On the Interaction between Mobile Processes and Objects

Suresh Jagannathan Richard Kelsey
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
suresh|kelsey@research.nj.nec.com

Abstract

Java's remote method invocation mechanism provides a number of features that extend the functionality of traditional client/server-based distributed systems. However, there are a number of characteristics of the language that influence its utility as a vehicle in which to express lightweight mobile processes. Among these are its highly imperative sequential core, the close coupling of control and state as a consequence of its object model, and the fact remote method calls are not properly tail-recursive. These features impact the likelihood that Java can easily support process and object mobility for programs which exhibit complex communication and distribution patterns.

1 Introduction

Distributed systems have historically been concerned with issues concerning the partitioning and transmission of data among a collection of machines [15]. Typically, these systems allow code to be distributed and accessed in one of two ways. In some systems, each node holds code controlling the resources found on that node. In others, the same code image is found on all nodes. In either case, some form of message-passing [19] is used to invoke operations on remote sites. By and large, mobility has not been an issue of significant importance. In client-server based systems, process mobility is essentially irrelevant: tasks are heavyweight and control resources resident on a particular machine. In systems where all machines share the same code image, process mobility may be used to help performance by improving locality and load-balancing. However, tasks typically execute heavyweight procedures often closed over a large amount of local state, making task migration expensive. Indeed, devising an efficient task migration policy that has a simple well-understood semantics is still a subject of active research.

The past few years has seen an increasingly apparent shift to a new computational paradigm. Instead of regarding the locus of an executing program as a single address-space physically resident on a single processor, or as a collection of independent programs distributed among a set of processors, the advent of languages like Java [8] has offered a compelling alternative. By allowing concurrent threads of control to execute on top of a portable, distributed virtual machine, Java presents a view of computation in which a single program can be seamlessly distributed among a collection of heterogeneous processors. Unlike distributed systems that require the same code to be resident on all machines prior to execution, Java allows new code to be transmited and linked to an executing process. This feature allows Java to upload functionality dynamically in ways not possible in a traditional distributed system.

Currently, the Java core only supports migration of whole programs; threads of control are not transmitted among machines. However, extensions like Java/RMI [20] that enable client-server (RPC-style) semantics do allow data as well as code to be communicated among machines in a Java ensemble. Such extensions permit Java programmers to view a computation not merely as a single monolithic unit moving from machine to machine in the form of applets, but as a distributed entity, partitioned among a collection of machines. By using a architecture independent virtual machine, information from one active portion of a computation can be sent to another without deep knowledge of the underlying network infrastructure connecting these pieces together.

In this paper, we explore how distributed extensions to Java such as RMI handle issues pertinent to lightweight task migration. We are particularly interested in how Java's object model, which pervades all aspects of its design and implementation, affects lightweight task and object migration. Some of the questions we examine include:

1. What impact does the close coupling of data and

code in Java have on the implementation of mobile processes?

2. How does Java's imperative sequential core influence influence the design of its distributed extensions?

3. How does a synchronous client-server communication model affect the construction of mobile objects?

4. How do we express where a task should execute, and how an object should migrate as part of an object's behavior? In other words, can we separate issues of concurrency and distribution from issues related to an object's sequential behavior?

Using Java/RMI as a base example, we conclude that features endemic to the Java design make it difficult to express lightweight process mobility. On the other hand, Java's object system significantly extends the functionality found in other distributed languages, most notably in its support for distributed dynamic linking of new code objects.

Our foucs will be on the interaction between mobile processes and objects. After providing some motivation, we present a simple view of a distributed system in the context of an Algol-like (imperative) setting. We then discuss interface description languages like CORBA [14], and their approach to handling distribution. Finally, we describe distributed computation from the Java perspective, and contrast these approaches. We conclude that while Java with RMI support offers significant advantages over its more traditional counterparts to programming heterogeneous networks, it lacks certain features that would enhance its expressivity.

