Most likely to be published in Communications of the ACM, October 2001

Multiparadigm Communications in Java for Grid Computing

Vladimir Getov, Gregor von Laszewski, Michael Philippsen, Ian Foster

The computational science community has long been at the forefront of advanced computing, because of its persistent need to solve problems that require resources beyond those provided by the most powerful computers of the day. Examples of such high-end applications range from financial modeling and vehicle simulation to computational genetics and weather forecasting. Over the years, such considerations have led computational scientists to become aggressive and innovative adopters of vector computers, parallel systems, clusters, and other novel computing technologies.

Recently, the widespread availability of high-speed networks and a growing awareness of the new problem solving modalities, made possible when these networks are used to couple geographically distributed resources, have stimulated interest in so-called Grid computing [5]. The term "the Grid" refers to an emerging persistent infrastructure providing security, resource access, information, and other services that enable controlled and coordinated resource sharing among "virtual organizations" formed dynamically from individuals and organizations sharing a common interest [6]. A variety of ambitious projects are applying Grid computing concepts to such challenging problems as the distributed analysis of experimental physics data, community access to earthquake engineering facilities, and the creation of "science portals" - thin clients providing remote access to the collection of information sources and simulation systems supporting a particular scientific discipline.

Underpinning both parallel and Grid computing is a common need for coordination and communication mechanisms that allow multiple resources to be applied in a concerted fashion to complex problems. Scientific and engineering applications have, for the most part, addressed this requirement in an ad hoc and low-level fashion, using specialized message-passing libraries within parallel computers and various communication mechanisms among networked computers.

While low-level approaches have allowed users to meet performance goals, an unfortunate consequence is that the computational science community has not benefited to any great extent from the significant advances in software engineering that have occurred during the past ten years in industry. In particular, the various benefits of Java, which seems ideal for multiparadigm communication environments, are hardly exploited at all. Java's platform-independent bytecode can be executed securely on many platforms, making Java an attractive basis for portable Grid computing. In addition, Java's performance on sequential codes, which is a strong prerequisite for the development of such "Grande" applications (see sidebar on Java Grande), has increased substantially over the past years [3]. Inspired originally by coffee house jargon, the buzzword "Grande" has become commonplace in order to distinguish this emerging type of high-end applications when written in Java.[1] Furthermore, Java provides a sophisticated graphical user interface framework, as well as a paradigm to invoke methods on remote objects. These features are of particular interest for steering of scientific instruments from a distance.

In this article, we argue that the rapid development of Java technology now makes it possible to support, in a

1"Grande" is the prevalent designation for the term "large" or "big" in several languages. In the U.S. (and the Southwest in particular), the term "grande" has established itself as describing size within the coffee house scene.

single object-oriented framework, the different communication and coordination structures that arise in scientific applications. We outline how this integrated approach can be achieved, reviewing in the process the state-of-the-art in communication paradigms within Java. We also present recent evaluation results indicating that this integrated approach can be achieved without compromising on performance.

Communication Requirements Analysis

Communication and coordination within scientific and engineering applications combines stringent performance requirements with (especially in Grid contexts) highly heterogeneous and dynamic computational environments. Different communication frameworks have been introduced during the years as a result of the development of computer networks, distributed and parallel computing systems. An approximate road-map of this development during last two decades is shown in Figure 1. The pioneering framework, remote procedure call (RPC), has been around for at least twenty years. The message-passing paradigm followed in the 80s as part of the introduction of distributed memory parallel machines. In recent years two other frameworks were developed on the basis of the RPC concepts - the remote method invocation (RMI) and the components framework. One can recognize three distinct regimes based on these existing technologies:

- Within parallel computers and clusters, communication structures and timings are often highly predictable for both senders and receivers and may involve multi-party ("collective") operations. Here efficiency is the main concern, and message-passing libraries such as the Message Passing Interface (MPI) [7] have emerged as the technology of choice.
- When components of a single program are distributed over less tightly coupled elements or when collective operations are rare, communication structures may be less predictable, and issues such as asynchrony, error handling, and ease of argument passing become more prominent. Here, technologies such as CORBA or Java's RMI have benefits.
- When constructing programs from separately developed components, the ability to compose and discover the properties of components becomes critical. Component technologies such as JavaBeans and their associated development tools become very attractive, as do proposed high-performance component frameworks [1].

As we explain in the following, all three approaches to communication and coordination can be expressed effectively within Java.

