Objects by Value: Evaluating the Trade-Off

André Spiegel

spiegel@inf.fu-berlin.de

Technical Report B–98–13
September 1998

Abstract

The single most important performance problem for distributed, object-oriented programs is that remote method invocations take several orders of magnitude longer than local invocations. If objects located on different computing nodes need to invoke each other frequently, performance can thus suffer intolerably. While this problem can be tackled by techniques like object migration and replication, such facilities are difficult to implement and even more difficult to standardize, which is why they do not yet exist in today's mainstream distribution platforms. Many of these platforms therefore resort to a simple, yet effective technique that can also lead to dramatic improvements in performance: passing objects by value. Implemented for example by Java/RMI and also proposed in a draft CORBA standard, this technique allows to create copies of parameter objects at remote sites, which the receiver can access in fast local invocations. In this paper, it is examined in detail when exactly it pays off to pass an object by value, using both empirical and analytical studies. A sound basis for evaluating the trade-off is thus provided, so that the programmer — or an automatic optimization tool — can make the right decision in a given scenario.

Keywords: Distributed programming, parameter passing, Objects-by-Value, Java, RMI, JavaParty, CORBA.

Freie Universität Berlin
Institut für Informatik
Takustraße 9
D–14195 Berlin, Germany

A shorter version of this report appears in Proc. PDCN '98, Brisbane, Australia, December 1998.

1 Introduction

Mechanisms for parameter passing, such as pass-by-reference and pass-by-value, are semantic concepts, but programmers have always been eager to exploit them for the sake of efficiency as well. For example, if a large array is passed to an object that only reads the individual elements, but is not supposed to change them, pass-by-value is semantically appropriate and safe — however, passing the array by reference avoids the expensive copying operation, and is therefore often used as an optimization.

There is a similar effect in distributed programming, and here the impact on performance is even more significant, which is why more and more distribution platforms provide mechanisms for passing objects either by reference or by value — even if the underlying programming language does not offer this distinction itself.

If an object is passed to a remote callee by reference, it usually means that the object remains physically at the site of the caller. The callee gets a network reference of the object, which it may then use to invoke the object remotely. However, if it needs to perform many separate invocations, the large overhead of each remote invocation can become a serious performance issue. Passing the object by value can be much more efficient in such cases: the entire state of the object is copied to the callee by means of serialization, and the callee may then access the copy through fast local invocations. Naturally, this only works if (a) it is semantically correct to pass the object by value, i.e. the callee only reads the object, and doesn't modify it, and (b) the object is not too large, because otherwise the time needed to transfer it may be longer than that for performing the invocations remotely.

The programmer is thus faced with a classical trade-off: if pass-by-value is semantically possible, then the two factors, object size and number of invocations, must be weighed against each other to see which passing mechanism will perform better. In this paper, we provide a sound basis for this evaluation, presenting both empirical and analytical studies of the problem. We will focus on Java-based platforms in these studies, because of their wide-spread use, but we believe that, qualitatively, our results are applicable to arbitrary programming languages and distribution platforms, especially to the upcoming CORBA facilities for passing objects by value.

The paper is organized as follows. In section 2, we will briefly look at various distribution platforms to see how the two passing mechanisms are realized on them. We will also consider alternatives to the two basic passing mechanisms. Section 3 describes a series of experiments that we made to determine the exact trade-off in a given situation. These results will be used to develop a more general, analytical model of the trade-off in section 4. Section 5 lists related work, and section 6 summarizes our findings.

2 Platforms

2.1 Java/RMI

RMI is Sun's proprietary distribution technology for the Java language [Gosling et al. 1996, Sun 1997]. It allows the programmer to create *remote* classes, objects of which can be invoked across distribution boundaries. Remote objects are represented by network references, and hence, always passed by reference. If a class is not remote, but implements the interface `java.io.Serializable`, then objects of this class are passed by value in invocations of remote objects, using Java's serialization mechanism (in local invocations, such objects are still passed by reference).

We may therefore say that in Java/RMI, passing mechanisms are determined on a *per-type* basis: it is the type of the formal parameter, and the type of the callee, by which it is determined whether an actual parameter is passed by reference or by value.

2.2 JavaParty

JavaParty [Philippsen 1997] is an alternative to RMI which handles distribution more transparently than RMI does. In JavaParty, almost all distribution-related issues are hidden from the programmer, so that distributed programming becomes very much like centralized programming. JavaParty is (currently) built on top of RMI, but realizes a number of features not found in RMI, e.g. remote variable access and, most notably, object migration.

Passing mechanisms in JavaParty work similar as in RMI. The programmer may create *remote* classes, objects of which are remotely invokable and always passed by reference. Objects of non-remote classes are passed by value in remote invocations, using serialization (JavaParty considers all non-remote classes serializable). It follows that in JavaParty, just as in RMI, it is the type of the formal parameter and the type of the callee by which the actual passing mechanism is determined.

2.3 CORBA

CORBA, the OMG's standard for distributed systems [OMG 1995], requires reference semantics for interface types, i.e. for CORBA objects defined in an IDL interface. However, the OMG has recognized that passing objects by value can lead to dramatic improvements of performance, and therefore an RFP "Objects by Value" has been issued [OMG 1996]. The most recent revised joint submission to this RFP [OMG 1998] proposes to add *value types* to CORBA, objects of which may be passed by value in CORBA invocations.

Unlike the other platforms described above, passing mechanisms are not determined on a per-type basis only, though. The proposal suggests that (a) value types may inherit from interface types, and (b) when an object that has a value type is passed to a method that expects an interface type, the object is passed by reference. In this respect, the CORBA proposal allows much more flexibility than the other platforms we looked at. Essentially, it will be possible to determine passing mechanisms *per parameter*, so that different methods that have parameters of a certain type may be optimized independently.

2.4 Beyond the Mainstream

While passing objects by value is attractive for performance reasons, the big drawback of this technique is that it can often not be used because the desired semantics does not warrant it. Pass-by-value is only applicable for *read-only* parameters, or more precisely, only for parameters where the caller does not depend on any modifications made by the callee. What is often needed is a means to bring objects "close" to each other while still maintaining standard object reference semantics. Such techniques have been proposed and demonstrated many years ago, although today's mainstream technologies like Java/RMI and CORBA do not currently support them.

Object *migration*, as shown in the Emerald system [Jul et al. 1988], is one approach. In Emerald, an object can be passed to a remote site *by move* or *by visit*, which means that the run-time system moves the object physically into the callee's address space, and re-directs any subsequent invocations that reference the old location. (In the case of pass-by-visit, the object is moved back after the invocation returns.) Reference semantics is thus preserved.

If an object is accessed from several different locations concurrently, *replicating* the object may be a better alternative, meaning that separate copies are kept at each site where the object is needed, and the run-time system ensures that the copies are synchronized. Orca [Bal et al. 1992] is an example for a platform where replication is utilized.

In a more general sense, all these techniques can be seen as variants of *object caching*, i.e. they apply techniques that have traditionally been used for memory management at the hardware level, to objects at the application level.