2 Motivation

Like many distributed languages [3, 5], Java's sequential core is object-based. A Java class defines a datatype, and an object is an instance of that type. Operationally, an object defines a collection of data along with operations on that data. A distributed Java program can be now viewed as a collection of objects resident on different address spaces or machines, communicating with each other through visible methods declared by these objects. Under a static view of a distributed computation, processes executing on different machines provide a global service or protect some shared resource. Once allocated to a given machine, however, processes remain stationary. Objects are a natural abstraction for expressing the behavior of distributed processes since they encapsulate shared data through instance variables, and define the operations permitted on this data by other processes through methods and interfaces. For example, a client of a resource need only have access to the operations provided by that resource, and not the actual data manipulated by the resource in order to use the resource effectively. Indeed, for many distributed applications, a static partitioning of shared resources and services via distributed objects is ideal.

Nonetheless, not all applications exhibit such static behavior. As the size of distributed systems grows, or as the complexity of an application increases, defining an efficient static partition of a collection of logically distributed processes becomes problematic. In this case, finer mobility of code and data becomes important [10]. For example, a process executing on a machine in an overloaded network ensemble may need to migrate dynamically to a less-loaded one. In highly heterogeneous systems like the Internet, this functionality becomes even more pronounced: an application may be distributed over many different kinds of machines with widely different capabilities, and may need to be frequently reconfigured to take advantage of changing work-loads or conditions among the nodes on which it executes. To achieve mobility of this kind, code and data must be more loosely coupled: migrating a process from one machine to another should not necessarily entail copying all of the data it may potentially reference as well. Similarly, moving data closer to where a computation requires it, should not entail copying all other processes that also happen to share references to that data. A mobile process should be able to move among a collection of nodes without communicating its intention to the node from whence it came.

3 Mobility in an Imperative Context

We first consider distributed execution in the context of an imperative Algol-like language. These languages have two features that inhibit distribution in general and mobility in particular. The first is that programs generally make progress via side effects, either by updating variables or modifying data structures. The second is that they are first order. Procedures can neither be returned from other procedures, or passed as arguments except in the most trivial cases. The only data available to a procedure are arguments and global variables. It is difficult to only use procedures to simulate the behavior of objects in object-oriented languages, or closures in functional languages.

The result is that computation involves frequent modifications to shared global data, which is exactly what a distributed program needs to avoid. There are two basic approaches to dealing with the problem: distributed shared memory (DSM) [2, 13] and remote procedure call (RPC) [4, 18]. With DSM, the distributed nature of the computation is largely invisible to the programmer, at a high cost in implementation complexity and communication overhead. All data is conceptually associated with a global address. Thus, the machine where a thread executes no longer influences the behavior of the program: dereferencing a global address may involve a remote communication to the node "owning" the contents of that address. While DSM provides a mechanism to implement parallel dialects of imperative languages in a distributed environment, programmers have little control in specifying how coherence and consistency are realized. In particular, issues of process mobility become largely irrelevant since the distribution of data and tasks is implicitly handled by the implementation, and not explicitly managed by the program. The high communication costs associated with DSM make it unattractive in a truly distributed environment. The alternative is to move the burden of handling communication from the implementation to the programmer.

Remote procedure call provides a way of breaking a program into discrete parts each of which runs in its own address space. Unlike DSM, communication is explicit in the program, so programmers have complete control over costs. The difficulty is that the semantics of RPC is substantially different from that of an ordinary procedure call. When procedure P makes an RPC call to Q, the arguments to Q are marshalled and shipped to the machine where the computation should be performed. Stub generators on procedures linked to the application program are responsible for handling representation conversion and messaging. Arguments passed to a remote procedure are passed by copying. Thus, side effects to shared structures can no longer be used for communication between caller and callee. As a result, imperative programs must be substantially modified to run in a distributed environment using RPC.

Process mobility is especially difficult [16, 6, 17]. The imperative nature of these languages means that a large percentage of data found in programs must be global. Communication among processes is enabled via side-effect, and not via allocation and copy. Thus, the advantages of having mobile processes is greatly mitigated. Conceptually, processes are highly mobile in these languages because they carry no state, but because they must frequently reference global (shared) data, process migration becomes useful only if the data they access moves along with the process requiring them. Given that global data is likely to be shared among several processes, the implicit coupling of data and code in imperative languages greatly weakens the utility of process mobility in these languages.

4 Distributed Glue Languages

CORBA [14] and ILU [9] are two well-known object-oriented glue languages that can be used to connect sequential components into a distributed program. The sequential components can be written in a variety of languages and components written in different languages can be freely intermixed within a single distributed program. These glue languages are based on object-oriented interface description languages. The programmer writes a description of each component in the interface language which is then compiled into stub programs. One stub program, in the language in which the component is written, is used by the component to communicate with clients. The rest are used by clients to communicate with the server and can be generated in any of the languages the glue language supports.

