Language Design for Distributed Objects

William R. Cook
University of Texas in Austin
wcook@cs.utexas.edu

Eli Tilevich
Virginia Tech
tilevich@cs.vt.edu

Ali Ibrahim
University of Texas in Austin
aibrahim@cs.utexas.edu

Ben Wiedermann
University of Texas in Austin
ben@cs.utexas.edu

ABSTRACT

The fundamental ideas of distributed objects have changed little in the last 20 years. Existing languages are retrofitted with transparent distribution mechanisms based on proxies. Experiments with mobile code demonstrate its power but have little impact on practice. The problems with transparency and mobile code have been well known since at least 1994. But in the absence of any fundamental new ideas, the same problematic approaches are used, for example in the design of Java RMI. In this essay we discuss a new programming construct called Remote Batch Invocation (RBI). A batch is a code block that combines remote and local execution over fine-grained object interfaces, but is executed by partitioning and remote evaluation. Remote Batch Invocation effectively addresses the shortcomings of transparent distribution with a controlled form of mobile code. Our experience leads us to believe that distribution cannot be implemented as a library, but requires specific language support. Viewing distribution as a language design problem represents a revolutionary step in the development of distributed objects.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.1.3 [**Programming Techniques**]: *Distributed Programming*

General Terms

Languages, Design

Keywords

distributed computing, mobile code, language design, batching

1. INTRODUCTION

Distributed Objects for the 21st Century 2009 Genoa, Italy

The most widely-used approach to implementing distributed objects is to retrofit existing languages with a form of remote procedure call. This pragmatic approach continues to be popular despite increasing awareness of its deficiencies. Numerous alternatives based on mobile code have also been explored, but we believe they have not achieved widespread adoption because they are difficult to use to implement common distribution patterns.

We have recently invented a new programming construct, called a `batch` block, that expands the unit of distribution from single procedure calls to blocks of code. In this essay we discuss the background for this new construct, using Waldo et. al.'s 'A Note on Distributed Computing' [23] as a guide.

2. REMOTE PROCEDURE CALLS

The original motivation for Remote Procedure Calls (RPC) was as a machine-oriented analog to the text-based, conversational command languages used in many distributed protocols [24]. The idea was to replace commands with stub procedures that send messages to the remote system using standard data encoding. The result was transparent distribution, where remote procedure calls work just like local calls. It is interesting to note that conversational command-line interfaces are still the backbone of internet functionality (mail, web, file transfer). Remote procedure calls were generalized to support remote method calls in object-oriented systems [15, 25]. Stubs were generalized to remote object proxies, providing transparent distribution of objects.

A key advantage of RPC is that procedure calls are ubiquitous in programming languages. Thus, most programming languages can express remote method calls in a natural fashion. In particular, a language can support remote calls by simply defining an appropriate library. These libraries can be created automatically by stub generators.

The problems with RPCs are well known [20, 23, 17, 22]. In 1994, engineers at Sun Labs published 'A Note on Distributed Computing' [23], a manifesto which attacked the widely-held view that the goal of distributed object systems should be to hide the complexities of distributed computing. Waldo and colleagues argued persuasively that local and distributed programming models are different, and thus, 'papering over the network' is bound to fail. They focus on four issues: *latency*, *memory access*, *partial failure*, and *concurrency*.

Latency causes inefficiency when invoking multiple remote operations. Numerous proposals have been made to address this problem, including asynchronous RPC and implicit batching [1, 4, 6]. Unfortunately, asynchrony does not

help when a later operation depends upon the result of an earlier one. In addition, fine-grained asynchrony can significantly complicate client programming. Implicit batching is sensitive to small changes in programs, leaving programmers without a clear performance model. Design patterns, including Remote Façades and Data Transfer Objects [8], optimize communication at the cost of tying the server interface design to typical client usage patterns, which compromises transparency and compositionality. In short, efforts to optimize latency within traditional RPC have side-effects that are often as bad as the problem they are meant to solve.