By comparison, passing objects by value is a rather simplistic technique, which suffers most from its limited applicability. We do not believe, however, that migration and replication facilities will completely *replace* pass-by-value in the foreseeable future. This is because in the limited realm where it *can* be applied, it is faster than any of the other options we listed, and it is therefore likely to survive as one of many optimization techniques that future distribution platforms will provide.

In the remainder of this paper, we will therefore restrict our discussion to the choice between the two basic passing mechanisms, assuming that we do have a situation where either can be used without affecting the program's semantics.

3 Experiments

For the experiments described below, we used JavaParty as our distribution platform. In the given scenario, JavaParty does not impose any noticeable overhead over RMI, so the results are applicable to RMI as well. While it is difficult to foresee the characteristics of the upcoming implementations of CORBA "Objects by Value", our results will also be instructive for these systems in a qualitative sense.

We will first describe the general setup of the experiment, and then the concrete technical realization.

3.1 Setup

We set up a scenario where an object, the *caller*, invokes another object, the *callee*, passing a *parameter object* to it both by reference and by value (Fig. 1). The callee performs n separate invocations on the parameter object, and then returns. The parameter object encapsulates an array of m small data items, for which we used both Java `int`s and simple `struct`-like objects that contain a single integer value each. By varying both parameters, m and n, and measuring the total time until control returns to the caller, we evaluated which passing mechanism performs better under what circumstances.

There are a number of assumptions implied by this setup:

- The scenario is *synchronous* in nature. We do not consider the case where either the caller or the callee do any other work while data transmission is in progress, or

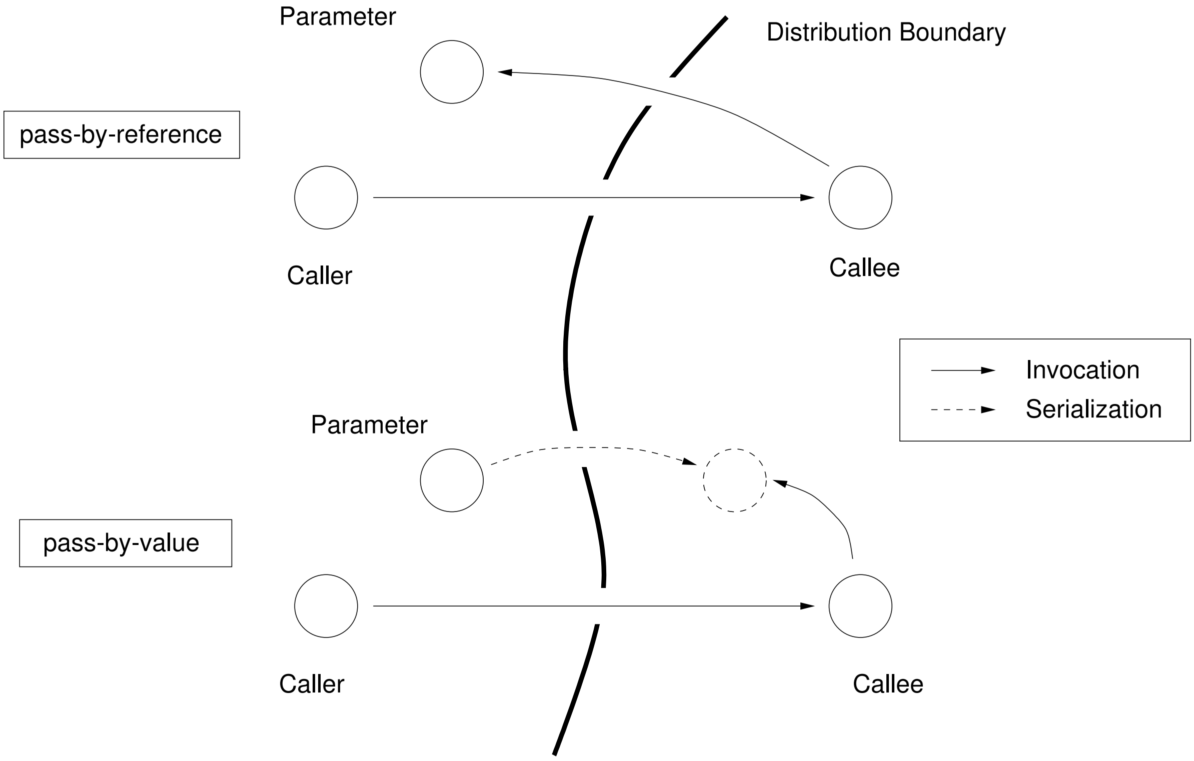


Figure 1: Setup for the Experiments

while the corresponding party is working. (Note that this only concerns parallelism at the *application level*; we will in fact consider parallelism at the system level.)

- We assume that the *code* of all objects is already present where it is needed. It is therefore only the data that needs to be passed.

- We assume that the invocations which the callee performs on the parameter object carry a *negligible amount of data* themselves. In the experiments, we used simple *ping*-type invocations with no parameters at all, and in separate experiments we confirmed that invocations with only a few parameters of primitive types do not take measurably longer than this.

- We assume that both machines, the one on which the caller resides, and the one on which the callee resides, have *equal performance*. Given this assumption, it is irrelevant how much work is actually done by the parameter object's methods, because it would show up as an identical constant added to the times for both passing mechanisms. In the experiments, we therefore used true *ping*-type invocations that didn't do any actual work, but returned immediately.

The following two observations were used to simplify the experiments, without any impact on the results:

- If a parameter object is passed by reference, the actual size m of the object is not important, and neither does it matter whether the object encapsulates an array of integers, or of structs. Hence, we only varied the number of accesses n in these cases.

- On the other hand, if a parameter object is passed by value, the number of invocations n is close to irrelevant, because these local invocations are two to three orders of magnitude faster than the single remote invocation that is needed to transfer the object to the callee in the first place. For this reason, only the size m of the objects was varied when passing by value.

3.2 Technical Details

We performed the experiments on different hardware platforms, and also used different transmission media, to study the impact of these on the results. The machines we used were an IBM Thinkpad 365X (P100, 24 MB) running under Linux 2.0.33, a SparcStation 10 (20 MHz, 32 MB, Solaris 2.5), and two UltraSparcs (166 MHz, 128 MB, Solaris 2.6).

All SparcStations were connected via 10 Mbps Ethernet (same physical segment), while for the IBM Thinkpad, we used both an Ethernet card and a modem to study the impact of different network speeds. The modem line "simulates" the speed of today's long-haul internet connections, and it has the advantage that the transmission rate can be varied, which will be important for our analyses.

In the experiments involving the Thinkpad, it was in the role of the caller, and the SparcStation 10 was the callee. While this may seem like a violation of our assumption that both machines be equal, we found that on the Ethernet, Thinkpad and Sparc 10 produced almost the same results as two Sparc 10 machines, so we may indeed consider these machines equal for our purposes.