Unlike imperative languages, the use of objects alleviates the problems caused by limiting procedures to being first-order. Because each instance of an object has its own local state, the number of side effects on the program's global state is reduced. Unfortunately, commication between components is still done using RPC. The values that may be sent between components are immutable ones: numbers, characters, sequences of values, and so forth. The only references that can be sent over the wire are references to the glue language's global objects. Because components may be written in different languages, each component has not only its own address space but, potentially, its own data representations. A data format used in one component may unrepresentable in another. For example, ILU can be used to connect a component written in C with one written in Common Lisp; the representation of arrays in C, and the operations on them, are quite different from those in Common Lisp. There is no way to send either code or mutable data between components. An object resides permanently on the machine on which it is created. Thus, these distributed glue languages work for large-grain distributed programs with simple, static interfaces. Unlike imperative languages that provide little support for distributed communication, glue languages allow data (either in the form of base types or objects referenced via a global handle) to be accessed in a distributed setting. However, since a CORBA program may consist of modules written in many different languages, there is no support for code or process mobility. Processes run in an address space executing a language processor for the language in which they were written.

Programs written in distributed glue languages are likely to perform well if written in a client/server style. The client/server model partitions computation among nodes in a network: all server-related computation is exercised on the node where the particular server object resides, and all client-related computation is exercised on the client. Because the location of clients and servers is highly static, decisions on where computation is executed is static as well. Once a remote method is invoked, control-flow within the method body remains on the node of the corresponding remote object; the caller of the remote method blocks until the remote method returns.

As long as computation is uniformily distributed among clients and servers, having code remain resident on the remote object site is acceptable. However, for mobile computations, a client/server model is clearly inappropriate. For applications in which computation is non-uniform or which is not easily partitioned into client- and server-code, a more flexible distribution model is required.

Consider a thread of control T executing on some machine M. Suppose T makes a remote method call to a remote object O (possibly written in a different language) executing on some other machine M'. The execution of this method can occur in one of two ways. The method can execute on M as part of T's control-flow, or it can execute on M' as part of a newly instantiated thread. The first option, rejected by distributed glue languages, is useful if references to O within M are infrequent. In this case, the overhead of introducing a new thread control on M' may quite likely be greater than the overhead of retrieving necessary state information from O. The second approach transfers control-flow to M'; this is the approach taken in Java/RMI.

5 Threads and Distributed Objects

In contrast to imperative languages, class-based object-oriented languages like Java encapsulate data and code together. A computational unit in Java is an *object*. An object includes a collection of data called instances variables, and a set of operations called methods to operate on this data. Object state is accessed and manipulated from the outside through publically visible methods. Because it provides a natural form of encapsulation, an object-oriented paradigm seems well-suited for a distributed environment. Objects provide regulated access to shared resources and services. In contrast to distributed glue languages, distributed extensions of Java permit objects as well as base types to be communicated. Moreover, certain implementations such as Java/RMI also permit code to be dynamically linked into an address space on a remote site.

Since a primary goal of Java was to support code migration in a distributed environment, the language provides a socket mechanism through which processes on different machines in a distributed network may communicate. Sockets, however, are a flexible *low-level* communication abstraction. Applications using sockets must layer an application-level protocol on top of this network layer, responsible for encoding and decoding messages, performing type-checking and verification, etc. This is generally agreed to be error-prone and cumbersome.

5.1 Remote Method Invocation

As we discussed earlier, RPC provides one way of abstracting low-details necessary to use sockets. RPC is a poor fit, however, to an object system. In Java, for example, communication takes place among objects, not procedures *per se*. Requiring methods in different objects to communicate directly with one another would break object boundaries and thus would violate the basis of Java's object model. Java/RMI is a variant of remote procedure call tailored for the object semantics defined by Java's sequential core. Instead of using procedure call as the basis for separating local and remote computation, Java/RMI uses objects. A remote computation is initiated by invoking a method on a *remote object*. Clients access remote objects through surrogate objects found on their nodes. These objects are generated automatically by the compiler, and compile to code that handles marshalling of arguments, etc. Like any other Java object, remote objects are first-class, and may be passed as arguments to or returned as results from a method call. Remote objects are implicitly associated with global handles or uids, and thus are never copied across nodes. However, any argument which is not a remote object in a remote object method call is copied, in much the same way as in an RPC semantics. This means that remote calls have different semantics from local ones even though they appear identical syntactically. The fact that Java is highly imperative means that distributed programs must be carefully crafted to avoid unexpected behavior due to unwanted copying of shared data.

