In Proceedings of the 1998 ACM European Workshop, Sintra, Portugal, September 1998.

Security versus Performance Tradeoffs in RPC Implementations for Safe Language Systems

Chi-Chao Chang, Grzegorz Czajkowski,
Chris Hawblitzel, Deyu Hu, and Thorsten von Eicken

Department of Computer Science, Cornell University, Ithaca NY 14853 USA

(Position Paper)

1 Introduction

In current distributed systems, the performance of remote procedure calls (RPCs) is determined primarily by the performance of the underlying network transport. While the overheads of the RPC system itself are secondary, two ongoing developments are likely to change this and will cause the current RPC systems to become the bottleneck in communication: user-level network interfaces and safe languages. User-level network interfaces such as VIA [2], U-Net [10], Fast Messages [8], NoW Active Messages [1], or Shrimp VMMC [3] are removing the operating system from the critical communication path by allowing applications to access the network interface directly. As a result, the overhead of the network transport underlying RPC decreases by almost an order of magnitude. At the same time, the increasing adoption of Java as the "internet programming language" places a heavier burden on the RPC system because communication among Java programs must satisfy the type safety properties assumed by the language run-time (this is a general issue with safe languages). Typically enforcing this type safety requires additional operations (e.g. checks) in the critical RPC path.

Designing an RPC system for a safe language requires carefully trading off security and performance. Depending on the degree of trust between the communicating parties, different sets of optimizations can be considered. If the run-time systems at the two ends trust each other and if the integrity of the channel between them can be guaranteed, then the marshalling of objects, their type checking at the receiving end, and the message dispatch on arrival can be optimized. In addition, when using user-level network interfaces, the buffer management can be coordinated to avoid copies at both ends. However, if the two run-time systems do not trust each other, then a severe penalty must be paid to enforce the integrity of the safe language environment at the receiving end. In addition, cryptographic techniques may be necessary to protect critical data as well as to guard against eavesdroppers if an untrusted data channel is used. Despite the fact that these techniques incur significant overheads in RPC, offsetting some performance gains obtained through run-time optimizations, many applications seem to be willing to pay for it.

This paper discusses the tradeoffs between security and performance in RPC in the context of the J-Kernel, a Java-based system that enables multiple protection domains to co-exist in a single virtual machine and that addresses many protection issues not dealt by Java itself. In the J-Kernel, domains have separate namespaces and communication occurs through method invocation on capabilities, which are unforgeable references onto resources and services offered by other domains. These capabilities can be freely passed among protection domains and are represented by regular Java objects. The RPC system described in this paper extends this cross-domain method invocation across the network: capabilities can be passed to protection domains on remote nodes and invocations of capability methods result in RPCs. In effect, the RPC system transparently distributes the J-Kernel from single VMs into multiple VMs running on separate machines. The driving application for a single J-Kernel is an extensible HTTP server where users can dynamically extend the server's functionality by uploading servlets that can communicate with one another using capabilities. The current J-Kernel prototype is written entirely in Java and runs on standard Java VMs.

The main motivation for supporting distributed components using the J-Kernel stems from the fact that an HTTP client/server application itself is distributed in nature. Java applets running on browsers often interact with the server from which they originated using I/O streams on top of HTTP and TCP sockets. Using the distributed J-Kernel, applets and servlets can be written using the same abstraction for communication (namely capabilities) while running in domains that reside on different machines. Another motivation is to support extensible clusters of servers for better scalability. Communication between servlets can be based on the same abstraction, independent of their physical location. The various uses for RPC lead to different security requirements that motivate the exploration of the implementation space and the security versus performance tradeoffs that it offers. For example, in a server cluster, the server can use a more secure RPC channel when listening and handling RPC requests coming from client applets, but may choose a faster, less secure channel to communicate with servlets in the cluster that it trusts.