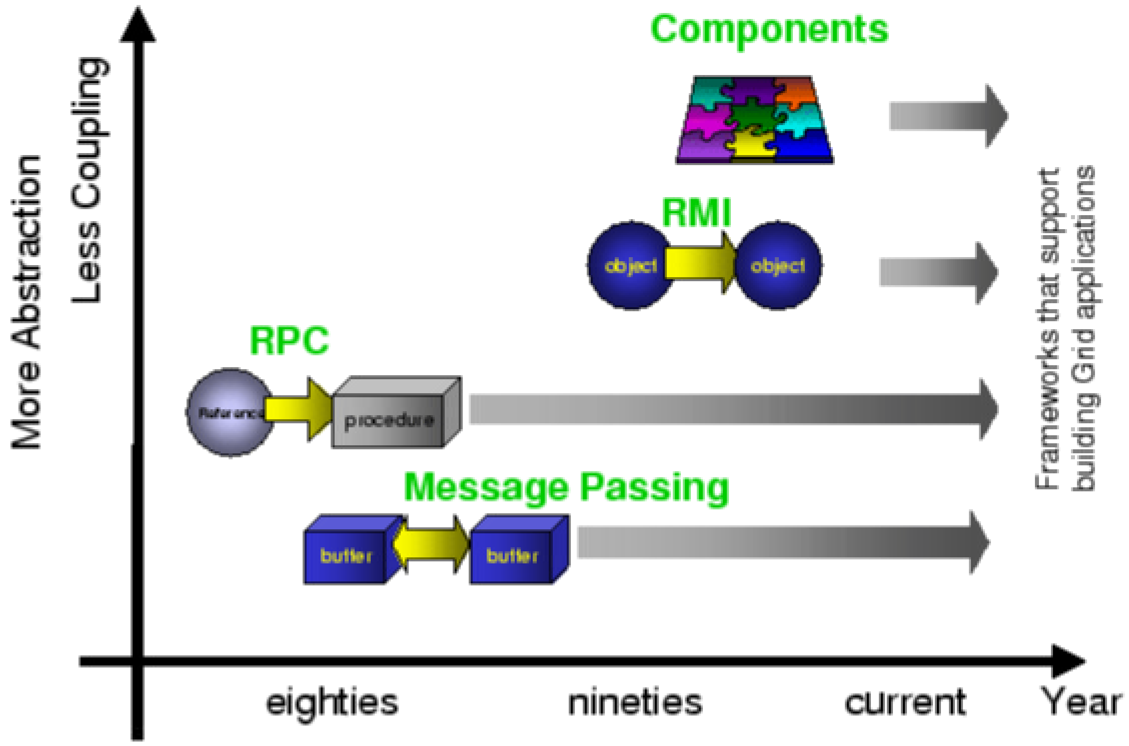

Figure 1: Multiple communication frameworks are used to support programming the diverse infrastructure that comprises Grids. Remote procedure calls (RPC), message passing, remote method invocation (RMI), and component frameworks have been introduced at various times and are now the technologies of choice for building Grid applications.

Message Passing

The Java language has several built-in mechanisms that allow the parallelism inherent in a given program to be exploited. Threads and concurrency constructs are well suited for shared memory computers, but not large-scale distributed memory machines. For distributed applications, Java provides sockets and an RMI mechanism. For the parallel computing world, the former is often too low-level and the latter is oriented too much towards client/server type systems and does not specifically support the symmetric model adopted by many parallel applications. Obviously, there is a gap within the set of programming models provided by Java, especially for parallel programming support on clusters of tightly coupled processing resources. A solution to this problem inevitably builds around the message-passing communication framework, which has been one of the most popular parallel programming paradigms since the 1980s.

In general, the architecture of a message-passing system can follow two approaches - implicit and explicit. Solutions that adopt the implicit approach usually provide the end user with a single shared memory system image, hiding the message passing at a lower level of the system hierarchy. Thus, a programmer works within an environment often referred to as the distributed shared memory programming model. Translating the implicit solution to Java leads to the development of cluster-aware Java virtual machines (JVM) that provide fully transparent and truly parallel multithreaded programming environments [8]. This approach preserves full compatibility with the standard Java bytecode format. The price to be paid for these advantages, however, is a nonstandard JVM that introduces extra complexity and overhead to the Java runtime system. This makes it difficult for such JVMs to keep up with the continuous improvements and performance optimizations of the standard technology.

In contrast with sockets and RMI, explicit message passing directly supports symmetric communications including both point-to-point and collective operations such as broadcast, gather, all-to-all, and others, as defined by the MPI standard. Programming with MPI is easy because it supports the single program multiple data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values.

