Thread-safe Efficient Mutable Arguments for Remote Method Invocation in Java

Paolo G. Giarrusso

Philipps University Marburg
Marburg, Germany
pgiarrusso@informatik.uni-marburg.de

Abstract. Distributed object-oriented middlewares, like Java RMI or CORBA, abstract message sends into method calls, providing programmers with a familiar object-oriented programming model. However, a remote method cannot mutate the heap of the client with the same ease as a local method without incurring significant overhead. Workarounding this limitation requires exposing implementation details of the server, thus compromising modularity. Existing solutions for Java do not implement the full Java semantics, in particular for multi-threaded programs. Alternatively, they require using non-standard JVMs, a requirement which compromises portability and interoperability.
We discuss a simple approach to eliminate the semantic difference, characterize its overhead and discuss novel optimizations tailored to our scenario, which we believe will minimize the overhead.

Keywords: distributed objects, parameter passing, thread safety, modularity

1 Introduction

Distributed object-oriented middlewares like Java RMI [23] or CORBA provide programmers with a familiar object-oriented programming model for distributed programming. However, writing *efficient* distributed applications is nowadays still substantially different to and harder than writing concurrent, centralized ones. Beyond the handling of partial failure, performance concerns complicate the programming model. Each interaction with a different network node requires a costly round-trip, and minimizing the number of round-trips is important to achieve suitable performance.

In particular, a method invoked on a remote object cannot directly modify the heap on the calling machine; instead, each modification requires a synchronous remote method invocation back on the caller machine, and the round-trip time is often significant, especially for fine-grained calls [15, 8].

Support for efficient mutation is desirable. Nowadays using immutable objects is often recommended to simplify concurrent programming and programming in general; one may therefore argue that restricting mutation is not problematic. However, we argue that being *restricted* to immutable objects in the

interface not only creates a mismatch with normal OO semantics, but also a modularity problem. If mutation is forbidden or restricted, programmers have to return mutated copies of arguments back to their callers, and thereby stipulate within the interface which objects can be mutated. While this allows equational reasoning on the code of the client, it restricts the possible implementation of the server. Small modifications to the implementation propagate then to the remote interface. This problem motivates research on distributed middleware not based on message passing, as argued for instance by Amza *et al.* [2]; abstracting from the context of distributed programming, this problem was already noted by Wadler [22] in relation to Haskell, where mutation is forbidden altogether. We further discuss elsewhere [14] what we consider to be the general underlying tradeoff between the strength of interfaces and information hiding.

Especially, parameters of a remote method cannot be efficiently modified, unlike the receiver of the method invocation. A few alternative parameter-passing conventions exist:

- With pass-by-copy, the argument values are transmitted, so that read operations can be performed locally, but no modification is propagated back to the original host. The client can apply needed modifications after the called method returns, but it is inconvenient [20].
- With pass-by-remote-reference, remote references to arguments are transmitted, not their values. Thus, accessing a remote object requires a nested remote method invocation and is many orders of magnitude slower than local modifications. This however preserves the same semantics as for local method calls.
- Finally, with pass-by-copy-restore, if correctly implemented as in NRMI [20], parameters are transmitted to the remote host, like with pass-by-copy, but modifications are propagated back to the original host and restored onto the modified objects. However, in a multi-threaded client this introduces data races with any conflicting update by other threads. Additionally, if the remote host retains references to arguments after the invocation completed, such references will refer only to the temporary argument copy.

Alternative solutions include algorithms object migration and object replication; however, no single solution ensures suitable performance for all object access patterns [4], and none is tailored specifically to argument passing. We further discuss alternatives solutions in Sec. 5.

Overall, we face a choice between efficient parameter access and a correct implementation of Java standard pass-by-reference semantics. Because of subtle semantic differences, converting a centralized application to a distributed one is error-prone, as semantic differences are not detected [20].

Automatic partitioning systems like J-Orchestra [18] allow running a centralized application in a distributed environment without changes to its source. Such systems use pass-by-remote-reference to preserve existing semantics. When an application is composed of multiple independent parts, performing little communication between them, this overhead can be tolerable. However, when remote method invocations are more frequent, the overhead is likely to be significant.

J-Orchestra [20] allows thus the programmer to enable pass-by-copy-restore for selected cases, where he believes that the semantic difference will not introduce bugs. This is however an error-prone process, which would be simpler if pass-by-copy-restore provided accurate pass-by-reference semantics.