The Thinkpad's Ethernet card was connected to the same physical segment as the Sparc 10, while the modem connection was made using PPP and physically went through a modem on the receiving side, a terminal server, a router, and the SparcStation's Ethernet adapter; however these additional hops do not measurably influence the speed dictated by the modem connection. The modem was set to different, fixed transmission rates, with V.42 error correction activated but no data compression, so that a constant transmission rate was achieved.

The distribution platform we used was JavaParty version 0.95, operating on top of JDK 1.1.3 (Linux port and native Solaris version, with JIT). All experiments were performed at times when there was little other activity on the network, and none on the machines themselves. Timings were obtained by calling the Java method `System.currentTimeMillis()`. Each experiment was performed once to warm the processor cache and allow JavaParty/RMI to initialize, then repeated fifty times, and the results averaged.

3.3 Results

We show results for four different system configurations: (a) the Thinkpad connected to the Sparc 10 using the modem at 4800 Bps and (b) 28800 Bps, (c) using the Ethernet card, and (d) the two UltraSparcs connected via Ethernet. Figures 2 and 3 show the results for passing an object that encapsulates an array of length m by value: in figure 2, the array elements are `int`s, and in figure 3, the array elements are `struct`-like objects that contain a single integer value each. Figure 4 shows the times for passing the parameter by reference, with the callee performing n separate remote invocations of the parameter object.

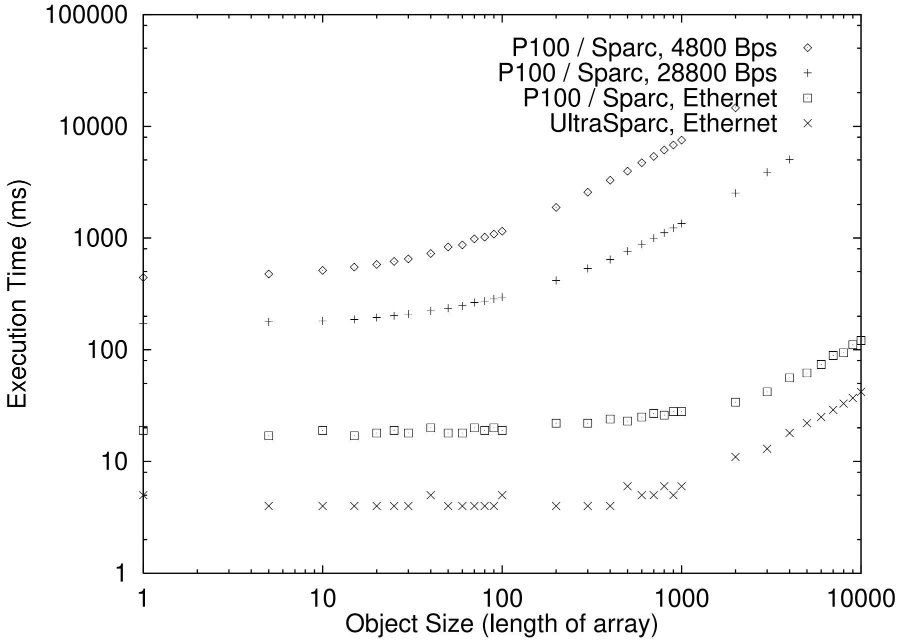


Figure 2: Passing Objects by Value (int)

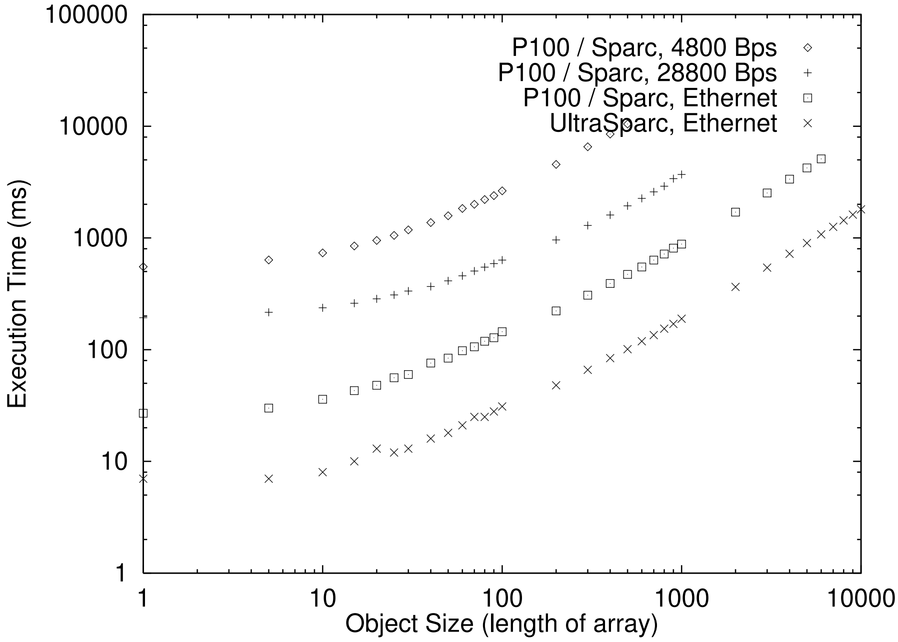


Figure 3: Passing Objects by Value (struct)