With the evident success of Java as a programming language, and its inevitable use in connection with parallel, distributed, and Grid computing, the absence of a well-designed explicit message-passing interface for Java would lead to divergent, non-portable practices. Indeed, the message passing working group of the Java Grande Forum was formed as a response to the appearance of various explicit message-passing interfaces for Java. Some of these early "proof-of-concept" implementations have been available since 1997, with successful ports on clusters of workstations running Linux, Solaris, Windows NT, Irix, AIX, HP-UX, and MacOS, as well as on parallel platforms such as the IBM SP-2 and SP-3, Sun E4000, SGI Origin-2000, Fujitsu AP3000, and Hitachi SR2201. An immediate goal was to discuss and agree on a common MPI-like application programming interface (API) for message passing in Java (MPJ) [4]. The purpose of this first phase of the effort was to provide an immediate, ad hoc specification for portable message-passing programming in Java, as well as to provide a basis for conversion between C, C++, Fortran, and Java.

MPJ can be implemented in two different ways: as a wrapper to existing native MPI libraries or as a pure Java implementation. The first approach provides a quick solution, usually with only negligible time overhead introduced by the wrapper software. However, the use of native code breaks the Java security model and does not allow work with applets - clear advantages of the pure Java approach. Unfortunately, a direct MPJ implementation in Java is usually much slower than the use of wrapper software to existing MPI libraries. One solution to this problem is to employ more sophisticated design solutions. For instance, the use of native conversion into linear byte representation, often referred to as "marshaling", and advanced compilation technologies for Java can make the two approaches comparable in terms of performance. In our experiments we have used the statically optimizing IBM High-Performance Compiler for Java (HPCJ), which generates native codes for the RS\6000 architecture, to evaluate the performance of MPJ on an IBM SP-2 machine. The results show that when using such a compiler, the MPJ communication component is as fast as the native message-passing library (see Figure 2).

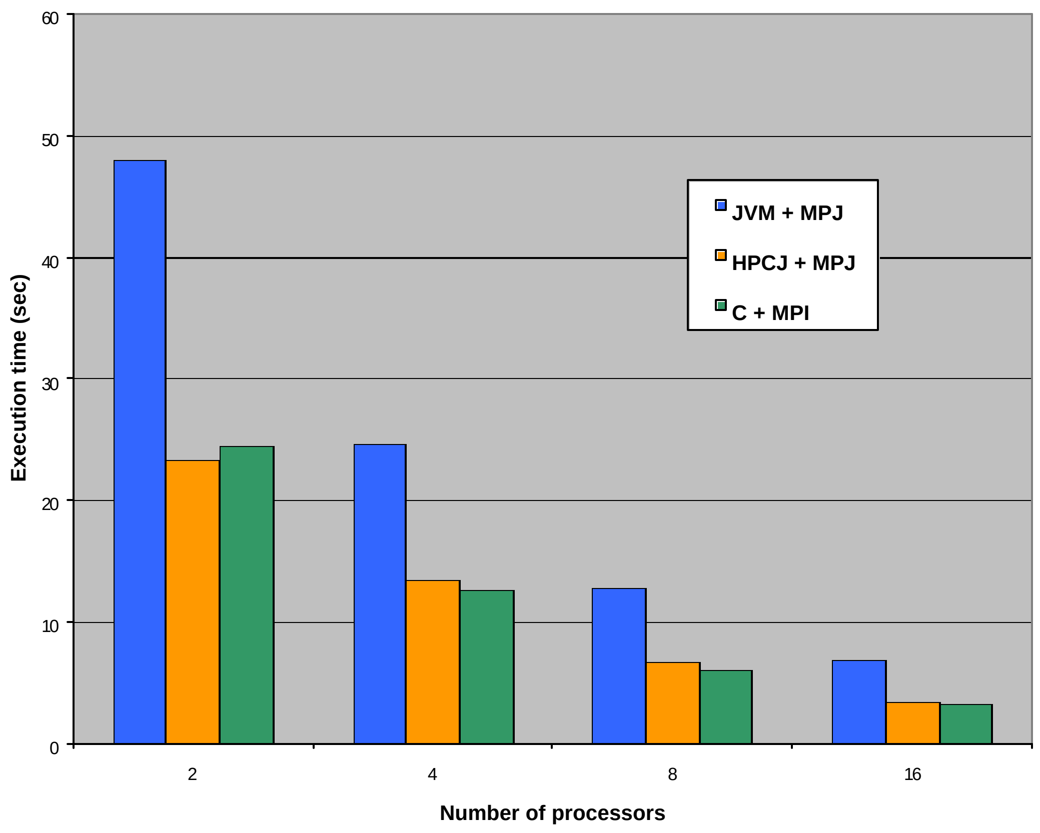

Figure 2: Execution time for the Integer Sort kernel from the NAS Parallel Benchmarks on the IBM SP-2. The use of JVM and MPJ in this particular case is approximately 2 times slower than the same code written in C and using MPI. When using HPCJ and MPJ, however, the difference disappears, and Java and MPJ perform as well as C and MPI for this experiment. This result confirms that the extra overhead introduced by MPJ in comparison with MPI is negligible.