Nonetheless, Java/RMI does fit into Java's object model in a number of other ways. Communication takes place via proxies to remote objects, and the encapsulation benefits provided by objects is preserved. In addition, Java/RMI supports a number of features not available in distributed extensions of imperative languages or distributed glue languages. Most important among them is the ability to transfer behavior to and from clients and servers. Consider a remote interface I that defines some abstraction. A server may implement this interface, providing a specific behavior. When a client first requests this object, it gets the code defining the implementation. In other words, as long as clients and servers agree on a policy, the particular mechanism used to implement this policy can be altered dynamically. Clients can send behavior to servers by packaging them as tasks which can then be directly executed on the server. Again, if the method to be executed is not already found on the server, it is fetched from the client. Remote interfaces thus provide a powerful device to dynamically ship executable content *with* state among a distributed collection of machines.

5.2 Tail Recursive Communication and Mobility

Although Java/RMI can be used to express non-client/server style applications, we expect this will not often be the case. By default, a remote method call in Java/RMI is synchronous: the caller waits for the callee to return before proceeding. Stated another way, remote method calls in Java/RMI are never properly tail-recursive. For example, suppose method A on machine M_1 wishes to make a remote method call to method C on M_3 supplying the result of calling remote method B on M_2. Ideally, we would like to have B invoke C directly. However, to do this requires modifying B's implementation. In certain agent-based systems [11], or distributed systems which support first-class continuations [1, 7], B may invoke C directly, wrapping A's continuation around the call. When C finishes, it returns immediately to A, avoiding an unnecessary communication with B. Although a form of asynchronous communication can be expressed in Java/RMI using an agent interface, it is cumbersome. (See Fig. 1.)

5.3 Thread and Object Migration

Despite its added functionality over distributed glue languages, Java's object model (like other object-based distributed systems [10]) encourages a close coupling of code and data. Because all non-local variable references in a method are either global, or refer to instance variables of the object (via self), the environment within which a method executes is explicitly expressed in the definition of the corresponding class. Unlike imperative languages, the location where a thread executes thus becomes very important. Since threads represent control-flow through methods, and methods are the only means of accessing internal object state, distributed object-oriented languages usually dicate that a thread executing a method evaluate on the same node as the method's object. Hence, thread migration is difficult to express: migrating a thread moves it away from the state accessed by the executed method via self. Indeed, Java/RMI provides no programmer-controlled mechanism to express thread migration: once a thread begins execution on a node, it remains resident on that node until the method it is executing completes. Furthermore, since objects contain mutable state, copying data to where the caller resides is likely not to be beneficial since the data must written back to the object when the method completes.

Part of the reason why thread migration appears to be a concept ill-suited in Java is because Java's thread model is so closely tied to its object model. Any object that inherits from the basic thread class, and provides a `run` method can be instantiated as a thread [12]. The code executed by the thread is the code found in the object's run method. Because threads are no different from any other object, thread and object migration are essentially the same. Moving a thread from one node to another is tantamount to moving an entire object, not just control. To achieve the benefits of lightweight thread migration, however, two features are required. First, we should be able to separate code from data, or at least not be required to explicitly package the two together. Second, we should be able to denote a piece of code as a thread without having to first define a class template. An arbitrary Java expression can be viewed as a thread only if it is encapsulated as a method within an object whose class implements or extends the basic `Thread` class. This leads to a significantly greater burden on the programmer to build lightweight threads.

