Performance Implications of Design Alternatives for Remote Procedure Call Stubs

Sung K. Chung, Edward D. Lazowska, David Notkin, and John Zahorjan

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

Remote procedure call (RPC) has become widely adopted as a distributed programming paradigm. An RPC facility provides a user-level mechanism across a communication network that, as much as possible, has the same syntax and semantics as local procedure calls within a high-level language. Hence, RPC supports communication among application programs while relieving programmers from concern with the mechanisms used, such as data encoding and transport protocol.

Performance is critical in an RPC facility. To date, essentially all examinations of RPC performance have concentrated on kernel-level issues. In the current paper, we take efficient kernel-level support as a "given", and study the performance implications of design alternatives one level up – in the stubs, which insulate the client and server from details about network communication. The design alternatives that we consider were motivated by our three-year experience in the University of Washington's Heterogeneous Computer Systems (HCS) Project. These alternatives represent a collection of well-motivated approaches to achieving "standard" RPC semantics. In particular, we consider the performance implications of compiled vs. interpreted stubs, procedural vs. inline code for moving data to/from packet buffers, block copy vs. individual data item copy for moving data to/from packet buffers, and the presence or absence of byte swapping.

Index Terms – Remote procedure call, distributed system, performance.

1. Introduction

Remote procedure call (RPC) has become widely adopted as a distributed programming paradigm since its introduction by Birrell and Nelson in the early 1980s [Birrell & Nelson 1984]. An RPC facility provides a user-level mechanism across a communication network that, as much as possible, has the same syntax and semantics as local procedure calls within a high-level language. Hence, RPC supports communication among application programs while relieving programmers from concern with data encoding, transport protocol details, etc.

This material is based upon work supported by the National Science Foundation (Grants No. DCR-8352098, DCR-8420945, CCR-8611390, and CCR-8858804), U S WEST Advanced Technologies, Digital Equipment Corporation (the External Research Program), Bell Communications Research, Boeing Computer Services, Tektronix, the Xerox Corporation, and the Weyerhauser Company. This work was done while Zahorjan was on sabbatical leave at Laboratoire MASI, University Paris 6.

Authors' addresses: Department of Computer Science FR-35, University of Washington, Seattle WA 98195.

To a first approximation, an RPC facility works as follows: The client (caller) and server (callee) modules are programmed as if they were intended to be linked together. A description of the server interface – that is, the names of the procedures that the server implements and the types of their arguments – is processed, yielding two *stubs*. The client stub is linked with the client; to the client this stub is indistinguishable from the server. The server stub is linked with the server; to the server this stub is indistinguishable from the client. A subsequent client call to the server is fielded by the client stub, which transmits the arguments of the call to the server stub. The server stub then actually invokes the server, receives any results, and passes them back to the client via the client stub. In this way, the two stubs shield the client and server from the details of network communication.

In this paper, of several activities performed by stubs, we are especially interested in marshalling operations. Marshalling is essentially the movement of user data (input and output parameters) to and from packet buffers. But marshalling is more than data copying and packetizing. Since RPCs handle typed data, marshalling operations should be able to understand data types and representations, and do data translations when needed.

Performance is critical in an RPC facility. To date, essentially all examinations of RPC performance have concentrated on kernel-level issues. Work in this vein includes the design of transport protocols that are particularly suited to supporting RPC semantics on a reliable local network [Birrell & Nelson 1984], the extension of the transport protocol and of RPC semantics to support the efficient transfer of bulk data [Gifford & Glasser 1988; Liskov et al. 1987], and the detailed optimization of aspects such as thread context switches [Schroeder & Burrows 1988].

In the current paper, we take efficient kernel-level support as a "given", and study the performance implications of design alternatives one level up – in the stubs. This investigation has several key motivations. First, several of the design alternatives make it significantly easier to construct certain kinds of distributed services. Second, improvements in kernel-level performance have made previously insignificant stub costs of greater concern; as very high speed networks become common, the ratio of transport times to stub costs will decrease even further. Third, even for current RPC systems, performance for calls with a large volume of arguments can be bounded by the marshalling, not the transport, costs.

