SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:667–703
Published online 11 March 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.654

Design, implementation, and performance of an automatic configuration service for distributed component systems

Fabio Kon[1,*,†], Jeferson Roberto Marques[1], Tomonori Yamane[2], Roy H. Campbell[3] and M. Dennis Mickunas[3]

[1]*Department of Computer Science, University of São Paulo, Brazil*
[2]*Energy and Public Infrastructure Systems Center, Mitsubishi Electric, Tokyo, Japan*
[3]*Department of Computer Science, University of Illinois at Urbana-Champaign, U.S.A.*

SUMMARY

Component technology promotes code reuse by enabling the construction of complex applications by assembling off-the-shelf components. However, components depend on certain characteristics of the environment in which they execute. They depend on other software components and on hardware resources. In existing component architectures, the application developer is left with the task of resolving those dependencies, i.e. making sure that each component has access to all the resources it needs and that all the required components are loaded. Nevertheless, according to encapsulation principles, developers should not be aware of the component internals. Thus, it may be difficult to find out what a component really needs. In complex systems, such as the ones found in modern distributed environments, this manual approach to dependency management can lead to disastrous results. Current systems rely heavily on manual configuration by users and system administrators. This is tolerable now, when users have to manage a few computers. But, in the near future, people will have to deal with thousands of computing devices and it will no longer be acceptable to require the user to configure each of them. This paper presents the results of our 6 year research (from 1998 to 2003) in the area of automatic configuration, describing an integrated architecture for managing dependencies in distributed component-based systems. The architecture supports automatic configuration and dynamic resource management in distributed heterogeneous environments. We describe a concrete implementation of this architecture, present experimental results, and compare our approach to other works in the area. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: automatic configuration; dynamic configuration; component-based systems; CORBA

*Correspondence to: Fabio Kon, Department of Computer Science, University of São Paulo, Brazil.
†E-mail: kon@ime.usp.br

Contract/grant sponsor: CNPq/NSF Collaborative Research Grant; contract/grant numbers: proc. 680037/99-3 (Brazil) and proc. 99-70139 (U.S.A.)

Received 28 October 2003
Revised 17 September 2004
Accepted 21 September 2004

1. INTRODUCTION

As computer systems are being applied to more and more aspects of personal and professional life, the quantity and complexity of software systems are increasing considerably. At the same time, the diversity in hardware architectures remains large and is likely to grow with the deployment of embedded systems, PDAs, sensors and actuators for ubiquitous computing, and other portable computing devices. All these platforms will coexist with personal computers, workstations, computing servers, and supercomputers. In this scenario, the construction of new systems and applications in an easy and reliable way can only be achieved through the composition of modular hardware and software.

Component technology has appeared as a powerful tool to confront this challenge. Recently developed component architectures support the construction of sophisticated systems by assembling together a collection of off-the-shelf software components with the help of visual tools or programmatic interfaces. Components are the unit of packaging, distribution, and deployment in the recent, sophisticated software systems. However, there is still little support for managing the dependencies among components. Components are created by different programmers, often working in different groups with different methodologies. It is hard to create robust and efficient systems if the dependencies between components are not well understood.

Mobile computing and active spaces are among the next revolutions in information technology. Mobile computing comprises systems as diverse as digital cellular telephony, road traffic information services, mobile videoconferencing, or simply Web browsing via wireless connections. Active spaces consist of physical spaces—such as offices, lecture and meeting rooms, homes, cars, hospitals, campuses, train stations, cities—that are augmented with computing devices integrated into the environment. The objective of these devices is to provide information to users of the space, helping them to perform activities they would not be able to perform otherwise, or helping them to perform conventional activities more easily.

Until recently, highly dynamic environments with mobile computers, active spaces, and ubiquitous multimedia were only present in science fiction stories or in the minds of visionary scientists like Mark Weiser [1]. But now, they are becoming a reality and one of the most important challenges they pose is the proper *management of dynamism*. Future computer systems must be able to configure themselves dynamically, adapting to the environment in which they are executing. Furthermore, they must be able to react to changes in the environment by dynamically reconfiguring themselves to keep functioning with good performance, irrespective of modifications in the environment.

The existing software infrastructure is not prepared to manage these highly dynamic environments properly. In fact, conventional operating systems already have a hard time managing static environments based on components. In the early days of the Windows NT operating system, even Microsoft researchers said that they were 'well aware of how difficult it is to install software on NT and get it to work' [2] because of the management problems that components induce. In shared UNIX systems such as Solaris, the administrator does not have a clean way of upgrading software that is being executed by users. When an executable is replaced, the running instances of the old executable crash.

Existing component-based systems face significant problems with reliability, administration, architectural organization, and configuration. The problem behind all these difficulties is the lack of a unified model for representing dependencies and mechanisms for dealing with these dependencies. Components depend on hardware resources (such as CPU, memory, and special devices) and software resources (such as other components, services, and the operating system). Not resolving these dependencies properly compromises system efficiency and reliability.

As systems become more complex and grow in scale, and as environments become more dynamic, the effects of the lack of proper dependence management become more dramatic. Therefore, we need an integrated approach in which operating systems, middleware, and applications collaborate to manage the components in complex systems, dealing with their hardware and software dependencies properly.

Software is in constant evolution and new component versions are released frequently. How can one run the most up-to-date components and make sure that they work together in harmony? This requires mechanisms for (1) code distribution over wide-area networks so we can push or pull new components as they become available and (2) safe dynamic reconfiguration so we can plug new components when desired.

In previous papers, we introduced a model for managing dependencies in distributed component systems [3,4] and described a reflective ORB that supports dynamic component loading in distributed environments [5–7]. This paper presents new experimental results and provides a comprehensive description of our work, describing the design, implementation, and performance of an integrated architecture that provides support for the following.

1. Automatic configuration of component-based applications.
2. Intelligent, dynamic placement of applications in the distributed system.
3. Dynamic resource management for distributed heterogeneous environments.
4. Component code distribution using push and pull methods.
5. Safe dynamic reconfiguration of distributed component systems.

The combination of these features in a single environment is an effective way to address the problems mentioned above. In such an environment, the user is freed from tiresome configuration activities such as manually downloading and installing different components for a single application and/or deciding in which machine to run each component of a distributed application.

To the best of our knowledge, no other system provides an integrated architecture which addresses all these functionalities. Fortunately, some of the ideas brought to the fore in the last decade by our research and that of our colleagues are finally reaching mainstream operating systems such as Linux, Windows, and MacOS and middleware systems such as JBoss. In Section 6 we discuss in more detail which functionalities have been incorporated into current commercial and open-source systems and which are yet to be assimilated.

1.1. Paper contents

Section 2 gives a general overview of our architecture for automatic configuration and dynamic resource management. Section 3 details the automatic configuration mechanisms, explaining the concepts of prerequisites (Section 3.1), component configurators (Section 3.2), and the Automatic Configuration Service (Section 3.3). Section 4 describes the Resource Management Service, addressing resource monitoring (Section 4.1) resource reservation (Section 4.2), application execution (Section 4.4), scalability (Section 4.5), and fault tolerance (Section 4.6).

Section 5 gives additional implementation details and presents experimental results. We then present related work (Section 6), future work (Section 7), and our conclusions (Section 8).

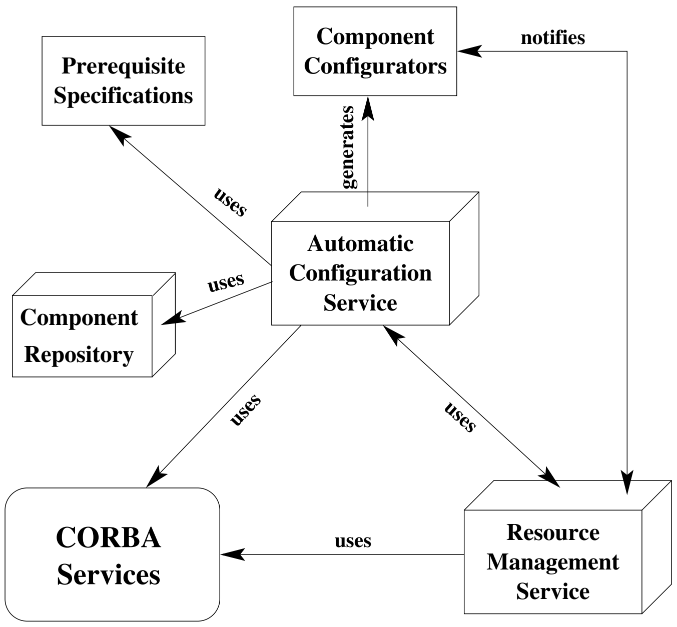

Figure 1. Architectural framework.

2. ARCHITECTURAL FRAMEWORK

To deal with the highly dynamic environments of the next decades, we propose an architectural framework divided in three parts. First, a mechanism for dependence representation lets developers specify component dependencies and write software that deals with these dependencies in customized ways. Second, a Resource Management Service is responsible for managing the hardware resources in the distributed system, exporting interfaces for inspecting, locating, and allocating resources in the distributed, heterogeneous system. Third, an Automatic Configuration Service is responsible for dynamically instantiating component-based applications by analyzing and resolving their component dependencies at runtime and taking into consideration the dynamic state of resources in the distributed system.

Figure 1 presents a schematic view of the major elements of our architecture. *Prerequisite specifications* reify static dependencies of components towards their environment while *component configurators* reify dynamic, runtime dependencies.

As we explain in Section 3, the automatic configuration process is based on the prerequisite specifications and constructs the component configurators. As the Automatic Configuration Service instantiates new components, fetched from the Component Repository, it uses the Resource Management Service to allocate resources for them. At execution time, changes in resource availability may trigger call-backs from the Resource Management Service to component configurators so that components can adapt to significant changes in the underlying environment.

As described in Section 4.4, when a client requests the execution of an application to the Resource Management Service, the latter finds the best location to execute the application and then uses the Automatic Configuration Service to load the application components.

The elements of the architecture are exported as CORBA services and their implementation relies on standard CORBA services such as Naming and Trading [8].

We have employed the architecture presented here to support a reliable, dynamically configurable Multimedia Distribution System. Readers interested in a detailed description of how our services were used in that particular application scenario should refer to [9]. In the following sections, we provide a more in-depth description of each of the elements of the architecture.

3. AUTOMATIC CONFIGURATION

Software systems are evolving more rapidly than ever before. Vendors release new versions of Web browsers, text editors, and operating systems once every few months. System administrators and users of personal computers spend an excessive amount of time and effort configuring their computer accounts, installing new programs, and, above all, struggling to make all the software work together[‡].

For almost a decade, the installation of new applications in operating systems has sometimes been partially automated by 'wizard' interfaces that direct the user through the installation process. However, it is common to face situations in which the installation cannot complete or in which it completes but the software package does not run properly because some of its (unspecified) requirements are not met. In other cases, after installing a new version of a system component or a new tool, applications that used to work before the update stop functioning. Often, applications cannot be cleanly removed; after executing special uninstall procedures, 'junk' libraries and files are left in the system. The application does not know if it can remove all the files it has installed because the system does not provide the clear mechanisms to specify which applications are using which libraries. These problems are normally referred to as 'DLL hell'.

In recent years, these problems have been somewhat alleviated by special package installers, such as dpkg and RPM for Linux and Windows Update for Windows, which manage static dependencies between system and application components.

To fully resolve this problem, we need a novel paradigm for installing, updating, and removing software from workstations and personal computers. We propose to automate the process of software maintenance with a mechanism we call *automatic configuration*. In our design of an automatic configuration service for modern computer environments, we focus on two key objectives:

1. network-centrism; and
2. a 'what you need is what you get' (WYNIWYG) model.