Closely modeled as it is on the MPI standards, the existing MPJ specification should be regarded as a first phase in a broader program aimed at defining a more Java-centric high performance message-passing environment. We can expect future work to consider more high-level communication abstractions and perhaps layering on other standard transports and on Java-compliant middleware. Middleware developed at this level should offer a choice of emphasis between performance or generality, while always supporting portability. One can also note the opportunity to study and implement aspects of real-time and fault-aware message-passing programs in Java, which can be particularly useful in Grid environments. Of course, a primary goal should be to offer MPI-like services to Java programs in an upward-compatible fashion. The purposes are twofold: performance and portability.

Fast Remote Method Invocation

Remote invocation is a well-known concept [2] which has been underpinning the development of both originally RPC and then Java's RMI. Today, CORBA uses RPC to glue together code that is written in different languages. To implement a remote invocation, the procedure identifier and its arguments are encoded (marshaled) in a wire format that is understood by both the caller and the callee. The callee uses a proxy object to decode (unmarshal) that stream of bytes and then to perform the actual invocation. The results travel in the other direction.

Although RMI inherits this basic design in general, it has distinguishing features that reach beyond the basic RPC. (For RMI details see the RMI sidebar in [8].) RMI is no longer defined as the common denominator of different programming languages. It is not meant to bridge between object-oriented and procedural languages or to bridge between languages with different kinds of elementary types and structures. With RMI, a program running on one JVM can invoke methods of other objects residing in different JVMs. The main advantages of RMI are that it is truly object-oriented, that it supports all the data types of a Java program, and that it is garbage collected. These features allow for a clear separation between caller and callee. Development and maintenance of distributed systems become easier.

Let us consider the remote invocation of a method `add(Atom name)` in order to illustrate these advantages. In contrast to earlier RPCs, RMI is truly object-oriented and allows the caller to pass objects of any subclass of `Atom` to the callee. The object is encoded and shipped in such a way that the callee gets hold of the object instead of a reference to it. This encoding into a machine independent byte representation - Java uses the term "object serialization" - includes also information on the class implementation. More precisely, if the callee does not know the concrete class implementation of `name`, it can load that class implementation dynamically. Thus, not only are the values of an object shipped, but also the whole implementation. When the caller invokes an instance method on `name`, say, `name.bond`, the `bond` code of the particular subclass of `Atom` will be executed at the side of the callee. Thus, one of the main advantages of object-oriented programming, the reuse of existing code with refined subclasses, can be exploited for distributed code as well. Caller and callee can be developed separately as long as they agree on interfaces. From the software engineer's point of view, another advantage of RMI is the distributed garbage collection. Since most practitioners agree that, for sequential code, garbage collection helps save programmer time, it is likely that the same is true for distributed code as well.

These novel features come at a cost. With a regular implementation of RMI on top of Ethernet, a remote method invocation takes milliseconds - concrete times depend on the number and the types of arguments. About a third of that time is needed for the RMI itself, a third for the serialization of the arguments, and a third for the data transfer (TCP/IP-Ethernet). While this kind of latency might be acceptable for coarse-grained applications with little communication needs, it is too slow for high-performance applications that run on low-latency networks, for example, on a closely connected cluster of workstations.

Several projects are under way to improve the performance of RMI, for example, the Manta project [9] and the JavaParty project [10]. In addition to simply improving the implementation, for example, by removing layering overhead in the RMI implementation, these projects use the following main optimization ideas.

- Precompile marshaling routines, rather than generating them at runtime via dynamic type inspection.
- Employ an optimized wire protocol especially for type encoding. It is necessary to ship detailed type descriptions only if the objects are stored into persistent storage. For communication purposes alone, a short type identifier may be sufficient.
- Cache (or replicate) objects so as to avoid retransmission if their instance variables do not change between calls.
- Minimize the number of copy operations on the way from the object to the communication hardware and back. Minimize thread switching overhead. Because Java itself is multithreaded, traditional RPC optimizations in general are insufficient. Optimized RMI implementations in Java cannot be as aggressive as native approaches because Java's virtual machine concept does neither allow direct access to raw data nor does it make the JVM's internal handling of threads transparent.
- Use an efficient communication subsystem. JavaParty's RMI is implemented on top of the Myrinet-

based ParaStation library (http://ParaStation.ira.uka.de/) that employs user level communication and hence avoids costly kernel operations.