2 RPC in the J-Kernel

Within a single J-Kernel, a domain can create a capability to an object representing a service, which is an automatically generated stub class that wraps the object and inherits its "exported" methods. Invoking a method of a capability obtained from some other domain causes a local cross-domain call. The generated stub is responsible for safely switching between the caller and the callee domain, protecting the caller thread, implementing capability revocation, and applying a special calling convention that enforces domain boundaries: arguments and return values are passed by copy unless they are capabilities, in which case they are passed by reference. In effect, only capabilities are shared between domains. A domain can make a capability available to other domains, either by storing it in a public repository under a certain name, or by passing it as an argument in local cross-domain calls.

A distributed J-Kernel is composed of multiple J-Kernels residing on separate machines. J-Kernel's RPC enables communication between remote domains and uses the same abstraction, namely the capability, as in the local case. A domain can make a service available to remote domains by exporting it under a certain name; similarly, it can import and use a service previously exported by a remote domain. When a domain exports a service, it obtains a capability that allows it to modify some service parameters and to revoke the service as well. When a domain imports a service, it obtains a reference to a proxy object that issues RPC calls to the actual service. The domain can create a capability to the proxy object and pass it to other domains (whether they are local or remote).

RPC in the J-Kernel is implemented by a trusted subsystem which runs on a separate protection domain and handles RPC binding, unbinding, and calls. When a service is exported/imported by a domain, a callee/caller proxy object is created in that domain. These proxy objects handle argument marshalling and unmarshalling and make cross-domain calls into the subsystem to issue RPC requests and replies. Likewise, the subsystem makes cross-domain upcalls to deliver RPC requests and replies. Passing a capability via RPC causes a proxy object to be created in the callee domain (if the capability is an argument) or in the caller domain (if the capability is a return value). To protect the capability from being forged, each proxy object digitally signs RPC requests with a large random number that is shared between the stub object and the RPC subsystem of the J-Kernel in which the service resides. The RPC runtime system resides in a separate protection domain because the J-Kernel does not allow arbitrary domains to access system services such as the network, and because it facilitates the management of resources such as send and receive buffers.

This design is similar in many ways to capability-based systems such as Amoeba [7] and to RPC systems based on safe languages such as Java's Remote Method Invocation [5] and Modula-3's Secure Network Objects [9]. In particular, the J-Kernel design borrows the Java RMI's usage of Remote interfaces to serve as compile-time annotations of objects that implement remote services, and can use Java Object Serialization [5] for data marshalling. An important difference is that the distributed J-Kernel uses the capability abstraction to represent a remote service regardless of its location.

3 Tradeoffs in Performance and Security

State-of-the-art user-level network interfaces (UNI) have enabled a process to directly access the network without the aid of a centralized kernel path. While this eliminates a significant portion of the software overhead in the critical path of an RPC in traditional operating systems, there are a number of hurdles that need to be overcome in order to deliver close-to-raw RPC performance in the distributed J-Kernel. This section describes the design tradeoffs by considering whether the J-Kernels are trusted and whether the data channels are trusted. By a trusted J-Kernel we mean one that issues RPCs with a type-safe data stream and unforged capabilities. By a trusted data channel we mean one that has no eavesdroppers. In all cases, the RPC semantics discussed in the previous section hold at all times. We assume that data channels are reliable (e.g. using TCP) and the OS/hardware is as trusted as the J-Kernels.