[‡]When even PhD students in Computer Science have trouble keeping their commodity personal computers functioning properly, one can notice that something is very wrong in the way that commercial software is built nowadays.

Network-centrism refers to a model in which all entities, users, software components, and devices exist in the network and are represented as distributed objects. Each entity has a network-wide identity, a network-wide profile, and dependencies on other network entities. When a particular service is configured, the entities that constitute that service are assembled dynamically. Users no longer need to keep and configure several different accounts, one for each device they use. In the network-centric model, a user has a single network-wide account, with a single network-wide profile that can be accessed from anywhere in the distributed system. The middleware is responsible for instantiating user environments dynamically according to the user's profile, role, and the underlying platform [10]. This model differs from other works that focus on automating the process of configuring the software installed on a specific node (e.g. [11]).

In contrast to existing operating systems, middleware, and applications where a large number of non-utilized modules are carried along with the standard installation, we advocate a *what you need is what you get* model, or *WYNIWYG*. In other words, the system should configure itself automatically and load a *minimal* set of components required for executing the user applications in the most efficient way. The components are downloaded from the network, so only a small subset of system services are needed to bootstrap a node.

In the automatic configuration model, system and application software are composed of network-centric components, i.e. components available for download from a *Component Repository* present in the network. Component code is encapsulated in dynamically loadable libraries (DLLs in Windows and shared objects in Unix), which enables dynamic linking.

Each application, system, or component[§] specifies everything that is required for it to work properly (both hardware and software requirements). This collection of requirements is called *prerequisite specifications* or, simply, *prerequisites*.

3.1. Prerequisites

The prerequisites for a particular inert component (stored on a local disk or on a network component repository) must specify any special requirements for properly loading, configuring, and executing that component. We consider three different kinds of information that can be contained in a list of prerequisites.

1. The nature of the hardware resources the component needs.
2. The capacity of the hardware resources it needs.
3. The software services (i.e. other components) it requires.

The first two items are used by the Resource Management Service to determine where, how, and when to execute the component. QoS-aware systems can use these data to enable proper admission control, resource negotiation, and resource reservation. The last item determines which auxiliary components must be loaded and in which kind of software environment they will execute.

[§]From now on, we use the term 'component' not only to refer to a piece of an application or system but also to refer to the entire application or system. This is consistent since, in our model, applications and systems are simply components that are made up of smaller components.

The first two items—reminiscent of the Job Control Languages of the mid-1960s—can be expressed by modern QoS specification languages such as QML [12] and QoS aspect languages [13], or by using a simpler, less sophisticated format such as SPDF (see Section 3.4.1). The third item is equivalent to the *require* clause in architectural description languages like Darwin [14] and module interconnection languages like the one used in Polylith [15].

The prerequisites are instrumental in implementing the WYNIWYG model as they let the system know what the exact requirements are for instantiating the components properly. If the prerequisites are specified correctly, the system not only loads all the necessary components to activate the user environment but also loads a minimal set of components required to achieve that.

We currently rely on the component programmer to specify component prerequisites. Mechanisms for automating the creation of prerequisite specifications and for verifying their correctness require further research and are beyond the scope of this paper. Another interesting topic for future research is the refinement of prerequisites specifications at runtime according to what the system can learn from the execution of components in a certain environment. This can be achieved by using QoS profiling tools such as QualProbes [16].

3.2. Component configurator

The explicit representation of *dynamic* dependencies is achieved through special objects attached to each relevant component at execution time. These objects are called *component configurators*; they are responsible for reifying the runtime dependencies for a certain component and for implementing policies to deal with events coming from other components.

While the Automatic Configuration Service parses the prerequisite specifications, fetches the required components from the Component Repository, and dynamically loads their code into the system runtime, it uses the information in the prerequisite specifications to create component configurators representing the runtime inter-component dependencies. Figure 2 depicts the dependencies that a component configurator reifies.

The dependencies of a component C are managed by a component configurator C^{c}. Each configurator C^{c} has a set of named *hooks* to which other configurators can be attached. These are the configurators for the components on which C depends; they are called *hooked components*. The components that depend on C are called *clients*; C^{c} also keeps a list of references to the clients' configurators. In general, every time one defines that a component C_1 depends on a component C_2, the system should perform two actions:

1. attach C_2^{c} to one of the hooks in C_1^{c}; and
2. add C_1^{c} to the list of clients in C_2^{c}.

Component configurators are also responsible for distributing events across the inter-dependent components. Examples of common events are the failure of a client and destruction, internal reconfiguration, or replacement of the implementation of a hooked component. The rationale is that such events affect all the dependent components. The component configurator is the place where programmers must insert the code to deal with these configuration-related events.

Component developers can program specialized versions of component configurators that are aware of the characteristics of specific components. These specialized configurators can, therefore, implement customized policies to deal with component dependencies in application-specific ways.

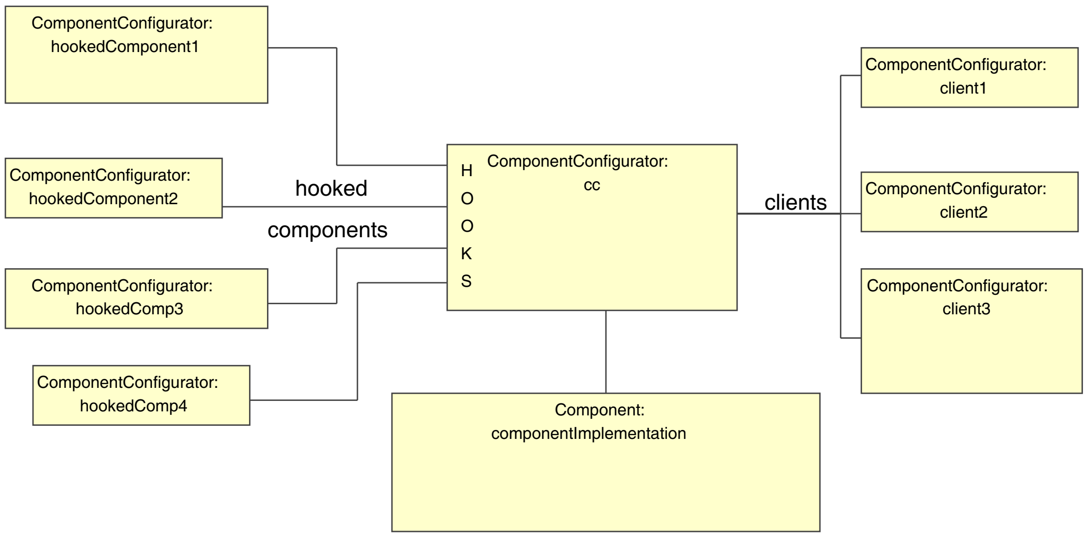
ComponentConfigurator: hookedComponent1
ComponentConfigurator: hookedComponent2
ComponentConfigurator: hookedComp3
ComponentConfigurator: hookedComp4
hooked components
HOOKS
ComponentConfigurator: cc
clients
ComponentConfigurator: client1
ComponentConfigurator: client2
ComponentConfigurator: client3
Component: componentImplementation

Figure 2. Reification of component dependencies.

As an example of customized component configurators, consider a QoS-sensitive video-on-demand client that reserves a portion of the local CPU for decoding a video stream. The application developer can program a special configurator that registers itself with the Resource Management Service. In this way, when the Resource Management Service detects a change in resource availability that would prevent the application from getting the desired level of service, it notifies the configurator (as shown in Figure 1). The configurator, with its customized knowledge about the application, sends a message to the video server requesting that the latter decrease the video frame rate. Then, with a lower frame rate, the client is able to process the video while the limited resource availability persists. When the resources go back to normal, another notification allows the video-on-demand configurator to re-establish the initial level of service.

The functionality provided by Component Configurators could also be implemented within the component itself. However, by separating application-related functionality from configuration-related functionality, the model opens more opportunities for reuse both of component configurators and application components. This clear separation of concerns (between what the application does and how it can be (re)configured) must be present in the design of any complex system that needs to be reconfigured at runtime.

To accomplish dynamic reconfiguration it is often necessary to transfer state from old components to the ones replacing them. The component developer must define methods to export and import the component state in a certain format so that a customized component configurator can extract the state from the old component, inject it into the new one and update its dependencies.

It is interesting to note that the Component Configurator mechanism is reminiscent of the dependence mechanism in the Smalltalk-80 programming language [17] and can be seen as a sophisticated implementation of the Observer pattern [18].

3.3. Automatic Configuration Service

As described above, automatic configuration enables the construction of network-centric systems following a WYNIWYG model. To experiment with these ideas, we developed an Automatic Configuration Service for the *2K* operating system [19].

Different application domains may have different ways of specifying the prerequisites of their application components. Therefore, rather than limiting the specification of prerequisites to a particular language, we built the Automatic Configuration Service as a framework in which different kinds of prerequisite descriptions can be utilized. To validate the framework, we designed the Simple Prerequisite Description Format (SPDF), a very simple, text-based format that allowed us to perform initial experiments. In the future, other more elaborated prerequisite formats including sophisticated QoS descriptions [12,13] can be plugged into the framework easily (e.g. in a related ongoing work, we are using QML [12], which is a powerful QoS Modeling Language).

In addition, depending upon the dynamic availability of resources and connectivity constraints, different algorithms for prerequisite resolution may be desired. For example, if a diskless PDA is connected to a network through a 2 Mbps wireless connection, it will be beneficial to download all the required components from a central repository each time they are needed. On the other hand, if a laptop computer with a large disk connects to the network via modem, it will probably be better to cache the components on the local disk and re-use them whenever is possible.

A similar mechanism was recently added to the Java platform: the Java Network Launching Protocol (JNLP) and its Java Web Start application [20]. However, it does not yet support the flexibility we provide by using a customizable framework and by decoupling the prerequisite parsers and resolvers.

Figure 3 shows how the architecture uses the two basic classes of the automatic configuration framework: *prerequisite parsers* and *prerequisite resolvers*. Administrators and developers can plug different concrete implementations of these classes to implement customized policies.

The automatic configuration process works as follows. First, the client sends a request for loading an application by passing, as parameters, the name of the application's 'master' component and a reference to a component repository (step 1 in Figure 3). The request is received by the prerequisite resolver, which fetches the component code and prerequisite specification from the given repository, or from a local cache, depending on the policy being used (step 2.1).

Next, the prerequisite resolver calls the prerequisite parser to process the prerequisite specification (step 2.2). As it scans the specification, the parser issues recursive calls to the prerequisite resolver to load the components on which the component being processed depends (step 2.3). This may trigger several iterations over steps 2.1, 2.2, and 2.3.

After all the dependencies of a given component are resolved, the parser issues a call to the Resource Manager to negotiate the allocation of the required resources (step 3). After all the application components are loaded, the service returns a reference to the new application to the client (step 4).

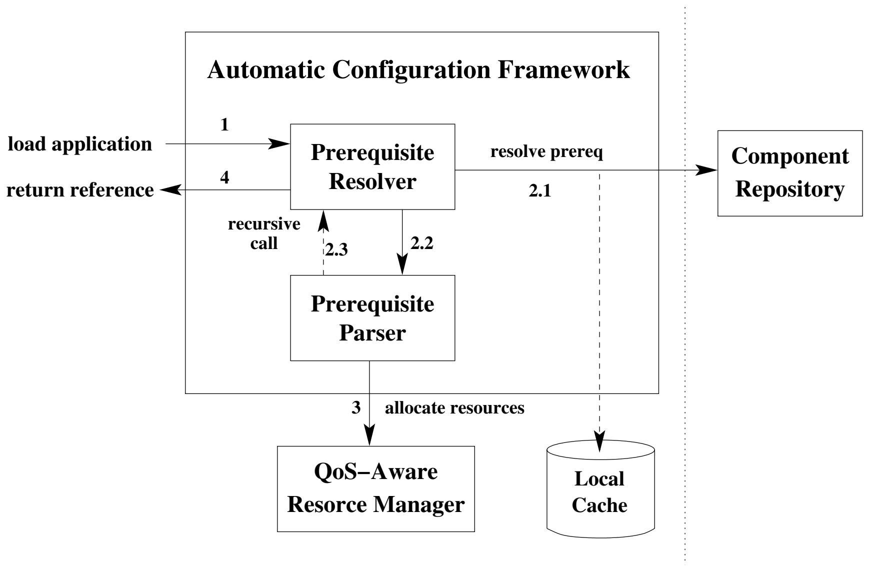

Figure 3. Automatic configuration framework.