In this paper we aim to support multi-threaded programs with correct semantics for parameter passing. Our additional goal is to minimize the overhead and make it comparable with standard pass-by-copy-restore.

In summary, we propose the following contributions:

- We combine existing algorithms to implement a thread-safe variant of pass-by-copy-restore, and argue that it has the same semantics as pass-by-reference (Sec. 3).
- To support this algorithm, we discuss semantic problems with visibility of memory actions in existing middlewares supporting distributed lock management (henceforth DLM), and a simple solution (Sec. 3.1).
- We characterize possible causes of overhead caused by the basic algorithm (Sec. 4), and we discuss which optimizations we plan to explore to reduce the additional overhead to a negligible amount (Sec. 4).
- We discuss how argument mutation can be useful for automatic partitioning and for batching of remote invocations (Sec. 1 and 6).

Additionally, we discuss related work in Sec. 5. We conclude and discuss future work, including experimental evaluation, in Sec. 6.

2 Background: Pass-by-copy-restore and its limitations

We first explain the pass-by-copy-restore algorithm and why it is not thread-safe.

Let us consider a remote method invocation `r.method(arg1, arg2, ...)` from host H_{client}, where `r` is a remote object located on host H_{server}, and `arg1, arg2, ...` are the arguments of the invocation. With pass-by-copy-restore all arguments are transmitted to host H_{server}, where they are unmarshaled and passed to `method`. After the execution completes, the return value and updated arguments are transmitted back to host H_{client}. The middleware then overwrites the original arguments with the updated ones.

However this algorithm is incorrect [20]. Consider the program in Fig. 1. We see that `method` modifies the object pointed to by a field of an argument, and then make the modified object unreachable. Therefore, the modified object `arg.a` will not be transmitted back to H_{client}. Tilevich and Smaragdakis [20] present also more compelling examples, together with a solution: In a nutshell, they propose to keep track of all the objects which were originally passed, and transmit all of them back, even if they are no more reachable from the arguments.

This solution does not yet eliminate all semantic differences with pass-by-remote-reference. If another thread concurrently updates the arguments on the first host, we get undesired semantics.

A first case is shown by the following example: On host H_{client} another thread executes `arg.b++` on the argument of `method`, while `method` is still executing.

```
class Pair<A, B> {
  A a;
  B b;
}
...
void method(Pair<Pair<Int, Int>, Int> arg) {
  arg.a.a++;
  arg.a = new Pair<Int, Int>();
}
```

Fig. 1. Argument modification which is not reflected back

Since the modification is on a separate field, Java semantics guarantee that no interference happens; yet, when `method` returns, the original value of `arg.b` will overwrite the modified value.

To avoid this particular problem, we can save a copy of the original argument, termed a *twin* [2], compare it with the modified argument and transmit only the differences back. Since `arg.b` was not modified on H_{server}, it will not be overwritten on the caller.

Yet, while this change is essential, it does not solve a more significant problem. What happens if two threads on different hosts perform conflicting memory operations, for instance if they both write to the *same* field?

3 Ensuring thread-safety

Fundamentally, pass-by-copy-restore creates *replicas* of the arguments on H_{client} on the server host H_{server}. Such replicas survive only until the invoked method returns. Until then, if multiple threads modify replicated objects, a consistency protocol needs to reconcile the replicas to ensure standard Java semantics.

Since typically a consistency protocols does not offer suitable performance in all scenarios, we tailor and optimize a consistency protocol for parameter passing.

As a starting point, we consider a variation of *release consistency*, introduced by Munin [3] and TreadMarks [2]. Distributed release consistency can be adapted to Java because it is similar to the Java memory model or JMM [12].

We summarize informally a few needed concepts from the JMM which we use in this paper. Two memory accesses to the same memory location are termed *conflicting* if at least one of them is a write. In a *correctly synchronized* program, both accesses have to happen under the same lock[1], and the JMM guarantees for such programs a tractable semantics for memory accesses, that is *sequential consistency*. Otherwise the accesses are said to form a *data race*, which is almost

[1] We exclude from the exposition volatile variables for simplicity, but writing to a volatile variable can be handled in a similar way to a lock release, from the point of view of visibility of memory operations – see later.

always a programming error. Semantics of incorrectly synchronized programs are even incorrectly specified by the JMM [26], and a correct specification is an open research question [1], thus it is unclear how to formally compare our semantics with the Java ones on such programs. In the following, we simply restrict our attention to correctly synchronized programs.