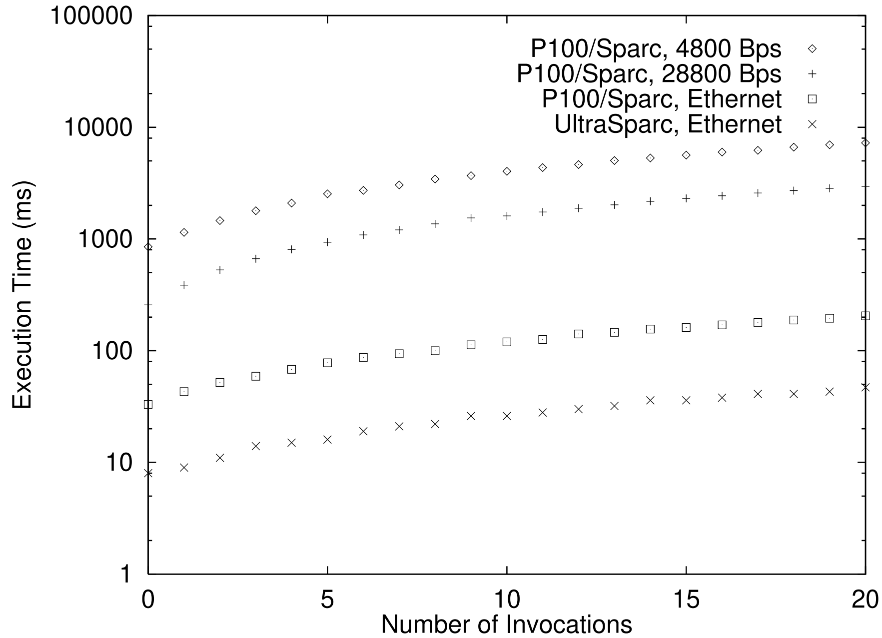


Figure 4: Passing Objects by Reference

hardware	media	data	pass-by-value	pass-by-reference
P100 / Sparc	4800 Bps	int	$t_{v_i}(m) = 447.7 + 7.082\ m$	$t_r(n) = 833.7 + 320.5\ n$
		struct	$t_{v_s}(m) = 580.8 + 19.84\ m$	
	28800 Bps	int	$t_{v_i}(m) = 164.3 + 1.218\ m$	$t_r(n) = 264.0 + 135.6\ n$
		struct	$t_{v_s}(m) = 235.2 + 3.434\ m$	
	Ethernet	int	$t_{v_i}(m) = 19.62 + 0.00926\ m$	$t_r(n) = 34.85 + 8.479\ n$
		struct	$t_{v_s}(m) = 42.35 + 0.8383\ m$	
UltraSparc	Ethernet	int	$t_{v_i}(m) = 4.484 + 0.00370\ m$	$t_r(n) = 7.752 + 1.907\ n$
		struct	$t_{v_s}(m) = 9.567 + 0.1786\ m$	

Table 1: Regression Analysis (times in milliseconds)

hardware	media	data	trade-off
P100 / Sparc	4800 Bps	int	$m < 54.5 + 45.3\ n$
		struct	$m < 12.7 + 16.2\ n$
	28800 Bps	int	$m < 81.9 + 111.3\ n$
		struct	$m < 8.4 + 39.5\ n$
	Ethernet	int	$m < 1644.0 + 916.0\ n$
		struct	$m < -8.9 + 10.1\ n$
UltraSparc	Ethernet	int	$m < 829.0 + 515.0\ n$
		struct	$m < -11.3 + 10.7\ n$

Table 2: Trade-Offs

All results are very close to linear, as could be expected (this may not be apparent in the figures because of the logarithmic scales). On the one hand, the time for passing the object by value is linear in the size of the object m; we did not measure any discontinuities due to the size of network packets. The time for pass-by-reference, on the other hand, is linear in n, the number of invocations. We may therefore summarize the values through linear regression analysis, the results of which are given in table 1.

Using these equations, we may now calculate the trade-off between the two passing mechanisms. It is faster to pass an object by value if $t_v(m) < t_r(n)$. Using the first row of table 1 as an example, we get

$$447.7 + 7.082\ m < 833.7 + 320.5\ n.$$

Solving for m yields

$$m < 54.5 + 45.3\ n.$$

Applying this calculation to all of our results, the inequalities in table 2 are obtained, which are shown graphically in figures 5 and 6. The lines in these figures indicate where both passing mechanisms have equal performance. The area to the upper left of each line represents situations where pass-by-reference is better (larger objects, accessed infrequently), while to the lower right, it is better to pass by value (small objects that are accessed more often).

3.4 Discussion

The first thing to be observed in figures 5 and 6 is that for typical fine-grained application objects, pass-by-value is always the better choice (provided that it is semantically possible). An object that encapsulates integers must contain at least 100 of them for pass-by-reference to pay off (P100 / Sparc, 4800 Bps, 1 invocation). Structs are more difficult to serialize, therefore the value is lower for structs, but even here, if the callee performs only two invocations, the object must already contain at least ten struct objects for pass-by-reference to be better (UltraSparc, Ethernet).

On the other hand, the figures indicate that if an object contains a large data structure, and is not invoked very often by the callee, then passing the object by reference should be carefully considered.

What is interesting in the figures is how the trade-off varies for the different hardware platforms and transmission media. Figure 5 shows that pass-by-reference becomes increasingly unattractive as the transmission rate gets higher. This seems natural because a single remote invocation in which a large amount of data is passed benefits more from the increased transmission rate than the many small remote invocations which pass-by-reference implies.

However, when the P100 / Sparc hardware is replaced with the much faster UltraSparcs, pass-by-reference gets *better* again. Table 1 shows that compared to P100 / Sparc (Ethernet), pass-by-reference is 4.4 times faster on the UltraSparcs, while for passing integers by value, only a factor of 2.5 is achieved. Obviously, this is due to the fact that the Ethernet transmission rate begins to be the limiting factor for passing integers by value — the UltraSparcs *saturate* the network in this case, while pass-by-reference is more computation-intensive, and therefore still has a higher potential for optimization.