3.4. A concrete implementation

To evaluate the framework, we created concrete implementations of the prerequisite parser and resolver. The prerequisite parser, called `SPDFParser`, processes SPDF specifications. The first prerequisite resolver, called `SimpleResolver`, uses CORBA to fetch components from the *2K* Component Repository. The second, called `CachingResolver`, is a subclass of `SimpleResolver` that caches the components on the local file system.

3.4.1. SPDF

We designed the Simple Prerequisite Description Format (SPDF) to serve as a proof of concept for our framework. Although we have also developed an XML-based SPDF format, we illustrate this paper with examples in the initial non-XML version of SPDF, which is easier to read by humans.

An SPDF specification is divided into two parts, the first is called hardware requirements and the second, software requirements. Figure 4 shows an example of an SPDF specification for a hypothetical Web browser. The first part specifies that this application was compiled for a SPARC machine running Solaris 2.7, that it requires at least 5 MB of RAM memory but that it functions optimally with 40 MB of memory, and that it requires 10% of a CPU with speed higher than 300 MHz.

The second part, software requirements, specifies that the Web browser requires four components (or services): a file system (to use as a local cache for Web pages), a TCP networking service

```
:hardware requirements
machine_type    SPARC
os_name         Solaris
os_version      2.7
min_ram         5MB
optimal_ram     40MB
cpu_speed       >300MHz
cpu_share       10%

:software requirements
FileSystem    CR:/sys/storage/DFS1.0 (optional)
TCPNetworking CR:/sys/networking/BSD-sockets
WindowManager CR:/sys/WinManagers/simpleWin
JVM           CR:/interp/Java/jvm1.5 (optional)
```

Figure 4. A simple prerequisite description.

(to fetch the Web pages), a window manager (to display the pages), and a Java Virtual Machine (to interpret Java Applets).

The components in the repository are organized in a hierarchical namespace. The first line in the software requirements section specifies that the component that implements the file system (or the proxy that interacts with the file system) can be located in the directory `/sys/storage/` of the Component Repository (`CR`) and that the component name is `DFS1.0`. It also states that the file system is an 'optional' component, which means that the Web browser can still function without a cache. Thus, if the Automatic Configuration Service is not able to load the file system component, it simply issues a warning message and continues its execution.

3.4.2. `SimpleResolver` *and* `CachingResolver`

The `SimpleResolver` fetches the component implementations and component prerequisite specifications from the *2K* Component Repository. It stores the component code in the local file system and dynamically links the components to the system runtime. As new components are loaded, they are attached to hooks in the component configurator of the parent component, i.e. the component that required it. In the Web browser example, the `SimpleResolver` would add hooks to the Web browser configurator, call them `FileSystem`, `TCPNetworking`, `WindowManager`, and `JVM`, and attach the respective component configurators to each of these hooks.

Resolvers can be extended using inheritance. For example, with very little work, we extended the `SimpleResolver` to create a `CachingResolver` that checks for the existence of the component on the local disk (cache) before fetching it from the remote repository.

3.5. Simplifying management

The Automatic Configuration Service simplifies management of user environments in distributed systems considerably. Whenever a new application is requested, the service downloads the most up-to-date version of its components from the network Component Repository and installs them locally. This provides several advantages including the following.

- It eliminates the need to upload components to the entire network each time a component is updated.
- It eliminates the need to keep track manually of which machines hold copies of each component because updates are automatic.
- It helps machines with limited resources, which no longer need to store all components locally.

3.6. Pushing component updates

The automatic configuration mechanism described here provides a pull-based approach for code updates and configuration. In other words, the service running in a certain network node takes the initiative to *pull* the code and configuration information from a Component Repository.

To support efficient and scalable management in large-scale systems, it may be desirable to allow system administrators to *push* code and configuration information into the network. Our architecture achieves this by using the concept of *mobile reconfiguration agents*, which we describe in detail elsewhere [21].

3.7. A J2EE-based implementation

When we started our research, there were no widely used, standard distributed component models. Thus, we were forced to develop our research with our own component model and most of our work was based on this proprietary model, which was built on top of CORBA. Later, during the development of the CORBA Component Model and of the Enterprise JavaBeans specification, we followed those works closely to make sure that our ideas could be deployed in those contexts.

To demonstrate that our Automatic Configuration Service could be implemented on the J2EE platform, a Masters student in our group carried out a research project on the topic. In this recent work [22], we extended the JBoss J2EE Application Server [23] with prerequisite parsers and resolvers for dynamically loading and configuring JBoss components recursively until all the prerequisites were met. In this implementation, the prerequisites are specified in the XML version of SPDF. It includes two major novelties in relation to our earlier work. First, the prerequisites can be specified as Boolean logic expressions; for example, a component developer can specify that component A depends either on component B or on both components C and D (i.e. $A \rightarrow B$ or (C and D)). Boolean expressions of any arbitrary complexity can be used. The second novelty is that the mechanism for locating and fetching remote components was extended. Dependencies can now be specified in three ways:

(1) as a remote URL pointing to the component bytecode in the component repository—in this case, the bytecode is downloaded and executed in the local machine;
(2) as a name in the distributed namespace of a naming service (e.g. the CORBA Naming Service or Java's RMI registry)—in this case the component is not downloaded, instead, the local components are configured to reference the components running in remote servers; and

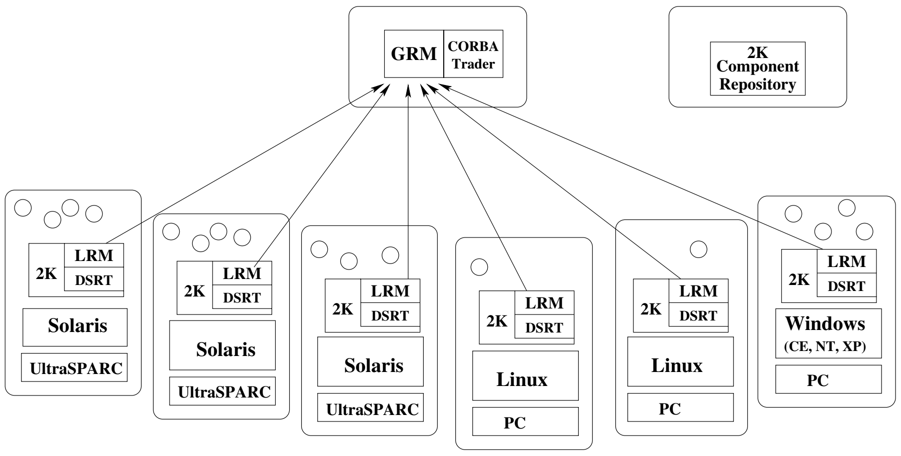

Figure 5. Resource management service in a single cluster.

(3) as a list of component attributes to be searched in a directory service (e.g. LDAP)—in this case, the developer specifies a set of characteristics that the required component must satisfy and the system queries the directory service to locate the best component meeting the requirements.

This implementation builds heavily on J2EE APIs such as JNDI [24] and JMX [25] since the goal was to show that our model can easily be implemented using standard middleware services.

4. RESOURCE MANAGEMENT SERVICE

The Resource Management Service [26] is organized as a collection of CORBA servers that are responsible for (1) maintaining information about the dynamic resource utilization in the distributed system, (2) locating the best candidate machine to execute a certain application or component based on its QoS prerequisites, and (3) allocating local resources for particular applications or components.

As shown in Figure 5, the Resource Management Service relies on Local Resource Managers (LRMs) present in each node of the distributed system. The LRM's task is to export the hardware resources of a particular node to the whole network. The distributed system is divided into clusters and each cluster is managed by a Global Resource Manager (GRM)[¶].

[¶]For historical reasons, we maintain the word 'global' in the name of the GRM; but one should note that once the system is extended to encompass multiple clusters, as described in Section 4.5, we have a global resource management service composed of multiple GRMs, one per cluster.

4.1. Resource monitoring

The LRMs running on the network nodes send updates of the state of their node resources (e.g. CPU and memory usage) to the GRM periodically. To increase portability, the LRM gets information about the underlying resources through the DSRT library [27], which provides a common interface to different operating systems like Solaris, Linux, and Windows.

The GRM implementation encompasses an instance of the standard OMG Object Trading Service [8]. A reference to the LRM of each machine in the cluster is stored in the GRM database as a trader 'service offer' and the state of its resources is stored as the offer's 'properties'.

To reduce network and GRM load, it is important to limit the frequency in which the LRMs send their updates to the GRM. Thus, although LRMs check the state of their local resources frequently, with a periodicity called p_1 (e.g. every 30 s), they only send this information to the GRM when (1) there were significant changes in resource utilization since the last update (e.g. a variation of more than 20% on the CPU load) or (2) a certain time (called p_2, or *keepalive* period) has passed since the last update was sent (e.g. 3 min). In addition, when a machine leaves the network, in the case of a shutdown or a voluntary disconnection of a mobile computer, the LRM unregisters itself from the GRM database. If the GRM does not receive an update from an LRM for a period twice as long as p_2, it assumes that the machine with that LRM is unreachable.

If desired, applications can further extend the service by defining new 'dynamic properties' in the trader's service offers. The value of these properties is evaluated at runtime by callbacks sent from the trader to objects distributed across the cluster nodes. These objects may return application-specific values such as number of clients using a given application, number of open connections, number of messages in incoming queues, etc.

4.2. Resource reservation

The LRMs are also responsible for performing QoS-aware admission control, resource negotiation, reservation, and scheduling of tasks on a single node. This is achieved with the help of the Dynamic Soft Real-Time Scheduler at the core of DSRT [27]. The scheduler runs as a user-level process on top of conventional operating systems (Solaris, Linux, or Windows) using the system's low-level, real-time API to provide QoS guarantees to applications with soft real-time requirements.

The LRM works as a CORBA wrapper for this scheduler, exporting its functionality to the distributed system. This scheduler can be used at any time by CORBA clients to request QoS guarantees on the availability of CPU and memory. For example, as explained in Section 3.3, a customized prerequisite parser may issue requests to reserve CPU and memory based on a component's hardware prerequisite specifications.

Whenever a node receives a request for running the components of a new application, the LRM selects a new process, which calls the DSRT API for allocating the required resources and hosting the application. To avoid the large latency of process creation, the LRM maintains a pool of pre-allocated seed processes.

DSRT version 2.0 does not provide support for managing network bandwidth but this support is expected in the future. Including this support is a difficult task but a prototype implementation based on RSVP-capable routers has been developed [28]. Once DSRT includes the facilities to inspect network utilization and reserve bandwidth, it will be straightforward to incorporate that into the *2K* Resource Management Service. All we have to do is to add a new property to the trader.

4.3. Dynamic adaptation

As we mentioned in Section 3.2, the Resource Management Service may issue call-backs to the application component configurators. These mechanisms can be used to adapt the application behavior in case of changes in resource availability. Thus, the Resource Management Service can be used not only to configure component-based applications dynamically but also to reconfigure them at runtime as directed by the application developers. Although we have not yet incorporated this feature into the C++ implementation of the *2K* Resource Management System, we have recently implemented a Java/CORBA framework for dynamic adaptation that demonstrates its usefulness [29]. Using this framework we have instrumented a distributed traffic information system [30] by implementing customized component configurators (as seen in Section 3.2). The system monitors resource utilization, detects the occurrence of complex events and notifies the customized configurators that reconfigure the system to adapt to the recent changes in the execution environment. Readers interested in more information about the issues involving dynamic adaptation should refer to [30].

