

A Generic Middleware for Intra-Language Transparent Distribution

E. Klinskog¹, Z. El Banna², P. Brand¹

July 16, 2003

¹Swedish Institute of Computer Science, Kista, Sweden

²IMIT/KTH – Royal Institute of Technology, Stockholm, Sweden

SICS Technical Report T2003:01

ISSN 1100-3154

ISRN:SICS-T-2003/01-SE

Contents

1	Introduction	4
1.1	Two Approaches to Middleware	5
1.2	The Distribution Subsystem	8
1.3	Outline	10
1.4	Terminology	10
2	Requirements	11
2.1	Transparency	13
2.2	Control and Tunability	15
3	Target Languages	17
3.1	Concurrency	18
3.2	Reference Security	18
3.3	Data Structuring	19
3.4	Distinction between Stateless and Stateful	19
3.5	Memory Management	20
3.6	Example languages/systems	20
4	An Abstract Model of Language Entities	20
4.1	Synchronous Stateful	21
4.2	Asynchronous Stateful	22
4.3	Monotonic	22
4.4	Stateless	23
4.5	Lock	23
4.6	Entity Types and Annotations	24
4.7	Classifying Language Entities	24
5	The OS Service Component	26
6	The Distribution Subsystem	27
6.1	The DSS Framework and VM interaction model	28
6.1.1	Sharing Entities	28
6.1.2	Guarded Replication	29
6.1.3	Passing References	31
6.1.4	The At Most One Copy Property	33
6.1.5	Lazy Replication	34
6.1.6	Completing the Framework	35
6.1.7	Globalization and Coordination	35
6.1.8	Memory Management in the DSS	37
6.1.9	Coordination and Protocols	38

6.1.10	Cooperative Memory Management	39
6.1.11	Bootstrapping	40
6.2	DSS Internals	40
6.2.1	Protocol Machine	40
6.2.2	Identifiers and Sites	40
6.2.3	Message Contents	41
6.2.4	Channels	42
6.2.5	Messaging	43
6.2.6	Fault Detection	44
6.3	Sharing Strategy	45
6.3.1	State of the Art	46
6.3.2	The Three Dimensions of Sharing Strategy	47
6.3.3	Coordination Strategy	48
6.3.4	Consistency Strategy	49
6.3.5	Memory Management Strategy	52
6.3.6	Discussion	53
7	Extending and Coupling a Virtual Machine	54
7.1	Core Extensions	54
7.1.1	Thread handling	54
7.1.2	Guards	55
7.1.3	Operation Tokens and Callbacks	55
7.1.4	Memory management	56
7.1.5	Marshaling	57
7.1.6	Annotation Service	58
7.2	Coupling	58
7.2.1	Cross Component References	59
7.2.2	Mapping the Concrete to the Abstract	60
7.2.3	Memory Management	60
7.3	Some Notes on Our Implementation	60
8	Extendibility of the DSS	61
9	Performance comparison	65
10	Conclusion	69
11	Future Work	70

1 Introduction

It is generally recognized that developing distributed applications is very difficult. The simplest approach is to program each process using established centralized programming techniques and complement with coordinated message sending between processes using some communication service like TCP. There are many aspects that must be dealt with; synchronization between processes, getting the coordination in message sending right, preventing race conditions, avoiding deadlock, constructing outgoing messages, and parsing incoming messages. The sheer complexity of programming distributed applications on this level is overwhelming.

To lessen the inherent complexity of distributed programming a number of distributed programming systems and/or middlewares have been developed over the years. Some target special kinds of networks, e.g. clusters on LANs. Others are more general and can be used on many different kinds of networks; LANs, WANs, the Internet and possibly wireless. In this paper we are concerned with systems of the second kind, systems that do not assume any special network topology or hardware.

Some middlewares are geared exclusively to messaging abstractions. MPI [11] and group communication services (GCS) are examples of systems of this kind.

On a higher level other middlewares support abstractions on the programming level. What most of these systems have in common is that they extend the ordinary programming constructs used in centralized programming to distribution. Put another way these systems strive to make distributed operations behave very much the same way as centralized operations. These systems strive for distribution transparency.

The remote procedure call(RPC)[3] mechanism makes a remote procedure call look, and feel, very similar to an ordinary procedure call for the programmer. Under the surface something very different is going on as the procedure is being executed on a different machine than the caller. There are numerous distributed object systems that let processes reference objects located in other processes. Invoking these remote objects via remote method invocation (e.g. Java RMI [23]) is deliberately similar to local method invocation.

While the above systems strive for transparency, they fall short as there are still significant semantically differences between distributed operations and local ones. Other systems actually do offer complete transparency. In the Emerald system [17], also a distributed objects system, the operational semantics of method invocation is the same irrespective of where the object is located. Erlang [2] offers complete transparency in the context of that lan-

guage. Mozart [4] offers complete transparency for the language OZ; this includes not only objects but other kinds of symbolic programming constructs, closures, single-assignment variables, and futures.

A slightly different approach was taken by the OMG [27] group. The philosophy here is, rather than making distributed operations as similar as possible to local ones, to standardize them. The main goal here was interoperability. Nevertheless, at some level, CORBA [28] also extends centralized programming constructs. The concept behind CORBA is to distribute objects; the difference is that method invocation follows the standard. Interoperability is outside the scope of this paper, but CORBA and similar systems are still of interest. For one thing, CORBA can be, and is, used even when interoperability is not needed, e.g. one Java process invoking an object in another via CORBA.