The specific design alternatives that we consider were motivated by our three-year experience in the University of Washington's Heterogeneous Computer Systems (HCS) Project [Notkin et al. 1988]. The HCS Project designed and prototyped techniques for reducing the cost of interconnecting heterogeneous computer systems. The specific areas that we attacked were communication, naming, filing, remote computation, and electronic mail. The demands of the latter four services in a heterogeneous environment highlighted a number of tradeoffs between flexibility and efficiency in the first service – our HCS Remote Procedure Call (HRPC)

facility [Bershad et al. 1987]. It is these tradeoffs that we will be examining here.

This paper is *not* of the "Stone Soup" variety frequently encountered these days in the RPC domain: "RPC semantics are nice, but they'd be even nicer if they included { asynchronous communication, multicast, ... }." The design alternatives that we consider represent a collection of well-motivated approaches to achieving "standard" RPC semantics. To be specific, we consider the performance implications of the following stub design alternatives:

compiled vs. interpreted stubs

In a compilation-based approach, each interface has "customized" client and server stubs that are based on the server's interface description; generally, these stubs are produced by a stub generator. In an interpretation-based approach, the system provides a single generic client and a single generic server stub, each of which is parameterized by the server's interface description; individual remote invocations pass actual parameters to these stubs that represent the remote procedure's argument types.

Compiled stubs have the advantage of run-time efficiency. Interpreted stubs have the advantage of flexibility. This flexibility arises in two ways. First consider what happens when a service description is changed. With compiled stubs, the stubs must be regenerated and then the client and server must be recompiled and relinked. With interpreted stubs, this added development cost is eliminated or reduced, depending on the actual change to the service description. Second, and perhaps even more important, is that interpretation makes it easier to construct generic clients and servers that can select the appropriate interface at run time. One example of this situation is the HCS Name Service [Schwartz, Zahorjan & Notkin 1987], in which a client using the name service must first make a call to a central server and then, based on the response of the first call, to an application-specific server. If interpretation had been available, the central server could have made the second call on behalf of the client. Another example of the benefit of interpretation is an HCS "bridge server," which acts as a translator between two incompatible RPC systems. Using compiled stubs requires that a separate bridge be generated for each service interface, while interpretation allows a single system-wide bridge server to be written.

procedural vs. inline code for moving data to/from packet buffers

In the procedural approach, a stub transfers each individual data item to or from a packet using a separate procedure call. For instance, a remote procedure with two integers as input arguments and a third as an output argument would use three procedure calls for data movement. In the inline approach, the code to perform such transfers are in the main body of the stub, eliminating the overhead of a procedure call for each data item.

The inline approach improves performance, especially given the cost of procedure calls. However, the procedural approach has two potential benefits. One benefit is that generating inline code can be difficult, especially in the absence of a compiler with the appropriate support. The other benefit is that calling the procedure indirectly allows great flexibility, especially with respect to handling heterogeneity. For instance, a stub can invoke a procedure to marshal an integer without concern for whether the integer is copied directly or has its bytes swapped. Hence, a single stub can be generated and used for RPC facilities that have significantly different data representations by linking the stub with appropriate marshalling routines.

individual data item copy vs. block copy for moving data to/from packet buffers

Elements in RPC arguments, such as fields or array elements, can be treated as individual items. Alternatively, when the items are to be copied to/from contiguous locations, a smaller number of block copies can be used to copy all the needed elements.

The cost of executing a stub can be reduced significantly using block copy, since it is generally cheaper to move bytes using fewer copy instructions. However, treating each element as an individual item makes it easier to write both a stub generator and a stub interpreter. Additionally, it may be difficult to identify the situations in which block copy can be used, since the layout of the data is compiler-dependent.

presence or absence of byte swapping

When two heterogeneous machines are communicating using RPC, data translation is often necessary. The most common form of translation is byte-swapping, which is needed when transferring integers between VAXes and SUNs, for instance. Most RPC systems define a standard "on-the-wire" representation for integers, letting each stub translate its data as needed. In such systems, if both the client and the server use the same representation, but they differ from the standard, then two unnecessary swaps are done.