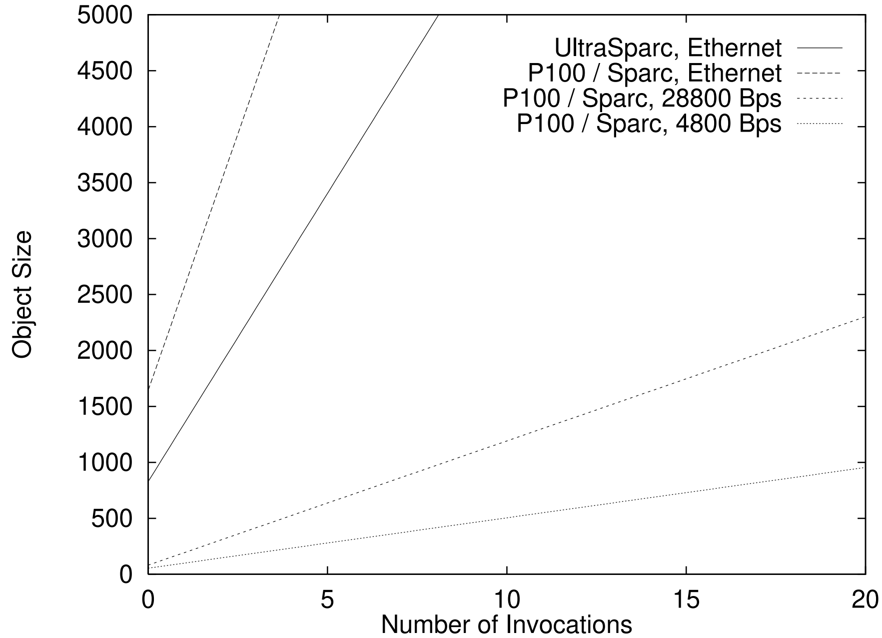


Figure 5: Trade-Off for Objects Encapsulating Integers

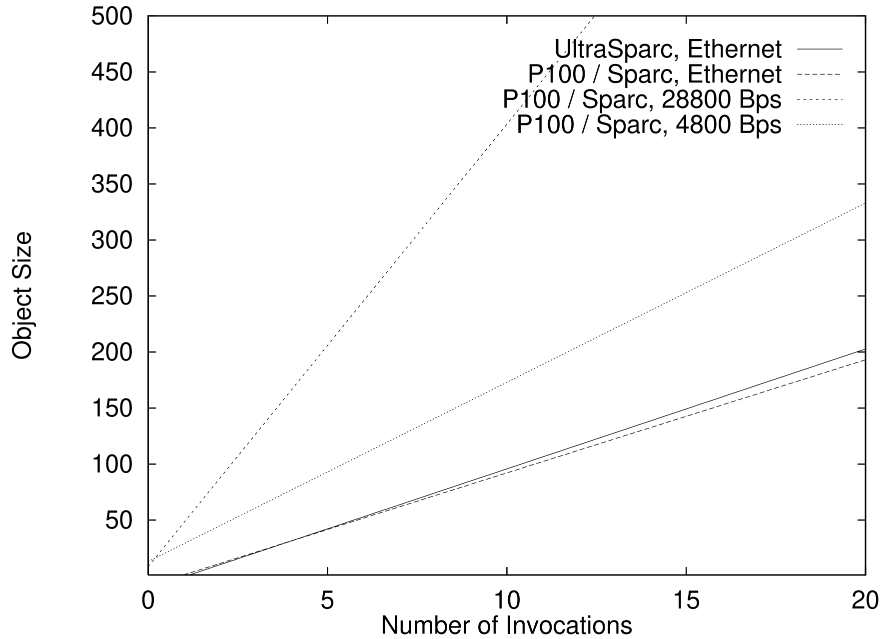


Figure 6: Trade-Off for Objects Encapsulating structs

(The reader should note that the above effect does not generally occur when the hardware gets faster but the transmission rate remains equal. We also measured and calculated the tradeoff for two Sparc 20 machines on the Ethernet, and in this case, the line simply moves to the upper left of P100 / Sparc: pass-by-reference gets 1.3 times faster on the Sparc 20, while for passing integers by value the factor is 2. In other words, a Sparc 20 does not, like the UltraSparc, saturate the Ethernet.)

Another phenomenon that needs explanation is apparent in figure 6. While it is natural that the tradeoff for structs occurs at much lower object sizes than for integers (because structs are more difficult to serialize), it is noteworthy that for both Ethernet configurations, the lines are *below the modem lines.* Looking at table 1, we find that with respect to P100 / Sparc at 28800 Bps, pass-by-reference is 16 times faster on P100 / Sparc (Ethernet), and 70 times faster on the UltraSparcs. For passing structs by value, however, the factors are only 4.1 and 20, respectively. To understand this phenomenon, we need some more insight into the mechanisms that determine the tradeoffs.

4 An Analytical Model

4.1 Developing the Model

We have seen in the experiments that the execution times for both passing mechanisms can be modeled as linear equations. For passing by reference, the equation is

$$t_r(n) = T_r + t_i \cdot n \tag{1}$$

where t_i is the time for the callee performing an individual remote invocation of the parameter object, and T_r is the constant time that does not depend on the number of invocations n. A similar equation models pass-by-value:

$$t_v(m) = T_v + t_e \cdot m \tag{2}$$

where t_e is the time for transferring a single data element to the callee, and T_v is the constant time that does not depend on the size of the object. Note that these coefficients have different values when the parameter encapsulates integers vs. structs; if we need to distinguish these, we will use further indices, like T_{v_i}, T_{v_s}, etc.

Calculating the trade-off $t_v(m) < t_r(n)$ and solving for m (as we did for concrete values in the previous section), we get

$$m < \frac{T_r - T_v}{t_e} + \frac{t_i}{t_e}\, n. \tag{3}$$

We will now examine how the parameters of this equation are affected when either computing power or network transmission rate changes. To achieve this, a simple model of what happens during a remote method invocation is required. Figure 7 identifies three distinct phases: first there is a *sending* phase, where the remote invocation message is assembled and passed to the hardware (this includes tasks like looking up the network address of the callee's machine, serializing parameter objects, and handing the message down through the protocol stack). Once the message has reached the network device, the *transmission* phase begins, which covers the physical transfer of the message to the callee's machine, up to the point where the network device has received all the data, and passes it to the software. The *receiving* phase then consists of tasks like passing the message up through the protocol stack, deserializing any parameters, and invoking the target method on the callee.

Assuming that the distribution middleware does not change, then the time for the *send* and *receive* phases is influenced only by the computing power of the machines, while the length of the *transmission* phase depends on the speed of the network only.

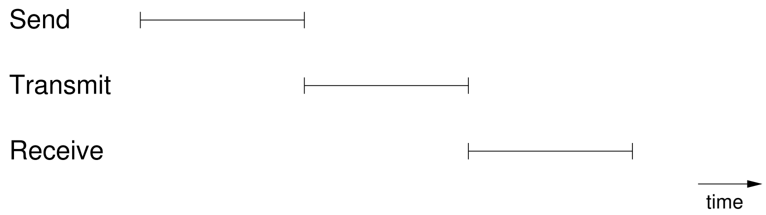


Figure 7: Sequential Model of a Remote Invocation

The model simplifies reality insofar as it doesn't account for any *replies* sent by the callee, like transport layer acknowledgements, or the message that signals the return of control to the caller, which might also carry result parameters with it. However, we can easily subsume these under the three phases indicated, since we are only concerned with the total time that each phase takes.

We must be aware, however, that the three phases might also *overlap* (fig. 8). This occurs if the information transferred is so large that it is split into several network packets, *and* if the distribution middleware is written so that this kind of parallelism may take place (RMI is an example for a platform that exhibits it). It is obvious that in these scenarios, the total time for a remote invocation is determined by the phase (or "task" in the parallel case) that takes longest to execute (marked by the broken lines in figure 8), plus a certain overhead for "filling the pipeline".

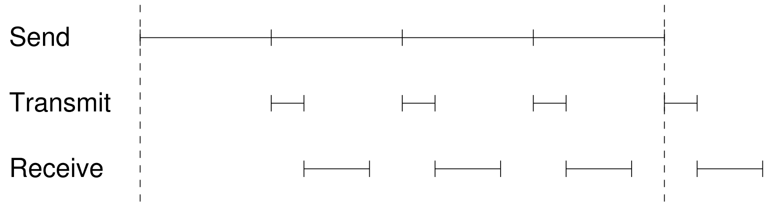
We may now use this model to analyze the coefficients in the above equations. The time T_r in equation 1 is the time for the remote invocation in which the parameter's network reference is passed to the callee, not counting the n remote invocations in the other direction. Since the size of a network reference does not usually exceed the capacity of a single network packet, the invocation is adequately modeled by figure 7, so that T_r can be thought of as the sum of the times for each phase:

$$T_r = T_r^s + T_r^t + T_r^r \tag{4}$$

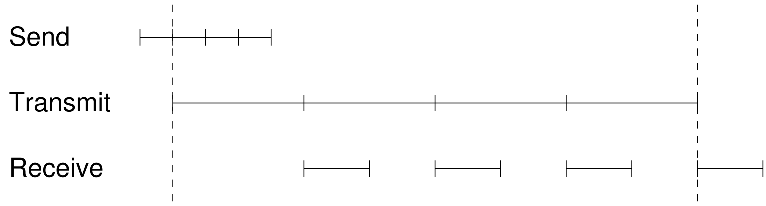
where the upper index indicates that the time belongs to the send, transmit, or receive phase, respectively. Since we are only interested in distinguishing between computation time and network-related time, the equation can be summarized as

$$T_r = C_r + \frac{S_r}{r} \tag{5}$$

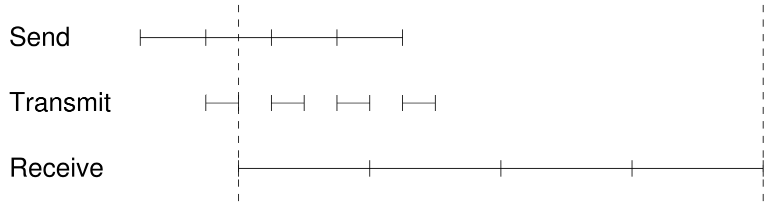
where C_r is the sum of the computational times T_r^s and T_r^r; S_r is the number of bits actually transferred during T_r^t (including the packet header and any acknowledgement packets), and r is the network transmission rate. Through similar reasoning, we can analyze the time t_i in equation 1, which is the time for one of the invocations that the callee performs on the parameter object:



(a) Sending Time Dominating



(b) Transmission Time Dominating



(c) Receive Time Dominating

Figure 8: Parallel Model

$$t_i = c_i + \frac{s_i}{r} \tag{6}$$

While this is similar to equation 5, the coefficients will not be the same in both equations, because, according to our assumption, the invocations of the parameter object do not have any parameters themselves.

When we turn to pass-by-value, as modelled in equation 2, we are mostly concerned with large objects that, when serialized, exceed the capacity of a single network packet. The parallel model of figure 8 must therefore be used to analyze the coefficients T_v and t_e. The time t_e for passing a single data element to the callee is given by

$$t_e = \begin{cases} t_e^s & \text{if } t_e^{max} = t_e^s \\ t_e^t = \frac{s_e}{r} & \text{if } t_e^{max} = t_e^t \\ t_e^r & \text{if } t_e^{max} = t_e^r \end{cases} \tag{7}$$

where, again, the upper index stands for the phase (or task) to which the time belongs. t_e^{max} is simply the maximum of t_e^s, t_e^t, and t_e^r, and s_e is the number of bits that go over the network for each data element. Note that none of these times is likely to correspond to a specific operation at run-time: of course data elements are sent packetwise, so that the times indicated are merely a proportional fraction of the total time for transferring the data. Also, the number of bits s_e is not simply the size of a data element, when serialized; it rather represents a proportional fraction of all the data that is transmitted (e.g. if data elements are 10 bit large when serialized, and the callee sends a transport level acknowledgement that is 100 bit long every 100 data elements, then the value of s_e would be 11 bit, not 10).

The times for filling and emptying the pipeline (outside the broken lines in figure 8) are not proportional to the size of the object m, still they are not negligible. We consider them a part of T_v, the constant time for pass-by-value:

$$
\begin{aligned}
T_v = C_v &+ \frac{S_v}{r} \\
&+ \begin{cases} p \cdot t_e^t + p \cdot t_e^r & \text{if } t_e^{max} = t_e^s \\ p \cdot t_e^s + p \cdot t_e^r & \text{if } t_e^{max} = t_e^t \\ p \cdot t_e^s + p \cdot t_e^t & \text{if } t_e^{max} = t_e^r \end{cases}
\end{aligned} \tag{8}
$$

C_v and S_v represent the constant computation time, and the constant amount of data that goes over the network if the parameter is passed by value, while p is the number of data elements that fit into a single network packet (for simplicity, we assume that all data packets have the same size). Substituting t_e^t with s_e/r and grouping differently, this becomes:

$$
\begin{aligned}
T_v = C_v & \\
&+ \begin{cases} \frac{S_v + p \cdot s_e}{r} + p \cdot t_e^r & \text{if } t_e^{max} = t_e^s \\ \frac{S_v}{r} + p \cdot (t_e^s + t_e^r) & \text{if } t_e^{max} = t_e^t \\ \frac{S_v + p \cdot s_e}{r} + p \cdot t_e^s & \text{if } t_e^{max} = t_e^r \end{cases}
\end{aligned} \tag{9}
$$

We may now use equations 5 to 9 to formulate the tradeoff (eq. 3) in a more detailed way. We show three different versions of it, for the case where either sending time, transmission time, or receiving time dominates, respectively:

$$m < \frac{C_r - C_v - p \cdot t_e^r + \frac{S_r - S_v - p \cdot s_e}{r}}{t_e^s} + \frac{c_i + \frac{s_i}{r}}{t_e^s} n \tag{10}$$

$$m < \frac{C_r - C_v - p(t_e^s + t_e^r) + \frac{S_r - S_v}{r}}{\frac{s_e}{r}} + \frac{c_i + \frac{s_i}{r}}{\frac{s_e}{r}} n \tag{11}$$

$$m < \frac{C_r - C_v - p \cdot t_e^s + \frac{S_r - S_v - p \cdot s_e}{r}}{t_e^r} + \frac{c_i + \frac{s_i}{r}}{t_e^r} n \tag{12}$$

4.2 Applying the model

Using the analytical model, the effects observed in the experiments can be explained. In particular, the reason for the counter-intuitive behavior when structs are passed by value becomes apparent.

Examining the individual network packets and their relative timing reveals that when structs are passed by value over the Ethernet, the callee's machine spends a noticeable amount of time *after* the last data packet arrived, but *before* sending the message that returns control to the caller. The amount of time grows with the size of the parameter object, and it can become quite large: several seconds for several thousand structs. On the other hand, we found that when the modem is used, the callee responds almost immediately after the last data packet.

The model shown in figure 8 explains this: it is the network transmission time that dominates on slow networks, but on fast networks, especially for structs, the message receive time dominates. The reason is that for the struct-like objects we used, true serialization and deserialization is required, which involves sending and analyzing type information, evaluating whether any references in the array point to the same objects, and, probably most time-consuming, allocating new objects on the callee's side. By comparison, passing integers by value is a simple copying operation, and therefore much faster (receive time does also dominate for integers on P100 / Sparc (Ethernet), but on the UltraSparcs, it is again the network that determines the data rate).

parameter	value
S_r	3976 bit
s_i	1368 bit
S_{v_i}	1872 bit
S_{v_s}	2304 bit
p_i	256
p_s	102
s_{e_i}	34.625 bit
s_{e_s}	86.59 bit

Table 3: Transmitted Data (Ethernet)

The formulas developed in the previous section allow us to analyze this effect in more detail. Table 3 shows the concrete values for the various size parameters in our experiments (these were found by examining the network packets themselves). Note that the size of a serialized integer is of course 32 bit; the additional 2.625 bit represent protocol information. A struct, when serialized, occupies 80 bit, and since fewer of these structs fit into a single packet, the protocol information amounts to a higher value (6.59 bit).

parameter	P100 / Sparc		UltraSparc
	Modem	Ethernet	Ethernet
C_r	139 ms	34.45 ms	7.35 ms
c_i	98 ms	8.34 ms	1.77 ms
$C_{v_i} + ...$	107 ms	19.34 ms	4.30 ms
$C_{v_s} + ...$	162 ms	42.12 ms	9.34 ms
$t^t_{e_i}$	(var.)	(0.00370 ms)	0.00370 ms
$t^r_{e_i}$	(0.00926 ms)	0.00926 ms	(0.00197 ms)
$t^r_{e_s}$	(0.8383 ms)	0.8383 ms	0.1786 ms

Table 4: Computational Times

Using equations 5 to 9 from the previous section, the values for the computational times shown in table 4 are obtained. (Parenthesized numbers were not measured directly, but inferred from the other values.) Formulas 11 and 12 can now be used to visualize how the tradeoff between pass-by-reference and pass-by-value is affected by the network transmission rate (fig. 9 and 10). The graphs extrapolate the values for P100 / Sparc

(Ethernet) for lower and higher network rates. (Note that the values around 10 kBps do not correspond to the modem experiments, because the modem entails a different, much higher computational overhead, as visible in table 4).