4.4. Executing applications

Both the LRM and the GRM export an interface that lets clients execute applications (or components) in the distributed system. The GRM maintains an *approximate* view of the cluster resource utilization state and it uses this information as a *hint* for performing QoS-aware load distribution within its cluster. The maintenance of a precise view of the cluster state would be too expensive and would compromise scalability. The concept of hint has been used before in operating systems [31] and distributed file systems [32] and is used here to provide the benefits of resource awareness while maintaining a good degree of scalability.

As shown in Figure 6, when a user wishes to execute a new application, it sends an `execute_application` request to the local LRM—step (1) in the figure. This can be done either through a programmatic or graphic interface or using a shell. The LRM calls the DSRT to check whether the local machine has enough resources to execute the application comfortably, with a good quality of service (2). If not, the system will then try to locate a remote machine with enough resources to run the application comfortably; thus, it forwards the request to the GRM (3). The latter uses the information stored in its local trader to find a machine that would be the best candidate to execute that application and forwards the request, as a `oneway` message, to the LRM of that machine (4). The LRM of the latter machine tries to allocate the resources locally (5). If it is not possible to allocate the resources on that machine, it sends a `NACK` back to the GRM, which then looks for another candidate machine. If the GRM exhausts all the possibilities, it returns an empty offer to the client LRM.

When the system finally locates a machine with the proper resources, it creates a new process to host the application. Next, it uses the Automatic Configuration Service to fetch all the necessary components (i.e. the master component's dependencies) from the Component Repository (6, 7) and dynamically load them into that process (8), as described in Section 3.3. Finally, the LRM sends a CORBA `oneway ACK` message to the client LRM (9) containing a reference to the new component, which is then sent back to the user so he can access it (10).

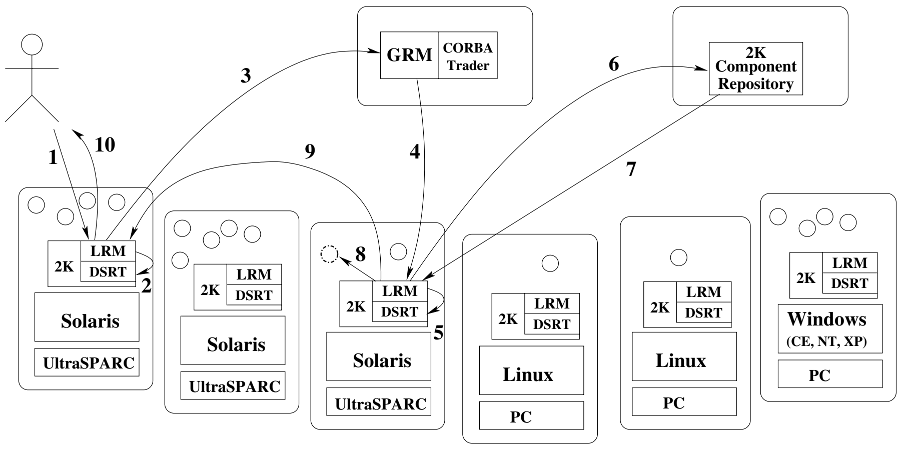

Figure 6. Resource reservation and component execution.

```
CosTrading::OfferSeq execute_application (
   in string categoryName,
   in string componentName,
   in string args,
   in CosTrading::PropertySeq QoS_spec,
   in CosTrading::Constraint platform_spec,
   in CosTrading::Preference prefs,
   in CosTrading::Lookup::SpecifiedProps
                                 return_props
);
```

Figure 7. Client request for application execution.

If the GRM could maintain a precise view of the cluster status–rather than an approximate view—it would not be necessary to try several machines in a row. However, the scalability of the algorithm depends on this uncertainty as it would be too expensive to maintain precise information about the cluster status. But note that the information update interval can be tuned so that, in most cases, the first candidate the GRM chooses is already capable of executing the application. As we will see in Section 5.2, it is possible to have a small update interval and still incur a low overhead.

4.4.1. Client request format

The format of the client request to the initial LRM is shown in Figure 7.

```
struct CpuReserve {
   long serviceClass;
   long period;
   long peakProcessingTime;
   long sustainableProcessingTime;
   long burstTolerance;
   float peakProcessingUtil;
};
```

Figure 8. CPU reservation specification.

`categoryName/componentName` specify which of the components in the *2K* Component Repository is the master component of the application to be executed and `args` contains the arguments that should be passed to it at startup time.

`QoS_spec` defines the quality of service required for this application. It is specified as a list of `<resourceName,resourceValue>` pairs. As an example, if the resource is the CPU, then the resource value should conform to the structure in Figure 8 (specified by the scheduler's CPU server [27]).

`platform_spec` are the criteria to select a cluster node, specified using the OMG Trader Constraint Language. For example, `(os_name == 'Linux') and (processor_util < 40)` will select a Linux machine whose CPU utilization is less than 40%.

`prefs` specifies the order of preference in case multiple machines satisfy the requirements. For example, `max(RAM_free)` specifies that the list of machines meeting the requirements must be sorted according to physical memory availability.

Finally, `return_props` specifies which properties (resource utilization information) should be included in the service offer that is returned. The returned value also includes a reference to the component configurator (see Section 3.2) of the new application.

4.5. Inter-cluster protocols and scalability

To enhance scalability across multiple clusters connected through the Internet, GRMs can be federated in a hierarchical way as depicted in Figure 9. Each GRM maintains an approximate view of the resource utilization state in each of its 'child' clusters. But, rather than maintaining information about each machine in its child clusters, the GRM maintains only a statistical representation (average and standard deviation) of the availability of resources for other clusters. Analogous to the intra-cluster update protocol described in Section 4.1, each GRM sends consolidated information about the state of the resources in its subtree to its parent GRM.

When a GRM receives a request for component execution, it still tries to execute it in the local cluster. If it cannot do that, it checks its local information about its child clusters, seeking a good candidate to execute the component in its subtree. But, if a request cannot be fulfilled in that subtree, the GRM forwards it to its parent GRM in the hierarchy. The parent GRM maintains an approximate

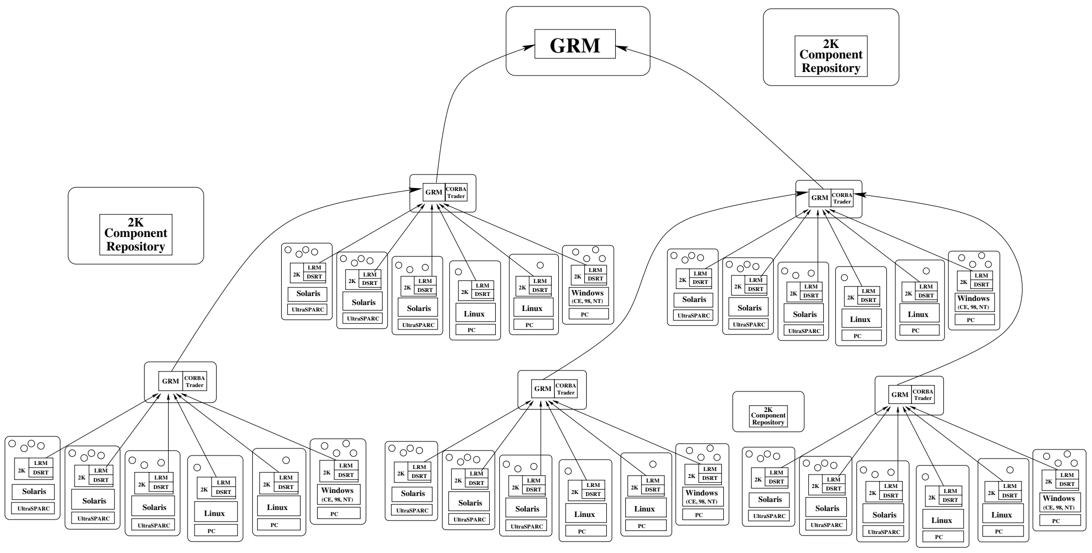

Figure 9. Cluster hierarchy.

view of the resource utilization in its child clusters and uses this information as a hint to locate a proper cluster to fulfill the client request.

4.6. Fault tolerance

To provide fault tolerance, the Resource Management Service architecture depends on a collection of replicated GRMs in each cluster. LRMs send their updates as a multicast message to all the GRMs in the cluster. Since strong consistency between the GRMs is not required, we can use an unreliable multicast mechanism. Client requests are sent to a single GRM and different clients may use different GRMs for load balancing.

Although we have designed and partially implemented the protocols and algorithms for fault tolerance mentioned in this subsection, we have not yet performed experiments to evaluate their effectiveness. We were not able to implement the multicast updates because the ORBs we were using did not support this facility. As new ORBs start to provide support for multicast [33], it will be easy to incorporate it into our implementation.

Another important fault-tolerance mechanism we did implement is the following. When an LRM detects that its GRM has failed, it stops sending status updates and tries to discover a new GRM by sending a multicast message to the cluster. So, if a GRM crashes and is replaced by a new one, or if it migrates to a new node, after some minutes, all the LRMs will detect it and the system structure will be reconstructed automatically.

5. EXPERIMENTAL RESULTS

In this section we describe a few details about the prototype implementations of the Automatic Configuration and Resource Management Services and present the results of the experiments we carried out.

The goal of this section is not to compare the performance of our system with other systems, since there are no other systems that provide the same functionality set that our system provides. The goal is to demonstrate that the architecture proposed in this paper can indeed be implemented in a real environment based on CORBA and that its performance, in terms of latency and CPU load, is so that it can be used in interactive, distributed environments, dynamically assembling applications in a fraction of a second. When we first started this research, we were often questioned whether such a dynamic approach for application composition would not make the system prohibitively slow. The experiments presented here show that this is not the case.

5.1. Automatic Configuration Service performance

The Automatic Configuration Service is implemented as a library that can be linked to any application. A program enhanced with this service becomes capable of fetching components from a remote Component Repository and dynamically loading and assembling them into its local address space. The library requires only 157 KB of memory on Solaris 7, which makes it possible to use it even on machines with limited resources such as a PalmPilot. In fact, we expect that services similar to this will be extensively used in future mobile systems to configure software automatically according to location and user requirements.

To evaluate the performance of the Automatic Configuration Service, we instrumented a test application [9] to measure the time for fetching, dynamic linking, and configuring its constituent components. We focused on two measurements: first, the time for loading a single component-based application as we vary the number of components in the application; and second, the time for loading components as we vary their size. In each case, we profiled the system to learn where it spends the time.

5.1.1. Loading multiple components

Figure 10 shows the total time for the service to load from one to eight components of 19.2 KB each. These experiments were carried out on two SPARC Ultra-60 machines running Solaris 7 and connected by a 100 Mbps Fast Ethernet network. The Component Repository was executed on one of the machines and the test application with the Automatic Configuration Service on the other. Each value is the arithmetic mean of five runs of the experiment. The vertical bars in the subsequent graphs and the numbers in parentheses in Table I represent the standard deviation. As the graph shows, the variation in execution times across different runs of the experiment was very small.

Table I shows, in more detail, how the service spends its time when loading a single 19.2 KB component. The current version of the Automatic Configuration Service fetches the prerequisites file from the remote Component Repository and saves it to the local disk. The same is done with the file containing the component code. Then, it uses the underlying operating system to perform the local dynamic linking of the component into the process runtime.