When byte swapping is required, data items like integers cannot be directly copied. So this design choice further divides the *individual data item copy* mentioned above into *item(-by-item) copy* and *byte(-by-byte) copy*. And the need for byte swapping prevents the use of block copy.*

In each of these dimensions, we are interested in determining rules of thumb that give some feel for the comparative costs. For instance, does the added flexibility of interpretation cost us a factor of two or a factor of ten relative to compilation? Answers to these questions should give guidance to the designers of future RPC facilities.

The remainder of the paper is organized as follows. Section 2 describes our experimental methodology. Section 3 presents the measurement results and our analysis. Section 4 provides a brief conclusion.

2. The Experiment

The relative performance of the alternatives depends on the characteristics of the particular interface under consideration; for example, the advantage of inline data movement over procedural data movement will increase with the number of arguments in the interface. We . were thus confronted with a "workload characterization" problem. Our approach is to consider three different procedures that represent plausible extremes, and thus will indicate how well each stub design alternative does in the face of particular styles of data composition. The first procedure has a single argument: a single (long) integer. The second procedure is passed an array of 256 integers. The third procedure is passed six arguments: the first three are integers, the fourth is an array of 256 integers, the fifth is an array of 86 records each of which contains three integers, and the sixth is a record containing an integer and a string of 1024 characters. In the notation of our RPC interface

* Actually, the possibility of direct data item copy or block copy with byte swapping depends on the instruction set of a processor or machine level support for such data movement.

description language, we denote these three interfaces as:

1. (I)
2. (A[256] I)
3. ((I) (I) (I) (A[256] I) (A[86] (I I I)) (I S<1024>))

Having settled on this workload characterization, we then implemented a collection of stubs that differed from one another only in the alternatives outlined at the end of the previous section. Some combinations of alternatives were omitted because they don't make sense; block copy in concert with byte-swapping is an example. Table 2 in the next section shows the specific combinations of alternatives that we explored.

The primary subdivision in our experiments is that of compiled vs. interpreted stubs. The compiled stubs are all based on the stubs we use in HRPC (which are produced by a stub generator created by modifying Cornell's Courier stub compiler [Johnson 1985]). The HRPC stubs use procedure calls for each data item, but are not bound with respect to byte-order. These stubs were manually modified to cover the sub-cases we wished to explore.

The interpreted stubs were based on a straightforward interpreter adapted for these experiments from one developed by Kimiko Gosney for a Franz Lisp version of HRPC [Gosney 1987]. In addition to the arguments, the interpreter accepts a short-form description of the types of the arguments, the form of which is similar to the interface description given above.

For all our measurements, we eliminated calls made to the transport layer, since the costs of transport do not change in the face of the alternatives. Although the transport costs cannot be ignored when measuring the full cost of RPC facilities, eliminating the transport calls represents the bounding condition of increasing network speeds increase and the associated decreasing transport costs.

Measured in this experiment are the times to drive the marshalling operations. Marshalling times are inherently deterministic. Careful repeated measurements were made on a MicroVAX-II/Ultrix system.

3. Measurement Results and Analysis

3.1. A Simple Example

We begin this section with a simple example to illustrate our approach. The example, which appears in Table 1, is a small excerpt of our full measurement study, which appears in Table 2. In particular, the example considers four different stub designs (versus ten in the full study) and one procedure (versus three in the full study). Specifically, Table 1 shows measured marshalling costs for compiled stubs with either single-item copy or byte copy of arguments into packets, and with either inline or procedural code for this data movement. The procedure has a single integer argument: (I) in our notation. Thus, the stub must move four bytes of data and check and update buffer pointers.

Table 1 shows that for the system under consideration (a MicroVAX-II running Ultrix 2.0), using compiled stubs for a procedure with a single integer argument:

- Marshalling costs vary by a factor of four, from 10μs to 40μs, depending upon the stub design chosen.
- The cost of procedural stubs over inline stubs is significant (25μs, roughly tripling the inline marshalling cost) and is independent of other choices made.
- The cost of byte copy stubs (used if byte swapping is necessary) over item copy stubs is not nearly so significant (5μs); it too is independent of other choices made.