Memory access is a problem because a local pointer is not valid when sent to a remote machine. Distributed object systems use *remote proxies* as a kind of remote pointer to an object on another machine. To unify local and remote computation, proxies must be created automatically whenever a reference to an object is sent remotely. Overuse of proxies leads to chatty communication and increased latency. One alternative is to move objects between client and server, rather than creating proxies. This requires that the local and remote systems share the same runtime, that the code be available on both machines or transmitted dynamically. Since objects themselves often have pointers to other objects, an important problem becomes how one can avoid sending too many objects with each remote procedure call [12]. Remote pointers also introduce the problem of distributed garbage collection [16].

Partial failure is a problem because each component of a distributed computation (e.g., client, server, or the network) can fail independently. Partial failure is both difficult to detect and difficult to handle in a general and reusable fashion. The shared wisdom among distributed system developers has been that effective failure handling is application-specific. That is, each distributed application requires that the programmer writes custom code to handle each possible case of partial failure.

Concurrency is inherent in remote calls, because a distributed application contains at least two separate threads of execution, at each side of distributed communication. These threads do not share a common resource manager, thus making their coordination challenging. For example, a synchronization operation performed on a client proxy of a remote object does not get propagated to the actual remote object on the server. Thus, maintaining the centralized semantics of Java built-in concurrency constructs requires special handling, which can be nontrivial [21].

Despite this analysis, two years later the authors of 'A Note' defined Java RMI, a distributed object model for the Java language [25]. Java RMI is basically an object-oriented implementation of the Remote Procedure Call paradigm that leverages the capacities of the Java language and the Java Virtual Machine to offer a more intuitive model for distributed computing. To address partial failure in RMI, all remote calls are declared as throwing a `RemoteException` that the programmer is responsible for handling. The programmer can too easily avoid this responsibility by simply leaving a `catch` clause empty. Even with these small improvements, we believe that RMI tries to paper over the network in exactly the way that 'A Note' criticizes. The issues of latency, memory access model, partial failure and concurrency remain. Given the problems with remote procedure calls, it is not surprising that alternatives to remote procedure calls have been developed.

3. REMOTE EVALUATION

Remote evaluation [19] is a form of mobile code in which a client sends code to a server to be executed. It generalizes remote procedure calls, which can be viewed as a form of remote evaluation where the code is a single call. Remote evaluation allows the code to contain multiple calls, conditionals, and possibly loops or other control flow constructs. A detailed review of remote evaluation and how it relates to other forms of mobile code is beyond the scope of this essay [9]. Instead we will discuss a few of the issues that have arisen in the exploration of remote evaluation. Some of these issues relate to problems discussed in 'A Note'.

First of all, remote evaluation solves the problem of latency. Any number of remote operations can be performed in one round-trip, and the operations can depend upon each other in complex ways.

Unfortunately, remote evaluation alone does not solve the problem of memory access. What data to transmit with the code, and how the data is migrated, is a difficult problem. There are also issues in deciding what resources the mobile code should be allowed to access. System calls? Constructors? Primitive data types operations? This is both a security issue and a portability issue.

Remote evaluation does have an impact on partial failure: the set of remote operations represented by a single object will be executed atomically with respect to network failures. If a network failure prevents a batch from being sent to the remote server, then no remote operations are performed. Otherwise, all the remote operations are performed unless one of the operations results in a logical exception. Many other issues relating to partial failure still remain, however.

Sending code in a specific byte-code format is often an easy way to go, but it is not language independent. Instead, some work on remote evaluation has defined an intermediate language for the code. Examples include SQL and Tube [10]. The latter uses a form of Scheme as the intermediate language. It is interesting to note that some RPC systems have also included mechanisms for executing multiple calls. This can be viewed as a primitive form of remote evaluation, where the code to be executed is a linear sequence of basic calls. Examples include NFS v4, ONC RPC [18], and Amazon Web Services [2].

The issues listed above are well-known. We have identified another issue in using mobile code for client-server interaction: *how are results returned to the client?* Work on mobile code has focused on sending code, but it is not clear which results to return, or how to return them. And how does the client name and bind variables to the results? If the mobile code makes many method calls and returns many results, the problem becomes worse.