The JavaParty project has optimized both RMI and the object serialization in pure Java. Remote invocations can be completed within 80 µs on a cluster of DEC-Alpha computers connected by Myrinet. Figure 3 shows, that for benchmark programs, 96% of the time can be saved if the fast serialization, the fast RMI, and Myrinet communication hardware are used. With similar optimization ideas, the Manta group compiles to native code and uses a runtime system written in C. Manta is therefore less portable but reports faster remote invocations (40 µs on a Pentium Pro) [8].

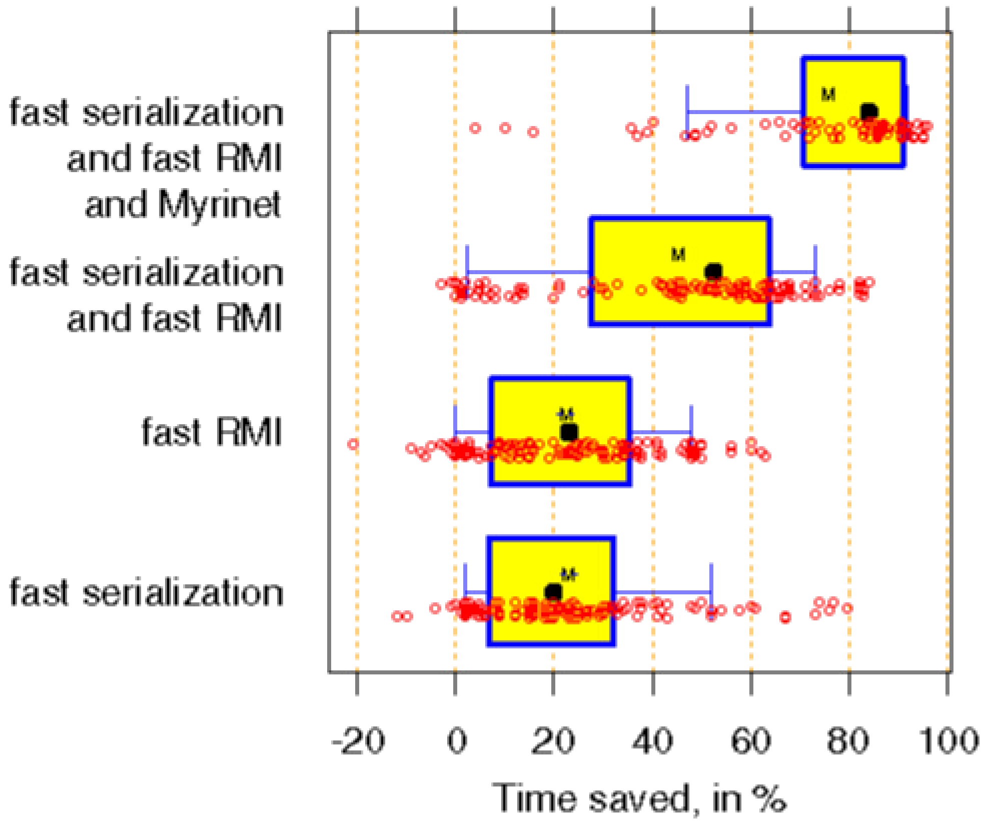

Figure 3: The bottom three box plots each show 2 * 64 measured results for diverse benchmarks (64 points represent measurements on PCs connected by Ethernet, 64 stand for Dec-Alphas connected by FastEthernet). The first (bottommost) box plot shows the runtime improvement that was achieved with regular RMI and the fast serialization. The second box plot shows the improvement that fast RMI achieves when used with Java's regular serialization. The third box plot shows the combined effect. The top line demonstrates what happens if Myrinet cards are used to connect the Alphas in addition to the fast serialization and fast RMI (64 measured results). In all box plots the small circles are the individual data points. The fat dot is the median; the box indicates the 25% and 75% quantiles; the whiskers indicate the 10% and 90% quantiles; the M and dashed line indicate the mean plus/minus one standard error of the mean.

Components: Enabling Adaptive Grid Computing

So far we have discussed how well-known communication paradigms can be made available and efficient in Java. Nevertheless, additional advances are needed to realize the full potential of emerging Grids, where we must deal with heterogeneous systems, diverse programming paradigms, and the needs of multiple users. Adaptive services must be developed that provide security, resource management, data access, instrumentation, policy, and accounting for applications, users, and resource providers.