Table 1: Example Measurement and Analysis Excerpted from Table 2

	Inline	Procedural	Procedure Call Overhead
Byte Copy	15μs	40μs	25μs
Item Copy	10	35	25
Byte Copy Overhead	5	5	

This simple example was presented mainly to clarify our approach, in particular our focus on the *relative* performance of the design alternatives rather than on the *absolute* performance of our experimental software. It's also important to note that although our measurements are of course for a particular computer system, we will present parameterized equations at the end of this section that allow our results to be extended to other systems. Finally, it is essential to take the measurements for marshalling in the context of the cost of a round-trip RPC call, including transport. Original RPC facilities made invocations in roughly 50 milliseconds; our basic HRPC invocation with no arguments costs roughly 17 milliseconds; the best reported times for such basic calls in any RPC system are in the two millisecond range.

3.2. Principal Performance Comparison

Table 2 contains our principal measurement results. It shows the marshalling costs for each of the three procedures in our "workload", for each of ten different combinations of stub design alternatives. In addition to the measurement results for each of these 30 cases, Table 2 contains (in parentheses) an analytic estimate of the results, calculated from a simple model of RPC stub performance that we have constructed and that we will discuss later. Note that the results in Table 1 were excerpted from rows 6-9 in column 1 of Table 2's results.

We begin by exploring the performance difference between interpreted and compiled stubs. This difference reflects the extra overhead of interpretation. One obvious observation is that the amount of interpreter overhead is highly dependent on the data structure of the argument being marshalled. For a single integer (data type (I)), for instance, the interpreter overhead involves some initial setup and the time to decode a single type specification code, a total of roughly 70μs. This is not a large absolute value (especially in comparison to full round-trip RPC costs, which are measured in milliseconds, not microseconds) but turns out to represent a substantial proportion of the total marshalling time. For an array of 256 integers (data type (A[265] I)), additional time is required for the initial setup, but this time is amortized over all of the array elements, so the total interpreter overhead becomes insignificant. In fact, Table 2 shows that the compiled stubs were actually slightly slower than the interpreted stubs for this case (when procedural rather than inline data movement was employed) – a definite anomaly caused by the difference in the for-loops of the compiled stub and the interpreter, among other minor implementation details and measurement noises.

We can see a significant difference between compiled and interpreted stubs in marshalling the (Structure). But most of the difference is caused by a single portion of the (Structure): more than 90% of the extra interpreter overhead is attributable to the array of records, (A[86] (I I I)). Since the interpreter does not have *a priori* information about nested structures, such structured arguments are handled by recursively calling the interpreter. And each field of a

Table 2: Principal Comparison of Marshalling Costs (MicroVAX-II, Ultrix 2.0)

Stub Type			Data Type (I)	Data Type (A[256] I)	Data Type (Structure)†
Interpreted	Inline	Byte Copy	0.085 *ms* (0.083)††	4.75 *ms* (4.82)	23.3 *ms* (23.9)
		Item Copy	0.079 (0.078)	3.40 (3.53)	19.0 (21.3)
	Procedural	Byte Copy	0.108 (0.108)	11.2 (11.2)	35.7 (36.9)
		Item Copy	0.101 (0.103)	9.90 (9.96)	31.0 (34.3)
	Block Copy		0.079 (0.079)	0.585 (0.628)	15.7 (16.0)
Compiled	Inline	Byte Copy	0.015 (0.015)	4.60 (4.62)	12.1 (11.9)
		Item Copy	0.010 (0.010)	3.32 (3.34)	9.65 (9.32)
	Procedural	Byte Copy	0.040 (0.040)	11.3 (11.0)	28.9 (27.1)
		Item Copy	0.035 (0.035)	10.0 (9.73)	25.8 (24.5)
	Block Copy		0.035 (0.035)	0.457 (0.432)	4.08 (3.87)

† (Structure) ≡ ((I) (I) (I) (A[256] I) (A[86](I I I)) (I S<1024>))
†† () denotes analytic estimate of marshalling time

record is individually processed for each recursion. Such a combination of inefficient aspects of the interpreter, which is inherent to all interpreters to some extent, explains the large interpretation overhead shown in the measurement data.