3.1 Performance Issues with Type-Safety and User-Level Network Interfaces

Marshalling is one of the remaining performance bottlenecks in network software. Protocol header marshalling consists of converting the RPC header between the end machines' format and the network representation specified in the protocol specification. Argument marshalling includes code to copy the data objects into a send buffer and later from a receive buffer. For cyclic or directed graph structures, this process may require tracking the objects that are being copied in order to avoid duplicate copies. The marshalling performance depends on whether the object formats (determined by class objects in Java) are assumed compatible or not. The J-Kernels can avoiding sending objects format information if they can agree on common object formats. Moreover, if the J-Kernels trust each other, they can copy the data out of the receive buffer without having to perform type and pointer validity checking: the receiver trusts the data it got from the sender. However, all these checks must be performed if the J-Kernels don't trust each other. Type checking requires fetching the argument's type field from the stream and comparing it to the expected type, which requires a table lookup. Pointer validity checking is usually less expensive since pointers refer to objects in the stream and can be represented as indices or offsets in the stream. When receiving a capability as an argument or return value of an RPC between trusted J-Kernels, an optimized implementation can pre-allocate the proxy object during service import-time. In this case, the wire representation will contain a pointer to the existing proxy object, and this pointer will have to be checked for validity if the J-Kernels are untrusted.

If the J-Kernels do not assume object format compatibility, they need to resort to more sophisticated stream protocols such as Java Object Serialization or the ASN.1 protocols, which substantially degrades the end-to-end RPC performance especially when large objects are passed as arguments. For example, the performance of the local RPC in the J-Kernel can be improved by more than one order of magnitude when using a fast copying mechanism instead of Java object serialization.

The RPC implementation uses custom I/O to interface to the network (using Java I/O would be exceedingly slow). Tight integration of the VM and the underlying UNI minimizes the amount of data copying in the critical path. A copy can be eliminated on the receiving end by pre-allocating the objects in a contiguous block of memory and informing the sender of its location, but this requires that the J-Kernels trust each other. In the sending side, zero-copy can only be easily achieved for shallow data structures (e.g. a Java array). Even with compile-time annotations, zero-copy of arbitrary data structures is difficult because the structure of the data cannot be determined ahead of time (e.g. during object allocation).

One final issue is efficient message dispatch, which includes efficient support for interrupt-driven message delivery and lightweight thread switching and scheduling. Most operating systems have elaborate software interrupt mechanisms with high overheads. Some UNI implementations depend on the OS for message arrival notification. Busy polling is an alternative that may be acceptable but may consume a great deal of CPU cycles. While a local cross-domain call abides by the blocking RPC semantics without requiring thread switches, a remote cross-domain call will certainly involve at least two thread switches in the critical path. This underscores the need for a lightweight threads implementation. Threads in the current prototype are built on top of kernel threads for portability reasons -- a user-level implementation of threads will require modifications to the virtual machine.

3.2 Impact of Untrusted Data Channels on Performance

When the data channel is untrusted, the RPC system must provide two security guarantees: integrity and secrecy. Integrity ensures that the callee receives an RPC request, it knows that this request has been issued by a trusted caller and hasn't been altered in transit, and the RPC reply received by the caller hasn't been altered in transit and is the response to that same request sent by the caller. Secrecy ensures that an eavesdropper can not obtain any information regarding the RPC request and reply.

The integrity requirement is enforced by authenticating the J-Kernels using digital signatures, and digitally signing all the handshaking communication between RPC subsystems with point-to-point keys assigned by a reliable, centralized key-distribution service. When an RPC point-to-point channel is established between two J-Kernels, the subsystems agree on a secret suffix for the channel. Besides the additional system complexity, this introduces four digital signature (e.g. MD5) computations on the RPC header and data as well as two byte-array comparisons in the RPC critical path. The RPC subsystems can further encrypt entire outgoing byte-streams to attain complete secrecy using DES.

4 Applications

The following is a brief description of applications we are currently developing using the RPC system in the distributed J-Kernel.

Extensible HTTP Browser: We are currently integrating the J-Kernel technology into Amaya [11], a Java-based web browser developed by W3C

Consortium. Java applets running on browsers often interact with the server from which it originated using standard I/O streams based on HTTP and TCP protocols. In the distributed J-Kernel environment, applets and servlets will use the same abstraction for communication (namely capabilities), allowing functionality to be moved transparently between server and client. In this application, the server and the client do not trust each other and the most conservative implementation of RPC must be used.