To implement the JMM on a multiprocessor, memory operations performed in a critical section must become *visible* to other threads within the critical section [10]. Namely, modifications accumulated in processor caches are flushed to main memory upon releasing a lock, and reads inside a critical section must not be speculatively performed before the lock is acquired. It is also essential that any reordering does not change the order of *synchronization operations*, which include lock acquires and releases.

Our goal is now to implement the JMM in a distributed setting. Each lock `l` is part of a Java object `o`, which is located on a host H_o. A thread which acquires `l` thus performs a remote request to host H_o and waits a reply, granting lock acquisition. A thread which releases `l` must inform H_o and proceed.

Note that threads accessing fields of an object need to coordinate their action by using *some* lock, not necessarily the lock associated with the object. In particular, different fields might be protected by different locks.

When a remotely invoked method, executing on H_{server}, releases a lock, we must both flush the processor caches and transmit argument modifications back to H_{client}; to compute modifications, we compare the (potentially modified) object with its twin object, containing the original value. To avoid transmitting the same modifications twice, we also update the twin. Finally we transmit back the modifications, and after they are applied we release the lock.

The source code does not invoke a different routine to invoke these modifications. Rather, bytecode transformation is applied to replace standard lock release operations with invocations to routines implementing the above algorithm, as in existing solutions for distributed lock management in Java (see Sec. 3.1).

A possible optimization, inspired by lazy release consistency, is to delay applying the modifications until another synchronization operation is performed (for instance, the same lock is reacquired), while the invoked method continues processing. For space reasons, we omit a detailed description of the protocol required for this optimization: we just mention that the host invoking the remote method, the host on which the method is invoked, and the host where the monitor is located may all be different. With lazy release consistency, when acquiring a lock, we must additionally ensure that all updates have already been applied.

Unlike us, TreadMarks uses an invalidation protocol, not an update protocol: modifications are requested later, when the shared data is actually accessed again. TreadMarks relies on virtual memory hardware to intercept the access; while intercepting accesses would be possible also in Java by modifying the bytecode, we currently believe that this would cause too much overhead. Moreover, such a protocol would save bandwidth but suffer from increased latency. Finally, TreadMarks operated on pages, not on objects: our unit of sharing is naturally smaller, limiting the potential network overhead. Therefore, we currently plan to

avoid intercepting accesses and use an update protocol, but performance measurements are needed.

3.1 Distributed lock management and its current problems

KaRMI [7] and J-Orchestra [19] implement DLM in different ways, and we need to use one of the two approaches. For our goal, we of course need the semantics of distributed locking to match the Java semantics for locking. It is however unclear whether it is the case, at least for the current Java memory model [12].

Programmers use locking to achieve mutual exclusion, and this is correctly implemented in both approaches. But as we have explained, locking also guarantees that memory operations become visible to other threads. Thus, when synchronizing on a remote monitor, it is not enough to request the remote host to acquire the lock, but a local synchronizing action must be also executed. Visibility of memory operations is not discussed by existing solutions for KaRMI and J-Orchestra, and the described algorithms perform just a remote request to operate on remote locks.

When acquiring and releasing a remote lock, we can ensure the correct visibility of memory operations by performing a synchronizing operation with the desired effects. An appropriate operation is to perform a dummy read from a `volatile` variable on lock acquisition, and a dummy write on the same variable on lock release. The same variable must be used for all locks on a machine, because using different variables might not be enough. Critical section under different locks can be related[2] because of operations by a remote thread using both locks, which is however not visible to the JVM on this node. If what happens in a critical section must be visible to another, they need to coordinate by synchronization operations on the same variable. Keeping track at runtime of when there is an ordering between two locks would be too expensive; therefore we assume conservatively that the two locks are not independent, similarly to what happens in typical JVMs.

3.2 Escaping references

As explained, if a remotely invoked method receives an argument `arg` and saves a references to it, under pass-by-remote-reference but not pass-by-copy-restore, it saves a remote reference, which can be used to modify the remote heap. Here we explain how to address this difference.

We can use escape analysis to discover whether remote arguments escape the invoked method, i.e., the server routine keeps any reference to remote arguments; on these code paths, we can construct and save then a remote reference to the original object to be stored in the server's heap. The construction of the remote reference could be inserted thanks to bytecode transformation.