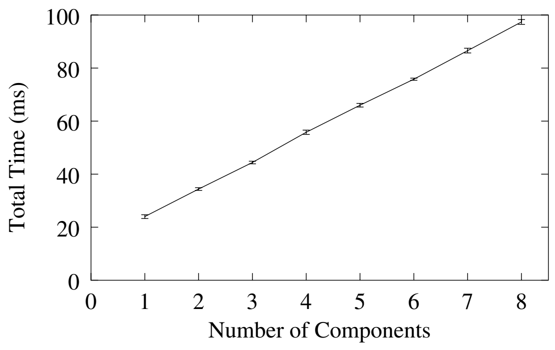

Figure 10. Automatic Configuration Service performance.

Table I. Discriminated times for loading a 19.2 KB component.

Action	Time (ms)	% of the total
Fetching prerequisites from Component Repository	2 (0)	8
Saving prerequisites to local disk	1 (0)	4
Fetching component from Component Repository	4 (0)	17
Saving component to local disk	1 (0)	4
Local dynamic linking	5 (0)	21
Autoconf protocol additional operations	11 (0.7)	46
Total	24 (0.7)	100

The table also shows the additional time spent by the service (row labeled 'Autoconf protocol additional operations') to detect if there are more components or prerequisite files to load, to parse the prerequisite file, and to reify dependencies. This overhead accounts for 46% of the total time required to load the component, which suggests that it would be desirable to improve this part of the service by optimizing the implementation of the `SimpleResolver` (see Section 3.4). We believe that an optimized version of the `SimpleResolver` could lead to improvements in the order of 20% for components of this size.

In the experiments described in this section, the component code and prerequisite files were cached in the memory of the machine executing the Component Repository. When the Component Repository program needs to read both files from its local disk, there is an additional overhead of approximately 20 ms.

5.1.2. *Components of different sizes*

To evaluate how the time for loading a single component varies with the component size, we created a program that generates components of different sizes. According to its command-line arguments,

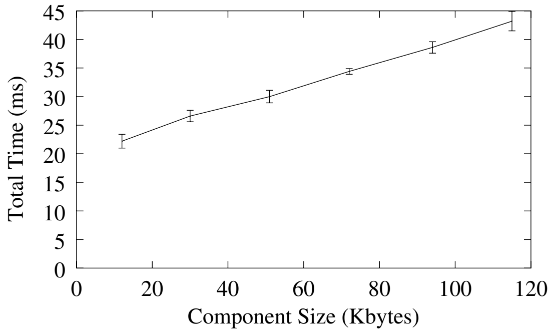

Figure 11. Times for loading components of different sizes.

this program generates C++ source code containing a given number of functions (which include code to perform simple arithmetic operations) and local and global variables. Using this program, we created components whose DLL sizes vary from 12 to 115 KB, which is the size range we chose to focus on. The rationale was that binary C++ components smaller than 12 KB cannot do much and would be useless; by contrast, with components larger than 115 KB, the components would be too coarse grained. However, one could use the service with components of several megabytes without any problem, although this would go against the philosophy of component-based programming. Figure 11 shows the time for the Automatic Configuration Service to load a single component as the component size increases.

Figure 12 shows the absolute times spent in each step of the process[‖]. Note that the time spent in the item labeled 'autoconf protocol' is approximately constant[**]. Hence, as the component size increases, its relative contribution to the total time decreases. This can be seen in Figure 13, which shows the same data in a different form. In this case, the figure shows the percentage of the total time spent in each of the steps of the process.

As the size of the component increases, the time for fetching the code from the remote repository to the local machine becomes the dominant factor. It is important to remember that these data were captured in a fast local network. If the access to the repository requires the use of a lower bandwidth connection, then this step would clearly be the most important with respect to performance. This suggests that deriving intelligent algorithms for component caching, taking component versions and user access patterns into consideration is an important topic for future research.

[‖]These steps are the same as those presented in Table I.

[**]This is expected since the messages processed in this step do not carry component code and therefore are not affected by the size of the component.

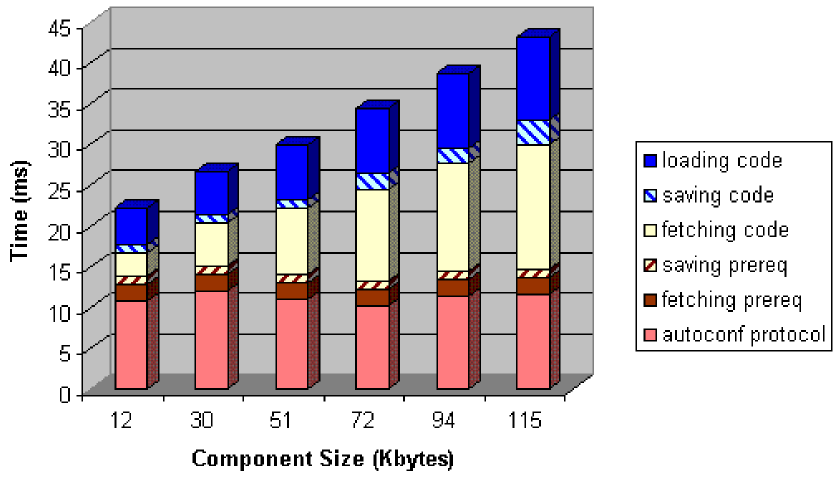

Figure 12. Discriminated times for loading components of different sizes.

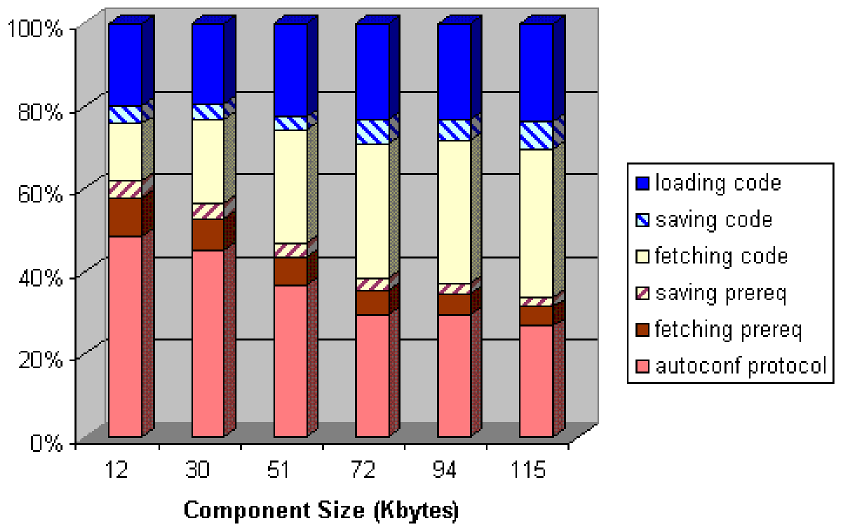

Figure 13. Discriminated relative times for loading components of different sizes.

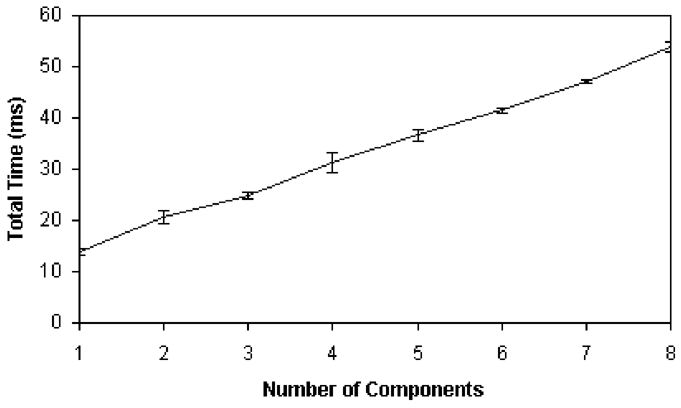

Figure 14. Times for loading applications from the local cache.

5.1.3. Component caching

As we saw above, the data collected from the first experiments showed the necessity of implementing a caching mechanism for components. We then extended the `SimpleResolver` to create a `CachingResolver`. The extension required less than 300 lines of highly commented C++ code and showed that the framework can be easily extended. The first time the component is executed in a machine, its code is fetched from the repository and stored in the local file system. In the subsequent times the same component is executed, its code is loaded directly from the local copy.

Figure 14 shows the times for loading applications made of one to eight components. In these experiments, the components were already in the local cache. The overhead of caching for loading components that are not in the cache was negligible.

Figures 15 and 16 show graphs similar to those in Figures 12 and 13 but now using a more optimized version of our code, including component caching.

Although there is still room for improvements and performance optimizations in the protocols used by the Automatic Configuration Service, the results presented here are very encouraging. They demonstrate that it is possible to carry out automatic configuration of a distributed component-based application within a tenth of a second, which is what we intended to show.

5.2. Resource Management Service performance

To evaluate the overhead imposed by the Resource Management Service in a cluster of computers, we carried out a series of controlled experiments in our laboratory. We intend to establish an intercontinental testbed composed of dozens of machines spread across multiple local area networks

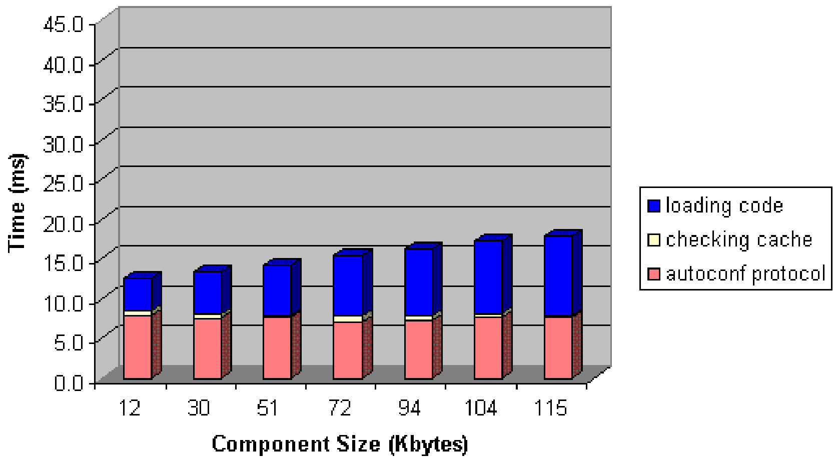

Figure 15. Discriminated times for loading components from the local cache.

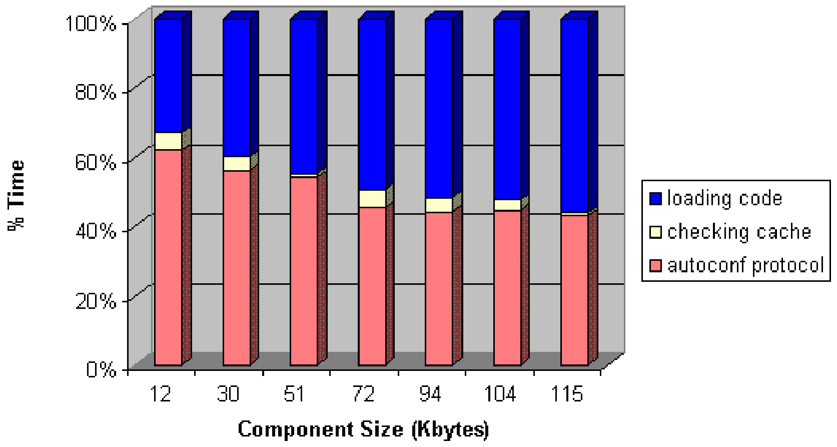

Figure 16. Discriminated relative times for loading components from the local cache.

in different countries[††]. However, as that has not yet been possible, we present experimental results obtained in two isolated machines connected by a 100 Mbps Ethernet, running the Linux 2.2 operating system. One of the machines (Pentium III 800 MHz, 192 MB RAM) executed the Name Server and the GRM, while the other (with two Pentium III 750 MHz, 768 MB RAM) executed multiple instances of the LRM to simulate the load generated by many clients, each one executing its own LRM. We believe that this configuration gives a good approximation of what one would have with multiple machines in a local area network since the load on the network and the GRM would be the same.