5.4 Specifying a Locus of Control

Another ramification of code and data coupling is that decisions about whether a method is remote or not is hardwired as part of the class specification in which the method is defined. Any object instantiated from a class that implements a `Remote` interface is treated as remote. Thus, all calls to methods found in such objects are executed remotely on the site where the object is located. It is not possible to have some methods execute locally and others execute remotely without having them defined in separate classes. For example, consider a class that contains an array. Methods which operate on all the elements in the array are best implemented via remote method call since it may be potentially expensive to move the

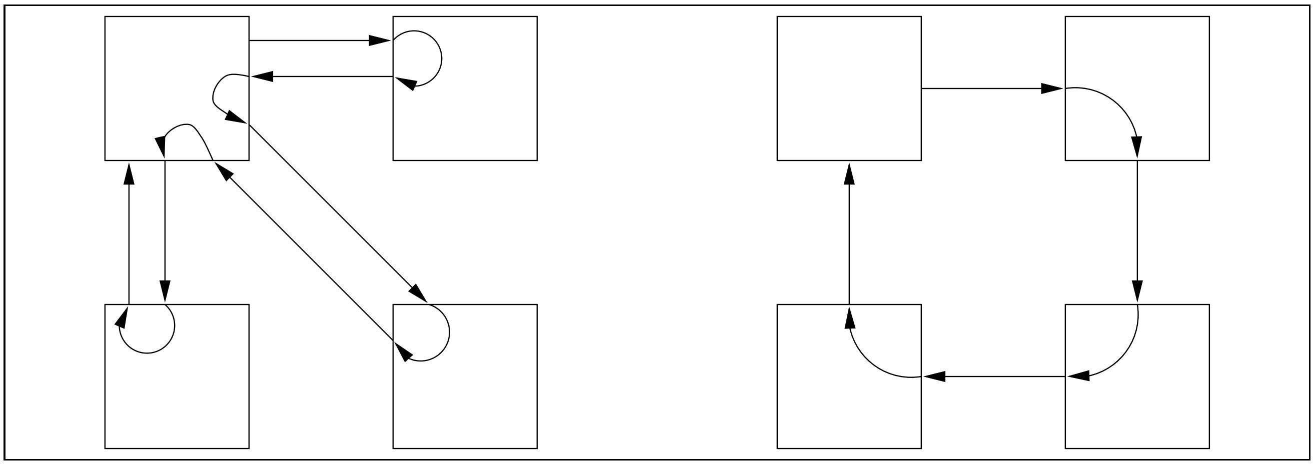

Figure 1: A client-server communication model requires control always to return back to the sender. In agent-based systems, control may move freely among nodes before ultimately returning.

array frequently to the object's clients. On the other hand, it may be more efficient to execute a method which extracts a particular field and operates on that field exclusively locally on the client since communication overhead is likely to be small in this case. In Java/RMI, these two methods cannot co-exist in the same object.

We believe that the choice of where methods execute should be under programmer control. By default, control should remain resident on the node where the thread making the call is currently executing. Field accesses thus involve copying data from the associated object's home. On the other hand, when a method M is annotated as an RPC method, calls to M translate to a shift in control to the location where M's object O is found. If O subsequently migrates while M is still executing, the corresponding thread created to evaluate M migrates as well. Neither of these protocols influence correctness, but their choice impacts efficiency. This finer granularity on control-flow would permit a given object to define methods that support both protocols.

Taken together, these features of Java (and similar class-based languages) make it unwieldy as a language in which to express mobile processes, and dynamic thread and object migration. Because remote calls by default are not tail-recursive. the creator of a thread on a remote node is closely coupled with the thread itself, limiting opportunities for the thread to migrate freely. Requiring that concurrency be expressed through the object system means that expression whose evaluation is to take place in a separate thread must be named as a method found in a `Runnable` object. Since Java provides no mechanism to reify scheduler state, programmers do not have the ability to capture a runnable thread, and move it to another node, or manipulate it in other ways. Any thread or object migration decisions are handled exclusively within the virtual machine. The close explicit coupling between state and code means that moving object state necessarily causes the locus of control for executing methods in that object to shift as well. This is because a class defines an explicit packaging of state used by the methods it defines. Because state referenced by a method is not implicitly constructed by the implementation, it is meaningless to consider mechanisms to distribute control (i.e., threads) as any different from mechanisms to distribute state (i.e., objects).

One way that Java addresses the latter point is through the use of inner classes. An inner class provides many of the features that closures in functional languages provide; in particular, an inner class, allows the same piece of code (an inner class definition) to be closed over many different environments. However, the Java specification requires that free variables in an inner class be final, i.e., immutable. In a functional language, such a requirement does not impose great burdens on expressivity, but functional programming in Java is hard to do because many of its most important features are defined in an imperative style. Thus, we suspect implementations are unlikely to view task migration as a critical issue because distributed programs written in Java will not be able to take advantage of inner classes to separate control from state.