Remote Debugging of Extensible HTTP Servers: We have built an extensible HTTP server by integrating the J-Kernel into Microsoft's web server (IIS v3.0). The J-Kernel runs within the same process as IIS (as an in-proc ISAPI extension) and includes a system servlet with access to native methods to receive HTTP requests from IIS and return corresponding replies. This HTTP system servlet forwards each request to the appropriate user servlet, each of which runs in its own J-Kernel domain. This allows arbitrary users to upload custom servlets onto the web server. While this setting is convenient in many ways, it makes debugging servlets difficult. We are using the J-Kernel's RPC system to enable remote debugging: the developer launches a Java VM running J-Kernel on its own machine and the IIS server forwards the HTTP requests through RPC. For this setting, we assume that the communication channels are trusted but not the J-Kernel's.

Scalable HTTP Server: To scale the above HTTP server to a cluster of machines we offload the execution of compute-intensive servlets to J-Kernels running on separate machines. Servlets that perform compute-intensive tasks can be installed on those J-Kernels, and the HTTP server can issue RPC to those servlets. In this scenario, the server can use a more secure RPC channel when listening and handling RPC requests coming from client applets, but may choose a faster, less secure channel to communicate with the compute-servlets which run on trusted J-Kernels in the cluster server.

5 Summary

This paper discusses the design tradeoffs related to two main issues in modern RPC systems: performance and security. In a highly trusted environment, it is possible to perform aggressive optimizations to achieve a base-line RPC performance that substantially closes the performance gap between the raw network performance and the RPC performance. But in an untrusted scenario, the cost of providing security guarantees may be prohibitively high, offsetting any substantial gains in software communication overhead. Still it is interesting to see how the base-line RPC performance varies when security guarantees are introduced one by one and the role a safe language plays during this process. This generates a spectrum of design points that can help application programmers to decide the level of security and performance that is more suitable.

Acknowledgments

This research is funded by DARPA ITO contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations. Chi-Chao Chang is supported in part by a doctoral fellowship (200812/94-7) from CNPq/Brazil.

6 References

[1] B. Chun, A. Mainwaring, D. Culler. *Virtual Network Transport Protocols for Myrinet*. Hot Interconnects V, Stanford, CA, Aug 1997.

[2] Compaq Computer Corp., Intel Corporation, Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0*. http://www.viarch.org/. December 16, 1997.

[3] Dubnicki, C., A. Bilas, Y. Chen, S. Damianakis, and K. Li. *VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication.* Hot Interconnects V, Stanford, CA, Aug 1997.

[4] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*. To appear in Proceedings of 1998 USENIX Annual Technical Conference, June 1998.

[5] JavaSoft. *Remote Method Invocation and Object Serialization Specification.* Available at http://java.sun.com.

[6] B. Lampson, M. Abadi, M. Burrows, and E.Wobber. *Authentication in distributed systems: Theory and Practice*. ACM Transactions on Computer Systems, 10(4):265-310, Nov. 1992.

[7] S. J. Mullender, A. S. Tanenbaum, and R. van Renesse. *Using sparse capabilities in a distributed operating system*. In Proceedings of the 6th IEEE conference on Distributed Computing Systems, June 1986.

[8] Pakin, S., M. Lauria, and A. Chien. *High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet.* In Proceedings of Supercomputing '95, San Diego, California, 1995.

[9] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. *Secure Network Objects*. In Proceedings of the IEEE Security and Privacy Conference, IEEE, Oakland, CA, V.S., 1996.

[10] T. von Eicken, A. Basu, V. Buch, and W. Vogels. *U-Net: User-Level Network Interface for Parallel and Distributed Computing.* In Proceedings of the 15th Annual Symposium on Operating System Principles, p.40-53, Copper Mountain Resort, Colorado, Dec. 1995.

[11] W3C Consortium at http://www.w3.org.