, the Brazil system is able to deliver a wide range of flexible web solutions, ranging from tiny micro-servers, to traditional web capabilities to fully functional meta-servers that provide sophisticated portal and content aggregation capabilities.

References

<http://www.experimentalstuff.com/>

A web site that uses, describes, and supplies downloads of the current Brazil prototype.

Index

Adams, Stuart	10-1
Baatz, Eric	10-1
Bergman, Eric	12-1
Bernabeu, Jose M.	9-1
Bookman, Lawrence A.	20-1
Bush, Bill	18-1
Cidon, Israel	16-1
Crabbe, Frederick	10-1
Czajkowski, Grzegorz	23-1
Daynès, Laurent	23-1
Diffie, Whitfield	22-1
Ditzel, David R.	1-1
Dutra, John	17-1
Garthwaite, Alex	15-1
Gibbons, James F.	17-1
Gibbons, Jonathan J.	3-1
Gosling, James	7-1
Green, Stephen	20-1
Gupta, Amit	16-1
Hamilton, Graham	3-1
Hanko, James G.	8-1
Hanna, Steve	13-1
Hayes, Roger	4-1
Houston, Ann	20-1
Isaacs, Ellen A.	2-1
Johnson, Earl	12-1
Jordan, Mick	11-1
Kadansky, Miriam	13-1
Kendall, Sam	5-1
Kessler, Peter B.	3-1
Khalidi, Yousef A.	3-1, 9-1
Kougiouris, Panos	3-1
Kuhns, Robert J.	20-1
Lam, Monica S.	19-1
Landau, Susan	22-1
Lexau, Jon K.	21-1
Madany, Peter W.	3-1
Martin, Paul	10-1, 20-1
Matena, Vlada	9-1
Mitchell, Jim	iii, 3-1
Nelson, Michael N.	3-1
Northcutt, J. Duane	19-1
Pannoni, Robert L.	17-1
Papadopoulos, Greg	i

Index (continued)

Powell, Michael L.	3-1
Radia, Sanjay R.	3-1
Reich, Vicky	24-1
Rom, Raphael	16-1
Rosenthal, David S. H.	24-1
Rosenzweig, Phil	13-1
Sankar, Sriram	4-1
Savoia, Alberto	4-1
Schmidt, Brian K.	19-1
Schuba, Christoph	16-1
Sherriff, Ken	9-1
Simon, Doug	18-1
Sipusic, Michael J.	17-1
Smith, Randall B.	6-1, 17-1
Steele, Guy L., Jr.	14-1
Sutherland, Ivan E.	21-1
Sutherland, William R. "Bert"	ii, 17-1
Taivalsaari, Antero	18-1
Tang, John C.	2-1
Thadani, Moti	9-1
Uhler, Stephen A.	25-1
Ungar, David	6-1
Waldo, Jim	5-1
White, Derek	15-1
Woods, William A.	20-1
Wollrath, Ann	5-1
Wyant, Geoff	5-1
Yankelovich, Nicole	10-1