However, for the server to be fault-tolerant, it must not keep transient per-client state, e.g., remote references to arguments [20]. In the context of different

[2] For readers knowledgeable with the JMM, they can be ordered by happens-before.

middlewares where this is enforced, e.g., Remote Batch Invocation [8], we would still detect potential escapes by escape analysis, and then report them to the programmer. We would also provide a method `T localize<T>(T o)` to explicitly mark escapes as safe. The semantics of `localize` is that it returns a copy of the passed argument, which has thus a distinct identity and is allowed to escape. It is thus clear that modifications to this copy are not sent back to the original host. Static analysis would be used to determine when the copy can be safely omitted, e.g., when the original reference is no more modified.

We therefore report to the programmer any problematic escape, instead of silently producing incorrect or simply unexpected behavior.

4 Minimizing additional round-trips

For a standard remote method invocation, network communication happens only at invocation and return time. With DLM, using a remote lock requires additional network communication. In our variant of pass-by-copy-restore, however, using a local lock can require additional communication. In particular, a lock used by the implementation of a library to protect other data can trigger unneeded remote communication. Additionally, comparing an object and its twin can be expensive [17].

We already outlined how lazy release consistency can alleviate this problem, by making the needed communication in essence asynchronous. Our scenario allows further optimizations:

- Though escape analysis, we can detect which locking operations affect only thread-local objects and remove them, similarly to what Java virtual machines do. In our case, we would piggyback this modification on the bytecode transformation which we perform on lock operations.
- Static analysis can identify statically which fields can or cannot be modified, allowing to reduce comparison costs.
- We can often hoist the locking operation outwards from the library calls (lock coarsening). This can reduce the number of synchronization operations and move them next to each other, allowing to perform them in a batch.
- Finally, by lock coarsening we can even move the locking operations to happen around the call, on the caller side, before arguments are sent. This avoids any additional round-trips for locking, at the only cost of modifying the client stubs at runtime to acquire the correct locks.

5 Related work

Our work is most closely related to object replication techniques. Our basic consistency protocol is in essence a multiple-writer, home-based, update protocol ensuring release consistency. Much work on optimizing Java RMI is based instead on sequential consistency [5, 13, 11], which is known to be slower [2],

and requires often totally-ordered multicast. An exception is presented for instance by Fang *et al.* [6]. They implement a distributed JVM where objects are transparently replicated, using a set of release-consistency protocols tailored for different scenarios. Unlike them, we propose to implement our solution in pure Java and not on a customized JVM. The same issue applies to page-based DSM solutions for Java [21, 25], which additionally use a coarser granularity. General-purpose object replication can speed up reads on replicated objects, but incurs the cost of keeping them consistent. Guaranteeing good performance is complex and requires different protocols for different access patterns [4].

Our work can be also compared to object mobility, as implemented for instance in Emerald [9] and J-Orchestra [18]. However, our protocol supports multiple writers operating on the same object without additional overhead. Additionally, solutions for Java for object mobility allow storing only indirect references to a proxy to the mobile object, slowing down local accesses; other solutions have to modify the JVM.

One of our technical contributions, compared to existing object replication techniques, is that we focus on parameter passing, which expresses a distinctive and important pattern in data usage. The use of lock coarsening techniques to optimize lock placement is also novel, and this is made possible by our scenario: We piggyback lock management on method invocation and return where this is possible without altering the semantics.

Moreover, a simpler technique to speed up accesses to objects is to batch together multiple remote operations, either implicitly or explicitly [16, 8]: Instead of maintaining a copy of the object locally, a set of remote operations is executed at once on the remote host. However, explicit batching requires slight modifications to the application by the programmer, while implicit batching achieves inferior performance.

Tilevich and Gopal [17] also extend copy-restore to transmit back only the changes to the argument, obtaining *copy-restore with delta*. However, their motivation is improved performance on low-bandwidth networks, and they neither aim for nor achieve thread-safety.

6 Conclusions and future work

We have described how a middleware could support pass-by-reference semantics for arguments, and why we believe it is important to support these semantics.

Our future plans include describing in more detail the proposed algorithms, writing a semi-formal correctness proof and implementing a prototype by extending an existing middleware.

We plan an experimental evaluation, at least on applications using pass-by-remote-reference, especially where pass-by-copy-restore is currently not applicable, to measure the improvement, and on applications where pass-by-copy-restore is used today, to measure the additional overhead (if any).