5.2.1. The LRM overhead on client machines

Under normal circumstances, the LRM should be configured to check the state of a machine's local resources every 30 or 60 s and to send updates to the GRM at least once every 2 or 3 min (as seen in Section 4.1). Nevertheless, to evaluate the overhead imposed on the client machines, we configured the LRM to perform these tasks much more frequently: checking the local state of resources every second and sending updates to the GRM at least every 2 s (i.e. $p_1 = 1$ and $p_2 = 2$). In this extreme case, CPU utilization by the LRM was always below 0.1%, while the average utilization in five experiments was 0.04%, which shows that the overhead on processor utilization caused by the LRM is negligible.

A potential problem we identified was that our prototype was developed using TAO [34], which is a powerful and complete CORBA ORB that requires a lot of memory to run. Thus, the LRM prototype requires 12 MB of memory in the client machine, which, in some cases, may be a significant requirement. This problem could be solved with the use of ORBs designed specifically for machines with scarce resources [6], requiring only a few kilobytes of memory.

5.2.2. The GRM overhead on the server

Typically, the GRM should be executed in a network's server machine and serve between 10 and 100 client machines. Figure 17 shows the percentage of CPU utilization by the GRM when the number of LRMs is 2, 8, 16, 32 and 64. In this experiment, we configured the LRMs with $p_1 = 30$ s and $p_2 = 60$ s. When we carried out the experiment, both the network and the processor were utilized exclusively for this task. To simulate fluctuations in the client's processor and memory utilization, we executed a program designed to cause large variations in resource utilization every 50 s, causing an increase in the number of messages from the LRM to the GRM. As shown in Figure 17, the overhead imposed by the GRM on the server processor is always very small, achieving 3.6% with 64 LRMs. This shows that the GRM machine can easily manage dozens of LRMs and still be used for other tasks without compromising the system's scalability. Each point in the graph is the arithmetic mean of five runs of the experiment; vertical bars represent the standard deviation, meaning that the majority of measured values falls in the interval represented by error bars above and below the mean.

The graph in Figure 18 also shows CPU utilization by the GRM. But, in this case, we fixed the number of LRMs to 32 and 64 and configured the parameters (p_1, p_2) to

[††]Another interesting approach, beyond the scope of this paper, would be to evaluate our protocols using wide-area network simulators since they allow simulations with thousands of nodes.

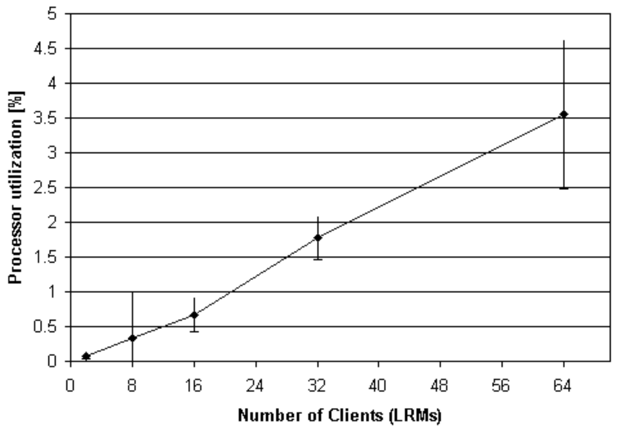

Figure 17. GRM overhead on the server.

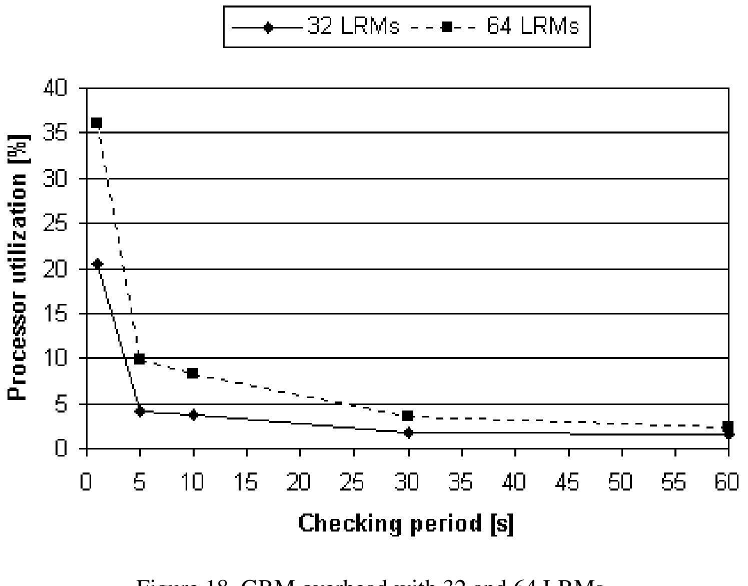

Figure 18. GRM overhead with 32 and 64 LRMs.

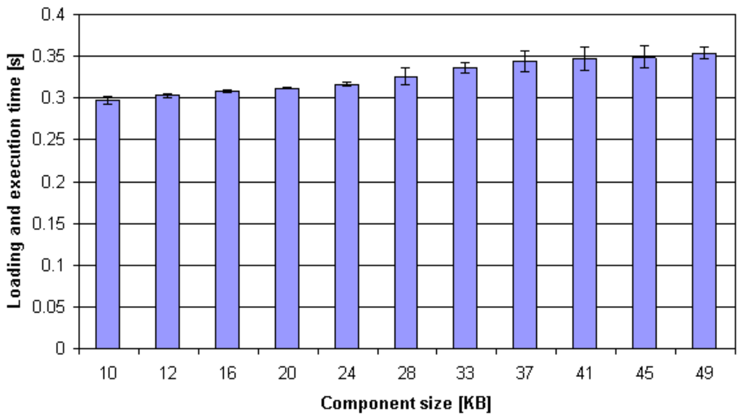

Figure 19. Time for fetching, loading, and executing components.

(1, 2), (5, 10), (10, 20), (30, 60) and (60,120) s. Even with 64 LRMs sending update messages at least every $P_2 = 2$ s (which is much above normal), the processor utilization did not go above 36%. When the period of local resource checks was $P_1 = 60$ s and the number of clients was 64 (which is a typical case), processor utilization was 2.3%, showing an excellent system scalability.

Figure 19 shows the time for fetching, loading, and executing components of various sizes. The times include all the 10 steps described in Section 4.4 and depicted in Figure 6. From this graph we see that the system is able to load and execute a 49 KB component in less than 354 ms, including all the overhead to select a candidate machine, reserve memory and CPU to guarantee QoS, fetch component code from the network repository, and dynamically load it.

In future work, we intend to carry out experiments concerning two other aspects of the Resource Management Service. First, we will evaluate the impact of having dozens of users in a cluster, requesting simultaneously the execution of hundreds of applications. Second, we intend to carry out a comprehensive evaluation of the performance of the inter-cluster protocols across inter-continental links.

6. RELATED WORK

Our work has been influenced by research carried out in many different fields of computer science. We present here previous and related work in UNIX tools for dependence management, component platforms, automatic configuration, Grid computing, software architecture and connectors, and semantic Web and standard description formats.

6.1. UNIX tools

UNIX tools such as `makedepend` and `nmake` provide support for managing compile-time dependencies among source-code and header files. They rely on an explicit description of dependencies in each source-code file; in C/C++, these dependencies are specified via the `#include` directive, for example. Unfortunately, in environments where applications are composed of multiple components that are assembled at runtime, it is not possible to use only the source code to specify inter-component dependencies. In this case, where a high degree of late binding is desirable, one needs an extra file specifying the component runtime dependencies in such a way that they can be resolved at application startup. Thus, besides writing source code to feed the compiler, the component programmer will also need to write an additional file to specify the dependencies.

Operating systems such as Linux, rely on a tool called GNU Autoconf [35] to detect the characteristics of the underlying hardware and software platforms and automate the configuration of new applications and services at compile time. Autoconf is used in a large number of free software packages that are designed to run on a wide variety of platforms. Obviously, the level of automatic configuration provided by such tools is very limited since it is restricted to compile-time configuration.

The Debian Linux operating system [36] includes `dpkg`, a package maintenance utility that manages the dependencies between the executables, libraries, documentation, and data files associated with Debian software packages. It maintains databases of dependencies that let users install, update, and delete packages without compromising consistency. Even when libraries are shared by multiple packages, `dpkg` is able to manage them consistently. Automatic configuration is restricted to package installation time and there is no support for automatic updates, i.e. they must be initiated manually by users. Similar tools are available for Red Hat Linux, the Red Hat Package Manager (RPM), and for MS-Windows, the Windows Update mechanism [37].

6.2. Component platforms

The OMG CORBA Component Model (CCM) specifies a standard framework for building, packaging, and deploying CORBA components [38]. Unlike our model, which focuses on prerequisites and dynamic dependencies, the CORBA Component Model concentrates on defining an XML vocabulary and an extension to the OMG IDL to support the specification of component packaging, customization, and configuration. The CCM *Software Package Descriptor* is reminiscent of our SPDF as it contains a description of package dependencies, i.e. a list of other packages or implementations that must be installed in the system for a certain package to work. *CORBA Component Descriptors*, on the other hand, describe the interfaces and event ports used and provided by a CORBA component.

We believe that our model and CCM complement each other and could be integrated. CCM provides a static description of component needs and interactions, while our model manages the runtime dynamics. Although CCM was already approved by OMG, publicly available ORBs do not yet support all its features. Once this happens, we intend to work towards the integration of the two models.

Among the major CORBA implementations, the one that most resembles our work is Orbix 2000 [39]. Its *Adaptive Runtime Architecture* lets users add functionality to the ORB by loading plug-ins dynamically. Whenever a request is sent to the ORB, it is processed by a chain of interceptors that can be configured in different ways using the loaded plug-ins. In that way, the ORB can be configured with interceptors that implement security, transactions, different transport protocols, etc. When the ORB loads a plug-in, it checks its version and dependence information. A centralized configuration repository specifies plug-in availability and configuration settings. Using this architecture it could be relatively easy to implement the functionality provided by our Automatic Configuration and Resource Management Services.

Enterprise JavaBeans [40] is a server-side technology for the development of component-based systems. It does not support the functionality for Automatic Configuration and Resource Management provided in our system. Nevertheless, it provides *deployment descriptors* that let one define, at deployment time, the configuration of individual components (Beans). Instead of recording the configuration information in a text format—like our SPDF and CORBA's XML formats—in the initial versions of EJB, the deployment descriptors were serialized Java classes; later, the specification moved towards an XML-based description. A deployment descriptor can customize the behavior of a Bean by setting environment properties as well as defining runtime attributes of its execution context, such as security, transactions, persistence, etc. [41].

Java Management Extensions (JMX) [25] is a set of Java APIs whose goal is to facilitate the management of complex Java applications. It allows external entities to browse, inspect, and modify internal attributes of Java components via programmatic or Web interfaces. The J2EE JBoss application server [23] is an example of a complex Java system that uses JMX for management. JBoss is based on a mickrokernel architecture whose core contains only a JMX spine capable of loading other components dynamically. JMX is used to manage a local namespace of JBoss components. It also provides standard services for (1) loading components from remote locations, (2) monitoring changes in the component attributes, (3) managing runtime *relations* among components (similarly to the management of runtime dependencies by our component configurator), and (4) creating timers that trigger periodic or scheduled notifications.