More generally, we think that a key problem is how local and remote code interact. It is not enough to simply send some remote code to the server. It must be meaningfully connected to the client that creates inputs to the remote code and then uses the results returned by the remote code.

4. REMOTE BATCH INVOCATION

Given 40 years of history in building distributed systems, is there any hope in discovering any fundamentally new approach to distributed computing? One must be careful how the question is asked. The functionality of distributed systems and messaging has been thoroughly investigated. It is

unlikely that researchers will find anything new at the level of wire formats or messaging. However, the programming language interface to distributed computing still has some potential for improvement.

One thing that both distributed objects and remote evaluation have in common is that they *assume an existing language* and implement distribution as a library. That is, they don't consider what kind of language changes might be useful for distributed computing. On the other hand some languages have been designed from the ground up for distributed computing. Emerald is an early example, which pioneered the idea of proxies and mobile code [3]. But at that point the issues with distributed objects had not yet surfaced. Research languages have also been designed, but they often have other goals besides practical improvements to distributed objects (Obliq [5], Lambda 5 [14], pi-calculus [13], join calculus [7], etc).

Our perspective is both practical and radical. From a practical viewpoint, we start with the design of widely-used imperative languages, using Java as an example. The same principles would apply to Python, C#, or ML. The radical departure is that we consider any language changes that are useful to support distributed computing. We eventually arrived at an interesting new statement form, analogous to `try` or `synchronized`, that we find useful in building distributed systems.

We have developed Remote Batch Invocation (RBI) as a new distributed programming abstraction that combines the clean programming model of RPC with the efficiency and simplicity of remote evaluation. The technical details of RBI are described in a paper appearing in the main technical program of ECOOP 2009 [11]. In short, the key concept in RBI is the `batch` statement, whose body combines remote and local computation. In Java, a batch block looks like RMI but it executes using remote evaluation. In addition, the batch block moves the necessary data between client and server in bulk. This essay focuses on the how RBI addresses the issues identified in 'A Note', and on its potential as a new approach to distributed objects.

RBI is an effective solution to latency of fine-grained calls. These benefits derive from the use of remote evaluation, which executes any number of fine-grained calls in a single round trip. But RBI provides a familiar high-level programming model that hides the complexity of remote evaluation. The net effect is that there is no need for Remote Façades or Data Transfer Objects to combat latency. The batch statement gives programmers a clear performance model: one round trip per lexical batch statement.

Our approach to RBI is based on strict interfaces: a client can only access the operations in the public service interface of the server.

RBI avoids problems relating to memory access by eliminating the need for remote pointers. Only primitive data types are transferred between the client and server. Any intermediate objects that are accessed or created on the server are used on the server within the batch, and then discarded when they are done. This makes sense, because the intuitive view of a batch is that it specifies some computations to be performed on a remote server, but when it is running the 'remote' code is actually executing locally on the server. Only the visible outputs of these computations can be seen by the client. RBI allows the server to be stateless, in the sense that it does not have to preserve state about clients after a batch executes. There is no need for distributed garbage collection.

To transfer a complex object, like a hash table, from client to server (or vice-versa), the batch must create a new object and then copy the primitive values from one object to another, using the public interface of the objects. This is a high-level form of serialization that also supports translation between disparate implementations. Future work on RBI will allow these serialization operations to be reused in a library.

More permissive memory access models are possible. The key point is that RBI supports a very clean and restrictive model in a natural way. From a philosophical viewpoint, there is also some benefit to transferring only primitive values. Complex data types and user defined data abstractions (classes and ADTs) can have very different semantics in different languages, making them difficult to transfer. Rather than implement complex data transmission, RBI moves more of the complexity into the remote evaluation script. We believe that this is a good approach because simple expressions and statements can be executed in nearly any programming language: Expressions (calls, sequences, conditionals, loops) are common to all programming languages. But data structures, and especially complex data types and abstractions, cannot be transferred easily.

For logical exceptions, our current implementation of RBI mimics conventional programming language semantics; exceptions transfer execution to the closest enclosing try/catch block on the client. In previous work, we have explored other options such as allowing for re-execution of remote operations or rolling back already executed remote operations. We are not sure yet how important these options are and how they would be expressed in our language.