Java provides some significant help to ease this software engineering problem. Because of its object-oriented nature, the ability to develop reusable software components, and the integrated packaging mechanism, Java offers support for all phases of the lifecycle of a software engineering project, including problem analysis and design, program development, deployment, instantiation, and maintenance.

Java's reusable software component architecture, known as “JavaBeans”, allows users to write self-contained, reusable software units (see sidebar on components and JavaBeans). With commercially available visual application builder tools, software components can be composed into applets, applications, servlets, and composite components. Components can be moved, queried, or visually integrated with other components, enabling a new level of convenient Computer Aided Software Engineering (CASE) based programming in the Grid.

Component repositories or containers provide the opportunity to work collectively on similar tasks and share the results with the community. Additionally, the Java framework includes a rich set of predefined Java APIs, libraries, and components that support access to databases and directories, network programming, and sophisticated interfaces to XML, to list only a few.

We have evaluated the feasibility of using these advanced features of Java for Grid programming, as part of our development of several application-specific Grid portals. A portal defines a commonly known access point to the application that can be reached via a Web browser. All of these portal projects use a common toolkit, called the Java Commodity Grid (CoG) Kit [11] allowing access to services provided by the Globus Toolkit (http://www.globus.org) in a way familiar to Java programmers. Thus, the Java CoG Kit is not a simple one-to-one mapping of the Globus API into Java; it makes use of features of the Java language that are not available in the original C implementation. These features, for example, include the use of the object-oriented programming model and the event model.

Another important advantage of Java is the availability of a graphical user interface for integrating graphical components into Grid-based applications. Our extensive experience with collaborators from various scientific disciplines, such as structural biology, or climatology, showed that the development of graphical components, hiding the complexity of the Grid, lets the scientist concentrate on the science, instead of dealing with the complex environment of a Grid.

For example, a structural biology application system developed by two of the authors with their colleagues [12] links in a pipeline a specialized scientific instrument (an X-ray source beamline) with a data acquisition computer, a parallel computer used for data reconstruction, and other computers employed as science portals for data visualization. The implementation uses a mixture of message passing for reconstruction and home-grown remote procedure call for communication within the pipeline. An acceptable computational speed for this application requires enormous data rates and compute power, easily reaching the Gbit/s and the Tflop/s range, respectively. Recently, we have begun to reengineer (see Figure 4) such a challenging interdisciplinary application to utilize Java technology while simplifying program development and use by the domain specialist.

Java not only simplifies program development but also eases the development and installation of client software that accesses the Grid. While it is trivial to install client libraries of the Java CoG Kit on a computer, the installation of client software written in other programming languages or frameworks such as C and C++ is much more involved, because of differences in compilers and operating systems. Another advantage of using the bytecode compiled archives is that they can also be installed on any operating system that supports Java, including the Windows operating system. Using the Java framework allows development of drag and drop components that enable information exchange between the desktop and the running Grid application during a program instantiation. Thus, it is possible to integrate Grid services seamlessly into the Windows or the Unix desktop.

A commodity technology such as Java as the basis for future Grid-based program development offers yet another advantage. The strong and committed support of Java by major vendors in e-commerce allows scientists to exploit a wider range of computer technology - from supercomputers to state-of-the-art commodity devices such as cell phones, PDAs, or Java-enabled sensors - all within a Grid-based problem-solving environment.