Summarizing these observations concerning interpreted versus compiled stubs, Table 3, which is based on the data in Table 2, shows the percentage of total marshalling time due to interpretation for various interfaces and stub types. The key conclusion is that the relative cost of interpretation is significant for large, irregularly-structured arguments, but insignificant otherwise. (For small arguments, the absolute costs might be high, but these will still be small relative to the overall cost of an RPC invocation.) Interpreter overhead is determined by the target data structure and is almost independent of other design choices.

We next consider the performance difference between procedural and inline stubs. Since each elementary data item causes a procedure call in a procedural stub, the degradation due to a procedural stub structure is easily calculated as *number of elementary data items* × *procedure call time*. There are also additional procedure call overheads to handle structured data with procedural stubs; for instance, to marshal an array of N records, if a record is handled by a "factored" marshalling procedure, it takes N procedure calls in addition to those required for the marshalling of the record fields. We call the costs due to such additional overheads to process structured data "combining costs".

Table 4 is analogous to Table 3, except that it shows the percentage of total marshalling time due to procedure calls. Our general conclusion is that a procedural organization can be expected to degrade performance by a factor of about two, more-or-less independently of other design choices.

The performance difference between the byte copy method and the item copy method is not so great as for the other pairs of design alternatives. Of course, data copying time is highly dependent on implementation details and on the data copy instructions of the target machine; also, as was the case for procedure call overhead, data copying overhead is proportional to the data size. Nonetheless, our conclusion is that there is a negligible performance implication to this choice.

The conclusion for block copying of data (that is, the movement of large numbers of data items using a single instruction) is considerably different. Block copy, where feasible, can improve marshalling performance dramatically. There are two main reasons for this. First, with block copy, data copying itself is faster. Secondly, for block copying it is assumed that there is no need for byte swapping or concern about the internal data structure; thus there is no "combining cost" (assuming the size of a data structure is known).*

Block copy causes an interesting interaction between interpreted and compiled stubs. For compiled stubs all the data sizes for non-variable size structures are known at compile time. But for the interpreted case, as we have discussed earlier, such information is generally unavailable for user defined data structures. This lack of information of data sizes makes the interpreter inefficient for marshalling structured data, as shown in Table 2 for the two block copy cases for (Structure).

3.3. An Analytic Model

As noted earlier, we constructed a simple analytic model of stub costs – a linear equation involving several parameters.

* But sometimes the cost of knowing the size of a variable size data structure is unexpectedly high. For example, in the C language, to obtain the size of a 1024 byte string requires 2.7ms using strlen(), a standard string length function. String representations in other languages store the length explicitly, reducing this cost to a small constant.

Table 3: Percent of Marshalling Time Due to Interpretation
$\frac{\text{interpreter overhead}}{\text{marshalling time}}$

Stub Type			Data Type		
			(I)	(A[256] I)	(Structure)
Interpreted	Inline	Byte Copy	80%	4%	51%
		Item Copy	86	6	63
	Procedural	Byte Copy	63	2	33
		Item Copy	67	2	39
	Block Copy		86	4	76

Table 4: Percent of Marshalling Time Due to Procedure Calls
$\frac{\text{sum of procedure call times}}{\text{marshalling time}}$

Stub Type			Data Type		
			(I)	(A[256] I)	(Structure)
Interpreted	Procedural	Byte Copy	23%	57%	36%
		Item Copy	25	65	42
Compiled	Procedural	Byte Copy	63	57	52
		Item Copy	83	64	59

The construction of such a model has two objectives. First, if the results of the model match the measurement data, then we have confidence that our abstractions really do capture the essence of the phenomena we are trying to understand. Comparison of the parenthesized and non-parenthesized results in Table 2 shows that the match between measured and predicted results is excellent. Second, the model allows the measured results to be extended to other computer systems, removing the system dependence inherent in any measurement study.

The cost model for the data type (A[256] I) (an array of 256 integers) is shown in Table 5. Similar cost models can be generated (automatically, based on the interface description) for other data types. The definitions of the variables in the cost equations, along with the values of these variables for a MicroVAX-II running Ultrix 2.0, appear in Table 6. (The initial term in each expression in Table 5 represents the cost for sending the size of the array.)