We also plan to integrate the technique with Remote Batch Invocation, to allow remote batches to modify objects on the calling host inside a remote batch.

Expected technical challenges We expect bytecode transformations to be difficult to apply to system libraries. We will either avoid the need to modify synchronizing operations in system libraries, or reuse existing solutions from J-Orchestra [18] to implement these modifications. The required static analyses are interprocedural; we plan to investigate devirtualization for them [24].

Acknowledgments. We thank Shriram Krishnamurthi, Giuseppe Pappalardo and Emiliano Tramontana for helpful discussion on the idea, and Tillmann Rendel for helpful feedback on this paper. This work is supported by the European Research Council, grant #203099.

References

1. Sarita V. Adve and Hans-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53:90–101, August 2010.
2. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29:18–28, 1996.
3. John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *SIGOPS Oper. Syst. Rev.*, 25:152–164, September 1991.
4. Jörg Domaschka, Thomas Bestfleisch, Franz Hauck, Hans Reiser, and Rüdiger Kapitza. Multithreading strategies for replicated objects. In *Proc. Middleware*, volume 5346 of *LNCS*, pages 104–123. Springer-Verlag, Berlin/Heidelberg, 2008.
5. John Eberhard and Anand Tripathi. Efficient object caching for distributed Java RMI applications. In *Proc. Middleware*, volume 2218 of *LNCS*, pages 15–35. Springer-Verlag, London, 2001.
6. Weijian Fang, Cho-Li Wang, and Francis C. M. Lau. On the design of global object space for efficient multi-threading Java computing on clusters. *Parallel Computing*, 29(11-12):1563 – 1587, 2003.
7. Bernhard Haumacher, Thomas Moschny, Jürgen Reuter, and Walter F. Tichy. Transparent distributed threads for Java. In *Proc. Int'l. Parallel and Distributed Processing Symp.*, Los Alamitos, CA, 2003. IEEE Computer Society.
8. Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *Proc. ECOOP*, pages 595–617, Berlin/Heidelberg, 2009. Springer-Verlag.
9. Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. Comp. Syst. (TOCS)*, 6(1):109–133, 1988.
10. Doug Lea. The JSR-133 cookbook for compiler writers, 2004. Last accessed on 15 April 2011.
11. Jason Maassen, Thilo Kielmann, and Henri E. Bal. Efficient replicated method invocation in Java. In *Proc. ACM Conf. Java Grande (JAVA)*, pages 88–96, New York, 2000. ACM Press.
12. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. *SIGPLAN Notices*, 40:378–391, January 2005.
13. N. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Transparent consistent replication of Java RMI objects. In *Proc. IEEE Int'l Symp. on Distributed Objects and Applications*, pages 17–26, Los Alamitos, CA, 2000. IEEE Computer Society.

14. Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proc. ECOOP*. Springer-Verlag, 2011. Accepted for publication.
15. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.
16. Eli Tilevich, William R. Cook, and Yang Jiao. Explicit batching for distributed objects. *Proc. Int'l Conf. Distributed Computing Systems (ICDCS)*, pages 543–552, 2009.
17. Eli Tilevich and Sriram Gopal. Expressive and extensible parameter passing for distributed object systems. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*. Accepted for publication.
18. Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proc. ECOOP*, pages 178–204, London, 2002. Springer-Verlag.
19. Eli Tilevich and Yannis Smaragdakis. Portable and efficient distributed threads for Java. In *Proc. Middleware*, volume 3231 of *LNCS*, pages 478–492. Springer-Verlag, Berlin/Heidelberg, 2004.
20. Eli Tilevich and Yannis Smaragdakis. NRMI: Natural and efficient middleware. *IEEE Trans. Parallel Distr. Systems (TPDS)*, 19:174–187, 2008.
21. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Run-time optimizations for a Java DSM implementation. *Concurrency and Computation: Practice and Experience*, 15(3-5):299–316, 2003.
22. Philip Wadler. The essence of functional programming. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 1–14, New York, 1992. ACM Press.
23. Jim Waldo. Remote procedure calls and Java remote method invocation. *IEEE Concurrency*, 6:5–7, 1998.
24. Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 19–36, New York, 2008. ACM Press.
25. Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.
26. Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java Memory Model. In *Proc. ECOOP*, volume 5142 of *LNCS*, pages 27–51. Springer-Verlag, Berlin/Heidelberg, 2008.