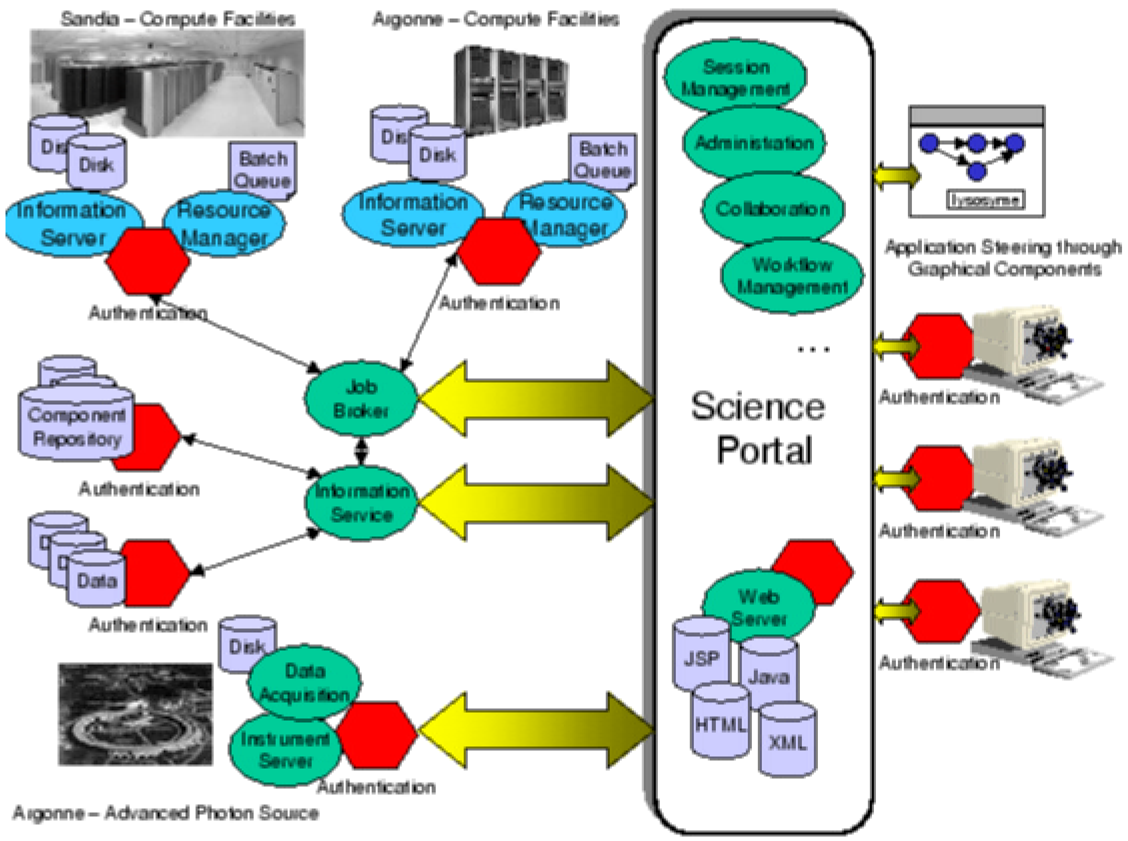

Figure 4: With the help of the Java Commodity Grid Kit, we are able to construct a domain specific science portal from reusable components. The components access Grid services for authentication, job submission, and information management. Advanced services such as workflow management must be developed to provide an integrated problem solving environment for scientists.

Conclusion

Advanced applications such as those that arise in science and engineering can require the use of multiple different communication abstractions, ranging from message passing to remote method invocation and component frameworks. We have shown here how a mixture of existing Java constructs and innovative implementation techniques allow one to use these different communication abstractions efficiently within a single integrated Java framework. The result is a programming approach that appears particularly advantageous in dynamic and heterogeneous Grid environments.

Acknowledgments

Research conducted by Vladimir Getov was supported in part by Higher Education Funding Council for England (U.K.) under the NFF program and by Los Alamos National Laboratory (U.S.A.) under the Exchange Visitors program. The research conducted by Gregor von Laszewski and Ian Foster, was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; by the National Science Foundation; and by the NASA Information Power Grid project.

Bibliography

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, and S. Parker. Toward a common component architecture for high performance scientific computing. In *Proc. 8th IEEE Symp. on High-Performance Distributed Computing*. IEEE Press, 1999.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.

[3] R. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *IEEE Computing in Science and Engineering*, 3(2):22-28, March/April 2001. To appear.

[4] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019-1038, September 2000.

[5] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[6] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001. To appear. http://www.globus.org/research/papers.html.

[7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[8] T. Kielmann, P. Hatcher, L. Bougé, and H. Bal. Enabling Java for high-performance computing: Exploiting distributed shared memory and remote method invocation. *Communications of ACM (this special issue)*, 2001. To appear.

[9] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *Proc. 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP*, pages 173-182, Atlanta, GA, May 1999.

[10] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495-518, May 2000. http://wwwipd.ira.uka.de/JavaParty/.

[11] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity Grid kit. *Concurrency and Computation: Practice and Experience*, 13, 2001. To appear.

http://www.globus.org/cog/documentation/papers/.

[12] G. von Laszewski, M. Westbrook, I. Foster, E. Westbrook, and C. Barnes. Using computational Grid capabilities to enhance the ability of an X-ray source for structural biology. *Cluster Computing*, 3(3):187-199, 2000.

Sidebar: Ten Reasons to Use Java in Grid Computing

1. Language:
The Java programming language includes features that are beneficial for large scale software engineering projects, such as packages, object orientation, single inheritance, garbage collection, and unified data formats. Since threads and concurrency control mechanisms are part of the language, it is possible to express parallelism directly at the lowest user level.

2. Class Libraries:
Java provides a wide variety of additional class libraries including functions that are essential for Grid computing, such as the capability of performing secure socket communication or message passing.