We close with two caveats. First, as we have seen, procedure call cost and data copy cost are the major contributors to marshalling time. These costs are machine dependent. For example, a machine may have highly optimized procedure call that is particularly fast relative to data movement. In such a case, the benefits of inline stubs would not be as great as indicated in Table 1.

Second, we note that the generality of our measurements is of course limited. Nonetheless, we feel that we have demonstrated the relative performance of various approaches to stub design, and that the incremental performance differences can be parameterized in terms of a few key characteristics.

Table 5: Cost Model for (A[256] I)

Stub Type			Cost Equation
Compiled	Inline	Byte Copy	$L_b+256\times(L_b+f)$
		Item Copy	$L_i+256\times(L_i+f)$
	Procedural	Byte Copy	$L_b+p+256\times(L_b+p+f)$
		Item Copy	$L_i+p+256\times(L_i+p+f)$
	Block Copy		$L_i+b(256\times4)+\alpha$
Interpreted			Add $(B+A+e)$ to each entry above

Table 6: Parameter Values for a MicroVAX-II Running Ultrix 2.0

Symbol	Meaning	Value
L_b	move a long integer by byte copy	15μs
L_i	move a long integer by item copy	10
f	for-loop bookkeeping	3
p	call for a marshalling procedure	25
$b(n)$	move n bytes by block copy(bcopy), b(1024)	382
α	block copy stub initial overhead (per a packet size)	40
B	initial setup for the interpreter	45
A	initial setup for an array in the interpreter	128
e	processing an elementary type specification code in the interpreter	23

4. Conclusions

Our objective in this paper has been to assess the performance implications of certain design alternatives for remote procedure call stubs. The design alternatives that we considered provide "standard" RPC semantics with varying degrees of flexibility. This flexibility, while particularly valuable in a heterogeneous environment, can be important elsewhere as well. We have provided measurements and analysis that we feel will be useful to designers of RPC systems who must consider whether a particular "flexibility enhancement" is worth the performance compromise. Our results can be summarized as follows:

- Compiled stubs have a significant performance advantage over interpreted stubs only in the case of large, irregular parameters. However, the flexibility of interpretation may often lead to other

benefits of system structure.

- Inline data movement offers, for the machine we measured, a factor of two improvement over procedural data movement, almost independently of other design decisions. On a machine where procedure invocation is less expensive relative to basic copy operations, the benefit will still be significant, although perhaps less than a factor of two.
- Byte swapping is not particularly costly relative to single item copying of data to packets.
- Block transfer of data to packets, where possible, offers a very significant performance advantage over single item copying.

Acknowledgments

Henry Levy and Brian Pinkerton assisted with the measurements.

References

[Bershad et al. 1987]
Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering SE-13*,8 (August 1987), pp. 880-894.

[Birrell & Nelson 1984]
Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2*,1 (February 1984), pp. 39-59.

[Burrows & Schroeder 1987]
Mike Burrows and Mike Schroeder. Performance of Firefly RPC. DEC Systems Research Center, November 1987.

[Gifford & Glasser 1988]
David K. Gifford and Nathan Glasser. Remote Pipes and Procedures for Efficient Distributed Communication. *ACM Transactions on Computer Systems 6*,3 (August 1988), pp. 258-283.

[Gosney 1987]
Kimiko Gosney. Heterogeneous Remote Procedure Call for Franz Lisp. (M.S. thesis.) Technical Report 87-07-03, Department of Computer Science, University of Washington, July 1987.

[Johnson 1985]
J.Q. Johnson. XNS Courier under UNIX. Cornell University, March 1985.

[Liskov et al. 1987]
Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler, and William Weihl. Communication in the Mercury System. Programming Methodology Group Memo 59, Laboratory for Computer Science, Massachusetts Institute of Technology, October 1987.

[Notkin et al. 1988]
David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting Heterogeneous Computer Systems. *CACM 31*,3 (March 1988), pp. 258-273.

[Schwartz, Zahorjan & Notkin 1987]
Michael Schwartz, John Zahorjan, and David Notkin. A Name Service for Evolving Heterogeneous Systems. *Proc. Eleventh ACM Symposium on Operating Systems Principles* (November 1987), pp. 52-62.