RBI is different from mobile code. With RBI, the code to be executed contains both *local* and *remote* code. The block is partitioned to separate the local and remote code. RBI also identifies all communication needed to connect them. Mobile code alone does not do this.

RBI is often compared to asynchronous remote method invocation because they both address the issue of latency in calling remote methods. We view synchronicity of the remote calls as orthogonal to batching. Our current implementation uses synchronous method invocation partly because it is built on Java RMI. However, a remote batch as a whole could be executed asynchronously in the same way individual remote method calls can be executed asynchronously. Another interesting way to combine RBI with asynchronous execution is to optimize the remote server script produced by RBI. Similar to the work on asynchronous methods calls, the remote server script could be parallelized by an interpreter or compiler.

5. CONCLUSIONS

This essay has discussed Remote Batch Invocation (RBI), a new programming construct that enables greater expressiveness in creating efficient distributed object systems. RBI retains the usability advantages of RPC-based programming abstractions for distributed computing, while eliminating or significantly improving on their limitations. This indicates that RBI may constitute a revolutionary step in the progression of programming technologies that use objects to tame the complexity of constructing and maintaining distributed systems.

Although only the future will tell whether RBI will enjoy wide adoption among distributed system programmers, we would like to ensure that tool vendors have a chance to seriously consider the advantages of RBI. To that end, we have made our reference implementation freely-available on the web for experimentation and porting to different languages and environments. Although the reference implementation is in Java and uses RMI as its transport, RBI can be implemented in any object oriented language and can use a variety of transport mechanisms. We hope that RBI will make it into the toolset of professional programmers charged with the challenges of creating the ever more sophisticated distributed systems of today and tomorrow.

Availability:

The reference implementation of RBI can be downloaded from: `http://research.cs.vt.edu/vtspaces/best`

6. REFERENCES

[1] M. Alt and S. Gorlatch. Adapting Java RMI for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2005.

[2] Amazon.com. Amazon associates web services.

[3] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.

[4] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. *ACM SIGPLAN Notices*, 29(10):341–354, 1994.

[5] L. Cardelli. A language with distributed scope. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 286–297, New York, NY, USA, 1995. ACM.

[6] K. Cheung Yeung and P. Kelly. Optimising Java RMI Programs by Communication Restructuring. In *ACM Middleware Conference*. Springer, 2003.

[7] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, 1998.

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24:342–361, 1998.

[10] D. A. Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, Cambridge, England, 1997.

[11] A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In *The 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, July 2009.

[12] C. V. Lopes. Adaptive parameter passing. In *ISOTAS '96: Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, pages 118–136, London, UK, 1996. Springer-Verlag.

[13] R. Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[14] T. Murphy, V. Karl, C. R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *In Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS*, pages 286–295. IEEE Press, 2004.

[15] The Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, 1997.

[16] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 211–249, London, UK, 1995. Springer-Verlag.

[17] U. Saif and D. Greaves. Communication primitives for ubiquitous systems or RPC considered harmful. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 240–245, 2001.

[18] R. Srinivasan. RFC 1831: RPC: Remote procedure call protocol specification version 2, 1995.

[19] J. W. Stamos and D. K. Gifford. Implementing remote evaluation. *IEEE Trans. Softw. Eng.*, 16(7):710–722, 1990.

[20] A. S. Tanenbaum and R. v. Renesse. A critique of the remote procedure call paradigm. In *EUTECO 88*, pages 775–783. North-Holland, 1988.

[21] E. Tilevich and Y. Smaragdakis. Portable and efficient distributed threads for Java. In *ACM Middleware Conference*, pages 478–492. Springer-Verlag, Oct 2004.

[22] S. Vinoski. RPC Under Fire. *IEEE INTERNET COMPUTING*, pages 93–95, 2005.

[23] J. Waldo, A. Wollrath, G. Wyant, and S. Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1994.

[24] J. E. White. RFC 707: High-level framework for network-based resource sharing, Dec. 1975.

[25] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the javatm system. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 17–17, Berkeley, CA, USA, 1996. USENIX Association.