3. Components:
A component architecture is provided through JavaBeans and Enterprise JavaBeans to enable component based program development.

4. Deployment:
Java's bytecode allows for easy deployment of the software through Web browsers and automatic installation facilities.

5. Portability:
Besides the unified data format, Java's bytecode guarantees full portability following the innovative and popular "write-once-run-anywhere" concept.

6. Maintenance:
Java contains an integrated documentation facility. Components that are written as JavaBeans can be integrated within commercially available Integrated Development Environments (IDEs).

7. Performance:
Recent research results have proven that the performance of many Java applications can come very close to that of C or FORTRAN.

8. Gadgets:
Java-based smart cards, PDAs, and smart devices will expand the working environment for scientists.

9. Industry:
Scientific projects are sometimes required to evaluate the longevity of a technology before it can be used. Strong vendor support for Java helps make Java a technology of current and future consideration.

10. Education:
Universities all over the world are teaching Java to their students.

Sidebar: Java Grande

The notion of a Grande application is familiar to many researchers in academia and industry but the term itself is relatively new. Such applications can potentially require any combination of high-end processing, communicating, I/O, or storing resources to solve one or more large-scale problems. The use of Java, both as a language and as a platform, for solving this type of problems has been the focus of the international Java Grande Forum (http://www.javagrande.org) since 1997. With active members from academia, research institutions, and industry, the main Forum goals are to:

- evaluate and improve the applicability of the Java environment for Grande applications;
- bring together the Java Grande community to develop consensus requirements and act as a focal point for interactions with the much larger Java community;
- create prototype implementations, demonstrations, benchmarks, API specifications, and recommendations for improvements, in order to make the Java environment useful for Grande applications.

The Forum organizes open public meetings, the annual ACM Java Grande Conference, workshops, minisymposia, and panels. A large portion of the Forum's scientific work including early workshop and conference proceedings have been published in some issues of *Concurrency: Practice & Experience* - vol. 9, numbers 6 and 11, vol. 10, numbers 11-13, vol. 12, numbers 6-8. Since 1999 the proceedings have been published annually by *ACM Press*.

Sidebar: Components and JavaBeans

Software component:

A software component is a unit of composition. Its design is restricted by a contract that requires a specific set of interfaces. Software components can be reused, interchanged, and deployed independently and are subject to composition into bigger systems by third parties. For instance, builder tools can extract design information, determine the component's capabilities, and expose them conveniently to the software engineer to encourage reuse.

JavaBeans:

A Bean is a reusable software component that may be visually manipulated using builder tools (http://java.sun.com/products/javabeans/). Typical features of JavaBeans include

- **dynamic type inspection** (often called introspection in Java's terminology) allowing other programs to analyze at runtime how the defined bean works.
- **customization** allowing a user to alter the appearance and behavior of a bean.
- **events** allowing beans to fire events and provide - by means of dynamic type inspection - information to builder tools, concerning both the events they can fire and the events they can handle.
- **properties** allowing beans to be manipulated and customized programatically.
- **persistence** allowing the state of a customized bean to be saved and restored.

Beans are particularly useful for computational scientists and application experts that benefit through the reuse of components designed for Grid computing.

About the Authors

Vladimir Getov
is a reader in high performance computing at the School of Computer Science, University of Westminster in London and a visiting scientist at the Computer & Computational Sciences Division, Los Alamos National Laboratory. His primary research interests include performance engineering, message passing, and distributed computing. **Author's Present Address**: School of Computer Science, University of Westminster, Watford Rd, Northwick Park Harrow HA1 3TP, U.K.; email: V.S.Getov@wmin.ac.uk

Gregor von Laszewski
is an assistant scientist in the Mathematics & Computer Science Division, Argonne National Laboratory and a fellow at the Computation Institute of the University of Chicago. His research interests include Grid computing and application of commodity technologies in computational science. **Author's Present Address**: Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.; email: gregor@mcs.anl.gov

Michael Philippsen
is a senior researcher in the computer science department at the University of Karlsruhe, Germany. He also manages the software engineering group of the Forschungszentrum Informatik (FZI) in Karlsruhe. His primary research interests are parallel systems, software engineering, and languages. **Author's Present Address**: Computer Science Department, University of Karlsruhe, 76128 Karlsruhe, Germany. email: michael@philippsen.com

Ian Foster
is a senior scientist and associate director in the Mathematics & Computer Science Division, Argonne National Laboratory, and a professor of Computer Science at the University of Chicago. His research interests include Grid computing and computational science. **Author's Present Address**: Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.; email: foster@mcs.anl.gov