JMX does not address recursive loading and caching of remote components and does not provide support for resource management. Nevertheless, as we described in Section 3.7, it is a well-defined management specification that can be used as a solid basis for implementing the automatic configuration mechanisms we propose.

Barr and Eisenbach [42] present a JMX-based framework that allows Java programs to be dynamically upgraded with components published in a central repository. The framework is capable of testing the binary compatibility of Java components so that they can guarantee that the upgrade will not cause link-time errors.

6.3. Automatic configuration

Jini is a set of mechanisms for managing dynamic environments based on Java. It provides protocols to allow services to *join* a network and *discover* what services are available in this network. It also defines standard service interfaces for leasing, transactions, and events [43]. When a Jini server registers itself with the Jini lookup service, it stores a piece of Java byte code, called a proxy, in its entry in the lookup service. When a Jini-enabled client uses the lookup service to locate the server, it receives, as a reply, a `ServiceItem`, which is composed of a service ID, the code for the proxy, and a set of service attributes. The proxy is then linked into the client address space and is responsible for communication with the server. In this way, the communication between the client and the server can be customized, and optimized protocols can be adopted.

This Jini mechanism for proxy distribution can be achieved in a CORBA environment by using the Automatic Configuration Service in conjunction with a reflective ORB such as *dynamicTAO* [5]. The Automatic Configuration Service would fetch the proxy code and dynamically link it, while *dynamicTAO* would use the TAO pluggable protocols framework [44] to plug the proxy code into the TAO framework. Jini is normally limited to small-scale networks and it does not address the management of component-based applications and inter-component dependence. Due to the large memory requirements imposed by Java/Jini and the limitations of lean Java versions such as Personal Java [45], this is not yet a viable alternative for many embedded devices and PDAs.

Unlike our system, which performs automatic configuration at application startup, Software Dock [11] is a research project that focuses on configuration at application deployment time. Software Dock provides a powerful and comprehensive framework for application installation, reconfiguration, adaptation, and removal. Similarly to our work, it is based on a well-defined description format (Deployment Software Description, or DSD) [46] for representing application prerequisites and client site characteristics. It comprises three major elements: the *release dock* represents software producers and is equivalent to our component repository, the *field dock* resides in the end-user machine and is equivalent to our local resource manager, and the *wide-area event system* (not present in our system), which is responsible for exchanging information between software producers and consumers (such as new release announcements). In addition, software *agents* play the role of our prerequisite resolvers enabling the use of customized mechanisms for deploying each application at each different site. Software Dock does not address runtime reconfiguration, distributed resource discovery, admission control, and resource reservation. While Software Dock aims at deploying software in a certain machine, our approach focuses on configuring a user environment irrespective of which machine the user is logged to. In summary, there are many similarities between the two systems but their goals are slightly different.

CoDelivery [47] is a Java-based environment for distribution of reusable software components, which is similar to our Automatic Configuration Service (described in Section 3.3). However, while our CORBA-based system supports components written in various programming languages, CoDelivery is limited to Java. CoDelivery uses Java RMI servers called containers as its component repositories and is able to assemble applications based on multiple components dynamically, fetching them from various repositories. It has no support for resource management such as admission control and resource reservation.

Recent work in the Java platform defined the Java Network Launching Protocol (JNLP). Using this protocol, an application such as Sun's Java Web Start [20] is able to assemble a component-based application dynamically when the user requests the execution of the application. This protocol covers many aspects of our research, mainly the idea of fetching the components from the network dynamically as we need them and using a static description of component dependencies to load all the components until the application is ready for execution. However, it does not provide the same level of flexibility for prerequisite parsing and resolution (see Section 3.3) and it does not provide any support for resource management such as admission control, resource reservation, etc., which is provided by our Resource Management Service.

6.4. Grid systems

The Globus project [48] provides a 'computational grid' [49] integrating heterogeneous distributed resources in a single wide-area system. It supports scalable resource management based on a hierarchy of resource managers similar to the ones we propose. Globus defines an extensible Resource Specification Language (RSL) that is similar to our SPDF (described in Section 3.4.1). RSL [50] allows Globus users to specify the executables they want to run as well as their resource requirements and environment characteristics. RSL could be integrated in our system by plugging an `RSLParser` into our Automatic Configuration framework. A fundamental difference between Globus and our work is that we focus on *component-based* applications that are dynamically configured by assembling components fetched from a network repository. In Globus, on the other hand, the user specifies the application to be executed by giving the name of a single executable on the target host file system or by giving a URL from which a monolithic executable can be fetched.

Legion [51] is the system that shares most similarities with *2K* as it also builds on a distributed, reflective object model. However, the Legion researchers focused on developing a new object model from scratch. Legion applications must be built using Legion-specific libraries, compiler, and run-time system (the Legion's ORB). In contrast, we focused on leveraging CORBA technology to build an integrated architecture that could provide the same functionality as Legion, while still preserving complete interoperability with other CORBA systems. In addition, our work emphasizes automatic configuration and dependence management, which are not addressed by Legion.

6.5. Software architecture and connectors

Systems based on architectural connectors like UniCon [52] and ArchStudio [53] and systems based on software buses like Polylith [15] separate issues concerning component functional behavior from component interaction. Our model builds on that work and goes one step further by separating inter-component communication from inter-component dependence. Connectors and software buses require that applications be programmed to a particular communication paradigm. Unlike previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. It can be used in conjunction with connectors, buses, local method invocation, CORBA, Java RMI, and other methods. In fact, we were able to use this additional freedom brought by our model in the implementation of *dynamicTAO* [5]. In the *dynamicTAO* reflective ORB [7], the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms. The ORB base-level remained untouched, using regular C++ method calls for communication, while our model for dependence management was used to build a meta-level for supporting dynamic reconfiguration of the base-level components.

Research in connector-based systems assumes that communication and dependence are often intimately related, which is true. But, in many cases, the distinction between inter-component dependence and inter-component communication is beneficial. For example, the quality of service provided by a multimedia application is greatly influenced by the mechanisms utilized by underlying services such as virtual memory, scheduling, and memory allocation. The interaction between the application and these services is often implicit, i.e. no direct communication (e.g. library or system calls) takes place. Yet, if the system infrastructure allows developers to establish and manipulate dependence relationships between the application and these services, the application can be notified of substantial changes in the state and configuration of the services that may affect its performance.

Research in software architecture [54] and dynamic configuration [55] typically focuses on the architecture of individual applications. It does not deal with dependencies of application components towards system components, other applications, or services available in the distributed environment. Our approach differs from them in the sense that, for each component, we specify its dependencies on all the different kinds of environment components and we maintain and use these dynamic dependencies at runtime. Approaches based on software architecture typically rely on global, centralized knowledge of application architecture. In contrast, our method is more decentralized and focuses on more direct component dependencies. We believe that, rather than conflicting with the software architecture approach, our vision complements it by reasoning about all the dependencies that may affect reliability, performance, and quality of service.

6.6. Semantic Web and standard description formats

Salutation [56] provides a schema for describing common entities (e.g. printers, faxes, etc.), for defining classes of service (e.g. print), and defines a standard vocabulary of attributes and values for each class. The emerging technologies of the 'Semantic Web' permit the exchange of descriptions and relationships among entities. In particular, *description logics* together with automated reasoning systems (such as FaCT [57]) allow the description of relationships between classes by specifying properties or attributes, and support powerful reasoning about those relationships. Increasingly such reasoning systems (such as Protégé-2000 [58]) are being adapted to the DARPA Agent Markup Language (DAML) and Ontology Interchange Language (OIL) [59]. DAML+OIL extend the W3C's Resource Description Framework (RDF) model and the RDF Schema (RDFS), providing a common format for exporting knowledge bases and schema.

The Distributed Management Task Force (DMTF) [60] is an industry consortium whose goal is to develop standards for management of desktop and Web-based computer systems. It defined a standard Desktop Management Interface (DMI) [61] for managing and tracking components in a desktop, mobile or server PC and a Web-Based Enterprise Management standard (WBEM) for managing Web applications. Both standards are based on the Common Information Model (CIM), an object-oriented schema for representing information about entities such as systems [62] and applications [63] in a standardized vendor- and platform-neutral format.

Future work in automatic configuration and deployment of distributed component-based systems must leverage the new Semantic Web technologies and DMTF standards to develop enhanced mechanisms for reasoning about component prerequisites and user needs, making the automatic configuration process more 'intelligent' and technology-neutral.

6.7. Comparative evaluation

Finally, Table II presents a feature checklist comparing our system (labeled as *2K*) to some of the related work in the field. The rows in Table II specify whether each of the systems offers support for loading component-based applications, independence of operating system and language, dynamic discovery of resources in the distributed system, QoS-aware admission control and resource reservation, and whether the system source code is available and open.

Among the systems that support components, we can classify them according to the time in which components are fetched and loaded, whether components are cached and if they provide any support for runtime reconfiguration (Table III).

The final solution to the problem of supporting reliable automatic (re)configuration may reside in the combination of our model with recent work in software architecture, dynamic (re)configuration, and standard description formats. This is certainly an important open research problem whose solution could lead to a unified and standardized model for automatic configuration in next generation computing systems.

Table II. System comparisons.

	System											
Feature	*2K*	JNLP	EJB/ JMX	Jini	CCM	dpkg & RPM	Windows Update	makedepend	Legion	Globus	Software Dock	CoDelivery
components	yes	yes	yes	no	yes	yes	yes	no	no	no	yes	yes
OS independence	yes	yes	yes	yes	yes	no	no	yes	yes	yes	yes	yes
language independence	yes	no	no	no	yes	yes	yes	no	yes	yes	yes	no
resource discovery	yes	no	no	yes	no	no	no	no	yes	yes	no	no
admission control	yes	no	no	no	no	no	no	no	no	no	no	no
resource reservation	yes	no	no	no	no	no	no	no	no	no	no	no
open source	yes	yes	yes	no	yes	yes	no	yes	yes	yes	yes	no

Table III. Comparison of systems supporting component-based applications.

	Feature		
System	Component loading performed at	Component caching	Enables runtime reconfiguration
2K	application startup	yes	yes
JNLP	application startup	yes	no
EJB/JMX	server startup	not specified	yes
CCM	server startup	not specified	yes
Linux dpkg & RPM	application installation	yes	no
Windows Update	application installation	yes	no
CoDelivery	application startup	yes	no
Software Dock	application deployment	yes	no

7. FUTURE WORK

Our group at the University of Illinois has been working, under the *2K* and Gaia projects [19,64], on QoS compilation techniques, addressing the problem of translating application-level QoS specifications to component-level QoS specifications, and then to resource-level component prerequisites [65,66]. Gaia is focusing on the dynamic configuration of component-based applications and systems for active spaces in ubiquitous computing [67].

In the University of São Paulo, our group is developing mechanisms for automatic runtime adaptations in distributed systems and running new experiments to evaluate the system's performance.

In the previous sections, we alluded to some other important topics for future work, namely, (1) automatic creation and refinement of prerequisite specifications, (2) intelligent algorithms for component caching taking versions into consideration, and (3) the integration of our dependence model with recent research in software architecture and standard description formats.

8. CONCLUSIONS

Component technologies will play a fundamental role in the next generation computer systems as the complexity of software and the diversity and pervasiveness of computing devices increase. However, component technologies must offer mechanisms for automatic management of inter-component dependencies and component-to-resource dependencies. Otherwise, the development of component-based systems will continue to be difficult and frequently lead to unreliable and non-robust systems.

Future ubiquitous computing environments will be composed of thousands of devices running millions of software components. Current systems rely heavily on manual configuration but with a system composed of millions of components this will no longer be possible. There are only two ways out of this situation: static configuration or dynamic, automatic configuration.

Since future environments tend to be more and more dynamic, it seems that automatic configuration is the only viable solution.