For both kinds of data, there are two sharply distinguished areas in the graphs. At lower network rates, data transmission time is the dominating factor, and it is pass-by-value that benefits most from any increases in network speed, while pass-by-reference becomes more and more unattractive (especially visible in fig. 9 for integers). The situation changes drastically when the *cutoff point* is reached, beyond which data transmission no longer dominates. For integers on P100 / Sparc (Ethernet), this occurs at

$$\begin{aligned} t^r_{e_i} &> \frac{s_{e_i}}{r} \\ r &> \frac{s_{e_i}}{t^r_{e_i}} \\ r &> \frac{34.625 \text{ bit}}{0.00926 \text{ ms}} \\ r &> 3.739 \text{ MBps}, \end{aligned}$$

while for structs the cutoff is already at $86.59/0.8383 = 103$ kBps. (On the UltraSparcs, the values are 17.5 MBps for integers, and 484 kBps for structs.) Beyond this cutoff point, higher transmission rates cannot make pass-by-value any faster, while pass-by-reference still benefits from further increases of speed. The curves then quickly approach the limit given by

$$m < \frac{C_r - C_v - p \cdot t^s_e}{t^r_e} + \frac{c_i}{t^r_e}\, n. \tag{13}$$

It is apparent that at very high network rates, all factors that determine the trade-off are purely computational, so that both mechanisms benefit equally from added machine power — the trade-off itself is not affected significantly in this range.

4.3 Perspectives for Middleware Design

Pass-by-value is very computation-intensive, as soon as true object serialization and deserialization is required. The computational overhead is in fact so high that faster networks will bring almost no speedup in such situations. One might argue that this is a typical Java problem, because serialization and deserialization are done by the virtual machine, and are therefore much slower as on, say, C++-based platforms. However, we did already use a Just-in-Time compiler on our fastest hardware, the UltraSparcs, and even if we assume an additional factor of ten for optimized, true native code, this would not be able to shift the cutoff point any higher than about 5 MBps with current high-speed hardware. The problem is thus going to persist.

Object serialization and deserialization is therefore an area where faster algorithms and optimizations are severely needed. Another way to tackle the problem, particularly in Java, would be to introduce further *value types*, like structs, objects of which do not have an “identity”. Such values could be passed almost as fast as integers, and programmers could avoid using the expensive object types in many situations.

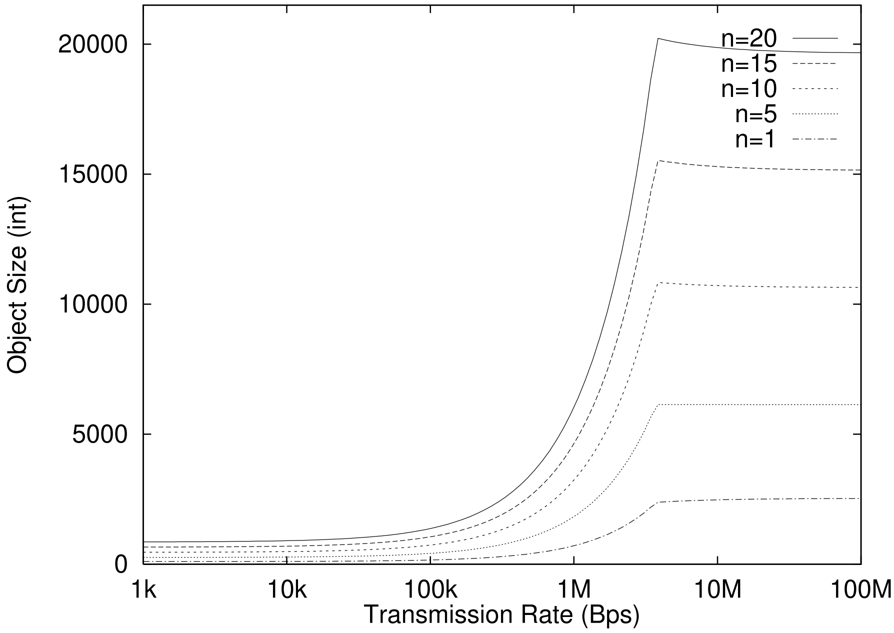


Figure 9: Dependence of Trade-Off on Transmission Rate (integers)

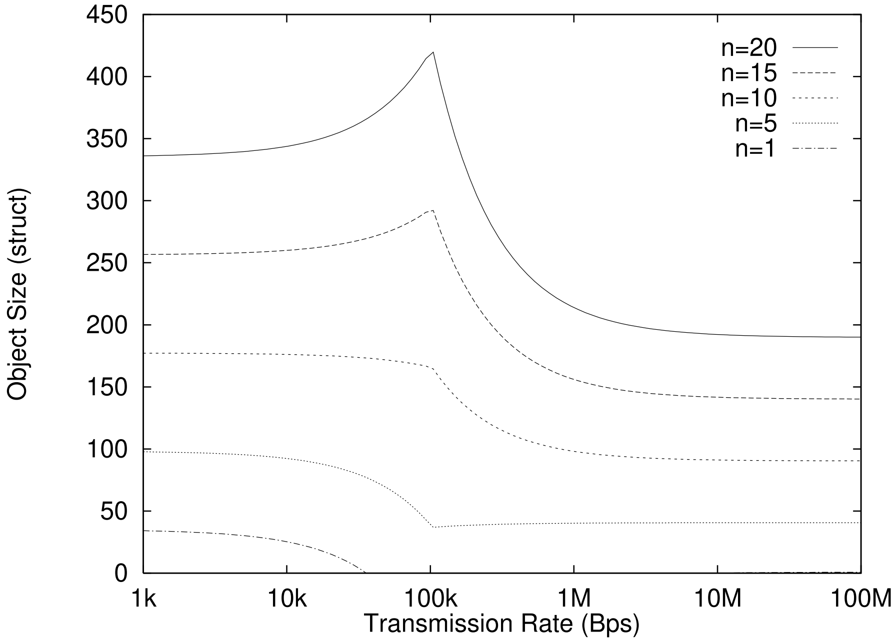


Figure 10: Dependence of Trade-Off on Transmission Rate (structs)

Our results show that, just as for pass-by-value, the crucial performance factor for the distributed version of pass-by-reference is also a computational one: the parameters C_r and c_i are decisive, since the terms S_r/r and s_i/r are small by comparison. Therefore, optimizations in the handling of network references and reducing the computational overhead of remote invocations in general would be most beneficial for distributed pass-by-reference.