6 Conclusions

Distributed extensions of Java such as Java/RMI combine features found in both agent-based languages and more traditional RPC-based distributed systems. While providing the encapsulation and protection benefits of traditional client/server RPC systems, Java's program model allows code to be dynamically linked and executed on remote nodes.

However, we have identified three fundamental characteristics of Java that we believe make it hard to express lightweight distributed mobile processes. First, the highly imperative nature of its sequential core complicates the semantics of distributed programming via message passing. The semantics of remote method invocation differs substantially from that of local method inocation. Second, a client/server communication model requires a remote call to return back to the sender once the call is complete. Since the continuation of the call cannot be explicitly supplied, mobility is hampered. Third, the close coupling of data and state make it difficult to express task migration as an issue orthogonal to object migration even though the circumstances under which the two would be exercised are very different.

We expect that there is much to be gained by exploring the interaction between an object semantics and distributed programming via mobile processes.

References

[1] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Munin: Distributed Shared Memory Based on Type–Specific Memory Coherence. In *Symposium on Principles and Practice of Parallel Programming* (March 1990).

[3] Birrell, A., Nelson, G., Owicki, S., and Wobber, E. Network Objects. In *14th ACM Symposium on Operating Systems Principles* (1993 December).

[4] Birrell, A. D., and Nelson, B. Implementing remote procedure call. *ACM Transactions on Computer Systems 2*, 1 (1984), 39–59.

[5] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and abstract data types in emerald. *IEEE Transactions on Software Engineering 13*, 1 (1987), 65–76.

[6] Carriero, N., Gelernter, D., Jourdenais, M., and Kaminsky, D. Piranha scheduling: Strategies and their implementation. *International Journal of Parallel Programming 23*, 1 (1995), 5–35.

[7] Cejtin, H., Jagannathan, S., and Kelsey, R. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems 17*, 5 (1995), 704–739.

[8] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Sun Microsystems, Inc., 1995.

[9] Jansen, B., Spreitzer, M., Larner, D., and Jacobi, C. *ILU 2.0alpha12 Reference Manual.* Xerox Corporation, 1997.

[10] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (January 1988), 109–133.

[11] Kotz, D., Gray, R., Nog, S., Rus, S., Chawla, S., and Cybenko, G. AGENT TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing 1*, 4 (1997), 58–67.

[12] Lea, D. *Concurrent Programming in Java: Design Principles and Patterns.* Sun Microsystems, Inc., 1996.

[13] Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (1989), 321–359.

[14] Mowbray, T., and Zahavi, R. *The Essential CORBA: Systems Integration Using Distributed Objects.* Wiley, 1996.

[15] Mullender, S., Ed. *Distributed Systems.* Addison-Welsey, Reading, Mass., 1993.

[16] Powell, M., and Miller, B. Process migration in demos/mp. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (New York, 1983), ACM, pp. 110–119.

[17] Rogers, A., Carlisle, M., Reppy, J., and Hendren, L. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems 17*, 2 (1995), 233–263.

[18] Schroder, M., and Burrows, M. Performance of firefly rpc. *ACM Transations on Computer Systems 8*, 1 (1990), 1–17.

[19] SNIR, M. *MPI: The Complete Reference.* MIT Press, 1996.

[20] WOLLRATH, A., WALDO, J., AND RIGS, R. Java-Centric Distributed Computing. *IEEE Micro 2*, 72 (May 1997), 44–53.

Biographies

Suresh Jagannathan received his Ph.D from the Massachusetts Institute of Technology in 1989. He has spent the past eight years as a Research Scientist at the NEC Research Institute. Prior to joining NECI, he was a Research Faculty member at Yale University. His interests are in the areas of program analysis for mostly-functional languages, compiler design, and distributed programming languages.

Richard Kelsey received his Ph.D from Yale University in 1989. He joined the NEC Research Institute as a research scientist in 1993 after spending three years on the faculty of Northeastern University. His interests are in the areas of programming language design and implementation, usually concentrating on Scheme and similar languages and on building simple, general-purpose compilers.