Although there are still a number of open problems for future research, we believe that this paper provides an important contribution to the area by presenting an integrated object-oriented architecture for automatic configuration and dynamic resource management in distributed component systems. Performance evaluation demonstrated that our system is able to instantiate applications dynamically by assembling network components in less than a tenth of a second.

In this paper, we have described research carried out by our group over the last 6 years. In recent years, some of the concepts presented here have been incorporated into commercial and open source systems. However, there is much progress to be made before our systems meet the requirements for effective automatic configuration and resource management imposed by next generation software environments.

Availability

Source code and more information about the Automatic Configuration Service and the Resource Management Service can be found at our site at the University of São Paulo (http://gsd.ime.usp.br/ResourceManagement) and at the *2K* site at the University of Illinois (http://choices.cs.uiuc.edu/2k).

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the help and wise input provided by the *2K* team at the University of Illinois during the development of this work. Herbert Yutaka Watanabe was responsible for the J2EE implementation of the Automatic Configuration Service. We are also grateful for the valuable feedback provided by the anonymous reviewers of this paper.

This research was supported by the CNPq/NSF Collaborative Research Grant proc. 680037/99-3 (Brazil) and proc. 99-70139 (U.S.A.).

REFERENCES

1. Weiser M. The computer for the 21st century. *Scientific American* 1992; **265**(3):94–104.
2. Milojicic D, Bolosky B, Black D, Kaashoek F, Liedtke J, Mogul J, Wilkes J. Operating systems—now and in the future. *IEEE Concurrency* 1999; **7**(1):12–21.
3. Kon F, Campbell RH. Supporting automatic configuration of component-based distributed systems. *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, San Diego, CA, May 1999; 175–187.
4. Kon F, Yamane T, Hess C, Campbell R, Mickunas MD. Dynamic resource management and automatic configuration of distributed component systems. *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, TX, February 2001; 15–30.
5. Kon F, Román M, Liu P, Mao J, Yamane T, Magalhães LC, Campbell RH. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000 (*Lecture Notes in Computer Science*, vol. 1795). Springer: Berlin, 2000; 121–143.
6. Román M, Kon F, Campbell R. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online* 2001; **2**(5). Available at: http://dsonline.computer.org/0105/features/rom0105_print.htm.
7. Kon F, Costa F, Campbell R, Blair G. The case for reflective middleware. *Communications of the ACM* 2002; **45**(6):33–38.
8. OMG. CORBAservices: Common object services specification. *OMG Document 98-12-09*. Object Management Group: Framingham, MA, 1998.
9. Kon F, Campbell RH, Nahrstedt K. Using dynamic configuration to manage a scalable multimedia distribution system. *Computer Communications Journal* (*Special Issue on QoS-Sensitive Distributed Systems and Applications*) 2001; **24**:105–123.

10. Carvalho D, Kon F, Ballesteros F, Román M, Campbell R, Mickunas D. Management of execution environments in 2K. *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'2000)*, July 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 479–485.
11. Hall RS, Heimbigner D, Wolf AL. A cooperative approach to support software deployment using the software dock. *Proceedings of the ACM International Conference on Software Engineering*, Los Angeles, May 1999; 174–183.
12. Frølund S, Koistinen J. Quality of service aware distributed object systems. *Proceedings of the 5th USENIX Conference on Object-Oriented Technology and Systems (COOTS'99)*, San Diego, May 1999; 69–83.
13. Loyall JP, Bakken DE, Schantz RE, Zinky JA, Karr DA, Vanegas R, Anderson KR. QoS aspect languages and their runtime integration. *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, Pittsburgh, PA, May 1998.
14. Magee J, Dulay N, Eisenbach S, Kramer J. Specifying distributed software architectures. *Proceedings of the 5th European Software Engineering Conference*. Springer: Berlin, 1995; 137–153.
15. Purtilo J. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems* 1994; **16**(1):151–174.
16. Li B, Nahrstedt K. QualProbes: Middleware QoS profiling services for configuring adaptive applications. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000 (*Lecture Notes in Computer Science*, vol. 1795). Springer: Berlin, 2000; 256–272.
17. Goldberg A, Robson D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley: Reading, MA, 1983.
18. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
19. Kon F, Campbell RH, Mickunas MD, Nahrstedt K, Ballesteros FJ. 2K: A distributed operating system for dynamic heterogeneous environments. *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, Pittsburgh, August 2000; 201–208.
20. Sun Microsystems. Java Web Start. Home page, 2002. http://java.sun.com/products/javawebstart [18 January 2005].
21. Kon F, Gill B, Anand M, Campbell RH, Mickunas MD. Secure dynamic reconfiguration of scalable CORBA systems with mobile agents. *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA'2000)*, Zurich, September 2000; 86–98.
22. Watanabe HY. Automatic configuration in the enterprise javabeans platform (in portuguese). *Master's Thesis*, Department of Computer Science, University of Sao Paulo, February 2003.
23. Fleury M, Reverbel F. The JBoss extensible server. *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, June 2003 (*Lecture Notes in Computer Science*, vol. 2672). Springer: Berlin, 2003; 344–373.
24. Sun Microsystems. Java Naming and Directory Interface. Home page, 2003. http://java.sun.com/products/jndi [18 January 2005].
25. Lindfors J, Fleury M. *JMX: Managing J2EE with Java Management Extensions*. Sams Publishing: Indianapolis, IN, 2002.
26. Yamane T. The design and implementation of the 2K resource management service. *Master's Thesis*, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2000.
27. Nahrstedt K, Chu HH, Narayan S. QoS-aware resource management for distributed multimedia applications. *Journal of High-Speed Networking* (*Special Issue on Multimedia Networking*) 1998; **7**:227–255.
28. Viswanathan AK. Design and evaluation of a cpu-aware communication broker for rsvp-based network. *Master's Thesis*, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.
29. da Silva e Silva FJ, Endler M, Kon F. A framework for building adaptive distributed applications. *Middleware'2003 Companion Proceedings, Workshop on Reflective and Adaptive Middleware Systems*, Rio de Janeiro, Brazil, June 2003. ACM/IFIP/USENIX, 2003; 110–114.
30. da Silva e Silva FJ, Endler M, Kon F. Developing adaptive distributed applications: A framework overview and experimental results. *Proceedings of the International Conference on Distributed Objects and Applications (DOA'03)*, Catánia, Italy, November 2003 (*Lecture Notes in Computer Science*, vol. 2888). Springer: Berlin, 2003; 1275–1291.
31. Patterson RH, Gibson GA, Ginting E, Stodolsky D, Zelenka J. Informed prefetching and caching. *SOSP15*, Copper Mountain, CO, December 1995. ACM: New York, 1995; 79–95.
32. Sarkar P, Hartman J. Efficient cooperative caching using hints. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.
33. OMG. Unreliable multicast final adopted specification. *OMG Document ptc/01-11-08*. Object Management Group: Framingham, MA, November 2001.
34. Schmidt DC, Cleeland C. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine Special Issue on Design Patterns* 1999; **37**(4):54–63.
35. MacKenzie D, Elliston B. *Autoconf—Creating Automatic Configuration Scripts*. Free Software Foundation: Boston, MA, 1998.
36. The Debian Project. Debian linux operating system. Home page, 2000. http://www.debian.org [18 January 2005].

37. Keizer G. Make a date with windows update—helping windows help itself, January 2002. http://www.winplanet.com/article/1822-2285.htm [18 January 2005].
38. OMG. CORBA components. *OMG Document orbos/99-07-01*. Object Management Group: Framingham, MA, 1999.
39. IONA Technologies. *Orbix 2000*, 2000. White paper. Available at: http://www.iona.com.
40. Thomas A. Enterprise JavaBeans technology: Server component model for the Java platform. *Patricia Seybold Group*, December 1998. Available at: http://java.sun.com/products/ejb/white.
41. Monson-Haefel R, Burke B, Labourey S. *Enterprise JavaBeans* (4th edn). O'Reilly: Sebastopol, CA, 2004.
42. Barr M, Eisenbach S. Safe upgrading without restarting. *IEEE Conference on Software Maintenance (ICSM 2003)*, September 2003; 129–137.
43. Arnold K, O'Sullivan B, Scheifler RW, Waldo J, Wollrath A. *The Jini Specification*. Addison-Wesley: Reading, MA, 1999.
44. O'Ryan C, Kuhns F, Schmidt DC, Othman O, Parsons J. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.
45. Sun Microsystems. Personal Java Application Environment. Home page, 2002. http://java.sun.com/products/personaljava [18 January 2005].
46. Hall RS. Agent-based software configuration and deployment. *PhD Thesis*, Department of Computer Science, University of Colorado, Boulder, April 1999.
47. da Silveira GE, Meira SL. CoDelivery: An environment for distribution of reusable components. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*, St.-Malo, France, June 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.
48. Foster I, Kesselman C. The Globus project: A status report. *Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998; 4–18.
49. Foster I, Kesselman C (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Francisco, CA, 1999.
50. The Globus Project. *Globus Resource Specification Language RSL v1.0*, 2000. Available at: http://www.globus.org/gram.
51. Grimshaw AS, Wulf WA and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM* 1997; **40**(1).
52. Shaw M, DeLine R, Zelesnik G. Abstractions and implementations for architectural connections. *Proceedings of the 3rd International Conference on Configurable Distributed Systems (CDS'96)*, Annapolis, MD, May 1996.
53. Oreizy P, Taylor RN. On the role of software architectures in runtime system reconfiguration. *Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, MD, May 1998.
54. Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall: Englewood Cliffs, NJ, 1996.
55. Purtilo J, Cole R, Schlichting R (eds.). *Fourth International Conference on Configurable Distributed Systems*, May 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998.
56. Salutation Consortium. Salutation, 2002. http://www.salutation.org [18 January 2005].
57. Horrocks I. The FaCT system. *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, 1998; 307–312.
58. Noy NF, Sintek M, Decker S, Crubezy M, Fergerson RW, Musen MA. Creating semantic Web contents with Protéé-2000. *IEEE Intelligent Systems* 2001; **16**(2):60–71.
59. Connolly D, van Harmelen F, Horrocks I, McGuinness D, Patel-Schneider PF, Stein LA. *Annotated DAML+OIL Ontology Markup*. W3C Note, December 2001. http://www.w3.org/TR/daml+oil-walkthru [18 January 2005].
60. DMTF. Distributed Management Task Force. Home page, 2003. http://www.dmtf.org/ [18 January 2005].
61. DMTF. Desktop management interface specification, version 2.0.1s. *Technical Report DSP0005*, Distributed Management Task Force, January 2003.
62. DMTF. Cim v. 2.7 system model white paper. *Technical Report DSP0150*, Distributed Management Task Force, June 2003.
63. DMTF. Understanding the application management model. *Technical Report DSP0140*, Distributed Management Task Force, June 2003.
64. Campbell RH, Nahrstedt K *et al.* Gaia: Active spaces for ubiquitous computing. Project home page, 2002. http://choices.cs.uiuc.edu/gaia [18 January 2005].
65. Nahrstedt K, Wichadakul D, Xu D. Distributed QoS compilation and runtime instantiation. *Proceedings of the IEEE/IFIP International Workshop on QoS (IWQoS'2000)*, Pittsburgh, PA, June 2000.
66. Wichadakul D, Nahrstedt K, Gu X, Xu D. 2KQ+: An integrated approach of QoS compilation and reconfigurable, component-based run-time middleware for the unified QoS management framework. *Proceedings of the 3rd IFIP/ACM International Middleware Conference*, Heidelberg, April 2001 (*Lecture Notes in Computer Science*, vol. 2218). Springer: Berlin, 2001; 373–394.
67. Román M, Hess CK, Cerqueira R, Ranganathan A, Campbell RH, Nahrstedt K. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing* 2002; 74–83.