5 Related Work

Other approaches for bringing objects "close" to another, like object migration and replication, have already been discussed in section 2.

Researchers have studied the performance of Remote Procedure Calls ever since the concept was invented. One fairly well-known study, which broke down the various performance factors for one particular platform is found in [Schroeder and Burrows 1990]; a similar study for more recent, CORBA-based technology was conducted in [Gokhale and Schmidt 1996].

The performance of Java-based distribution platforms has received considerable attention recently, e.g. [Breg et al. 1998, Hirano et al. 1998]. In agreement with our own observations, these studies identify Java's true object serialization as the crucial performance bottleneck, although the reasons for this are sometimes not thoroughly understood: one must be aware that a true serialization mechanism, which tracks and restores references in arbitrary object graphs, is inherently more expensive than a simple copy without tracking references. (In particular, [Breg et al. 1998] fails to make this explicit.)

RMI's parallelism at the system level is also recognized in [Breg et al. 1998], although analyzed in somewhat less detail.

Our work is distinguished from the aforementioned studies in that we are not so much concerned with the overall throughput for very large objects, but rather with the trade-off point beyond which pass-by-reference is more efficient. Since this occurs at comparably modest object sizes, there was no need for us to measure RMI's behaviour for objects much larger than that. In fact, it should be noted that for such objects RMI's performance is not necessarily linear in the object size anymore, because of the reference-tracking algorithm used. These effects, mentioned for example in [Breg et al. 1998], were negligible in the setup used for our own experiments.

More theoretically oriented, the LogP model for parallel computation [Culler et al. 1993] is an attempt to describe inter-processor communications in a formal way, similar to the approach presented here. LogP identifies three distinct times, namely, the *Latency*, the computational *overhead* for message generation, and the *gap*, which is the shortest possible interval between consecutive messages. Limitations of LogP with respect to our approach are that LogP, in its base form, only considers "short" messages, and it cannot easily model the three phases, or parallel tasks, that we identified in a remote invocation.

While passing objects by value can be seen as a way of moving data towards the receiver (in fact, copying it), we have not considered platforms and scenarios where *code* is moved as well. A useful taxonomy of the various options for code mobility, including remote evaluation, mobile agents, etc. is found in [Carzaniga et al. 1997], which also shows initial attempts at formal models to evaluate the performance of mobile code paradigms.

6 Summary

Passing objects by value is a useful optimization technique for distributed programming, offered by more and more distribution platforms. Our experiments and analytical studies indicate that for typical fine-grained application objects, pass-by-value is almost certainly faster than pass-by-reference. Only for rather large objects, which are not invoked very often by remote callees, pass-by-reference could be more efficient.

The analytical model that we developed helps to make the right decision in a given scenario, provided that the programmer knows the individual parameters that characterize the working environment — a general benchmark suite could be developed that determines these parameters automatically for a given setting.

Our model also shows that changes in computing power and network transmission rates may affect both passing mechanisms in counter-intuitive ways. It has been revealed which parts of the distribution platform have the highest impact on performance, the general result being that computational issues are so dominating that faster networks would virtually have no effect in some areas. In particular, true object serialization and deserialization has been shown to be a very expensive process where better algorithms and further optimizations are needed.

The most severe drawback of passing objects by value is that it is really a semantic concept that is merely exploited for performance reasons. It is therefore only of limited applicability, and should be complemented by facilities for object migration and replication on future distribution platforms. As these become available, our model can be extended to analyse them in a similar way.

The ultimate goal is to use our model within an analysis tool that decides about passing mechanisms and other distribution-related issues *automatically*, without programmer interference. The tool is envisioned to use both static, compile-time analysis and dynamic techniques, taking into account both the efficiency tradeoffs involved and the semantic constraints that need to be met so that no changes of program behavior are induced.

Acknowledgements

I would like to thank Boris Bokowski, Gerald Brose, Markus Dahm, Matthias Horn, and Klaus-Peter Löhr for many helpful comments and inspiring discussions.

References

[Bal et al. 1992] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum: *Orca: A Language for Parallel Programming of Distributed Systems.* IEEE Trans. SE, vol. 18, No. 3, March 1992, pp. 190-205.

[Breg et al. 1998] Fabian Breg et al.: *Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++.* Proc. ACM Workshop on Java for High-Performance Network Computing, Stanford University, Palo Alto, February 1998.

[Carzaniga et al. 1997] Antonio Carzaniga, Gian Petro Picco, Giovanni Vigna: *Designing Distributed Applications with Mobile Code Paradigms.* Proc. ICSE '97, May 1997.

[Culler et al. 1993] David Culler et al.: *LogP: Towards a Realistic Model of Parallel Computation.* Proc. 4th ACM SIGPLAN Symp. PPoPP, San Diego, May 1993.

[Gosling et al. 1996] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification.* Addison–Wesley, 1996.

[Hirano et al. 1998] Satoshi Hirano, Yoshiji Yasu, Hirotaka Igarashi: *Performance Evaluation of Popular Distributed Object Technologies for Java.* Proc. ACM Workshop on Java for High-Performance Network Computing, Stanford University, Palo Alto, February 1998.

[Jul et al. 1988] Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black: *Fine-Grained Mobility in the Emerald System.* ACM Trans. CS, Vol. 6, No. 1, February 1988, pp. 109-133.

[OMG 1995] OMG: *The Common Object Request Broker: Architecture and Specification.* Revision 2.0, July 1995.

[OMG 1996] OMG: *Objects–by–Value: Request For Proposal.* OMG document orbos/96–06–14, June 1996.

[OMG 1998] OMG: *Objects–by–Value: Joint Revised Submission.* OMG document orbos/98–01–01, January 1998.

[Philippsen 1997] Michael Philippsen, Matthias Zenger: *JavaParty: Transparent Remote Objects in Java.* Concurrency: Practice and Experience, Vol. 9, No. 11, November 1997, pp. 1225–1242.

[Gokhale and Schmidt 1996] Aniruddha Gokhale and Douglas C. Schmidt: *Measuring the Performance of Communication Middleware on High-Speed Networks.* Proc. SIGCOMM, Stanford University, August 1996.

[Schroeder and Burrows 1990] Michael D. Schroeder and Michael Burrows: *Performance of Firefly RPC.* ACM Trans. CS, Vol. 8, No. 1, February 1990, pp. 1–17.

[Sun 1997] Sun Microsystems: *RMI — Remote Method Invocation.* Located on the Java website: `http://java.sun.com/products/jdk/1.1/docs/`