These middleware systems have made developing distributed applications easier, although in our view, only slightly easier. In this document we present what we believe to be a middleware architecture that, realized in implementations, will significantly simplify development of efficient distributed applications. Furthermore we claim that the architecture presented here provides a road map on how to maximize distribution support based on extending centralized programming constructs to distribution. (There may, of course, be all kinds of distribution support, not based on language extension). We claim to have a new approach to building middleware, with many advantages over middleware systems of today. Proof of concept is supported by an implementation.

1.1 Two Approaches to Middleware

High-level middleware systems of today can be divided into two groups depending on whether the implementations are coupled to one particular programming language or not. Systems that are coupled to one particular language we call **language-dependent middleware**(LDM) and those that are not, **language-independent middleware**(LIM).

Java (RMI) [23], Java (Object Voyager) [34], Erlang [2], and Mozart [4] are all examples of the first group, **language dependent middleware**. From the viewpoint of the programmer these are all examples of **distributed programming systems** (DPS) that allow the programmer to develop distributed applications. The middleware component is totally integrated with the programming environment to form one monolithic system. Effort is usually made to provide some transparency; from the programmer's point of view programming distributed applications is not so different from programming centralized applications. The four examples all represent different program-

ming languages; even the two java-based systems are different in the sense that the language has different constructs and semantics.

The one good example of a **language-independent middleware** is CORBA [28]. Here the middleware component is separate; to make practical use the programmer will in addition need a (properly instrumented) centralized programming system. Here all activity that involves distribution will involve using the system's CORBA interface. Since the middleware is separate from the programming system, distributed applications may be built partly in one programming language and partly in another; this solution also caters for interoperability. If the distribution services are basically limited to sending messages between machines, the language-independent middleware may be reduced to an interchange format, e.g. XML [6].

Before we compare the two approaches we should make clear that we are not addressing interoperability issues. If it is necessary that the application is developed partly in one language and partly in another, or (which from our point of view is the same thing) if it is necessary to connect applications written in different languages together, then a language-independent middleware is needed. We are concerned with tools to develop distributed applications within one language.

Comparing the two approaches there are three important factors; ease of use, efficiency and applicability. First, we consider these factors conceptually.

Ease of Use. Since a LDM is integrated with a programming language/system into a distributed programming system, the programming model can be simplified. The centralized programming constructs can be extended for distribution minimally to make distributed programming differ as little as possible from centralized programming. In Java with RMI, for instance, a method invocation on a remote object may be made in much (but as we shall see, not totally) the same way as method invocations on local objects. With a LIM (e.g. CORBA) the local programming constructs and the distributed ones have their own separate models.

Efficiency. We consider the efficiency issue, i.e. execution performance and memory usage, from a conceptual point of view. Since a LDM is tailor-made for one programming system it can be highly optimized. Redirection of operations from the language level to the middleware, and from the middleware to the language level can be done optimally. Data structures can in an efficient way store their distribution status internally. Data structures may be serialized and deserialized optimally. The serialized representation may be made optimally small taking full advantage of language invariants. These kinds of optimization are not possible in LIM systems

Applicability. Language-independent middleware has inherently wider applicability. There are many different programming languages and systems in the world and they all have their proponents. Middlewares that are useful for all, or even most, programming languages will, everything else being equal, be more attractive than any language dependent solution, even if the most popular programming language is taken as a base.

Conceptually the picture is very clear. LDMs thus have two major advantages, easy of use and efficiency, and LIMs one, applicability. In practice, considering the systems of today the issue is clouded.

First of all, efficiency in any system depends to a considerable degree on the skill, knowledge, and effort spent on developing the system. From the above argument one would expect that LDMs are more efficient than LIMs. In the evaluation section [9] we actually do make this comparison (for one simple benchmark) between some existing LIMs and LDMs (as well as our system), and although there is a tendency for LDMs to outperform LIMs there is a lot of noise.

Secondly the LDMs of today are all incomplete in that they do not contain a full repertoire of distributed algorithms. A full repertoire is necessary for efficiency under all conceivable patterns of use. For instance, most but not all LDMs that offer distributed objects support only the simplest distributed algorithm for maintaining state consistency, where the object state is never moved, cached or replicated. We shall have more to say about this in later sections. For now we merely note that in the light of what could be achieved LDMs are always deficient on the algorithmic level, to varying degrees, and often deficient on the engineering level.

How can we explain the inefficiencies of LDM systems of today? This is undoubtedly partly due to the fact that this field is still in its infancy. The middleware does not have the same depth of experience or have had the same amount of time to evolve as centralized programming systems. However, we believe that there are two other important contributing factors both related to the fact that the middleware is geared to one programming language only:

Firstly, with the possible exception of a really popular programming language like Java, the limited applicability of LDMs compared to LIMs may prevent developers from putting in the required time and effort to produce good systems.

Secondly, and this is probably even more important, developing a good LDM is very difficult. The integration of the middleware component into a programming system requires expertise in distributed algorithms and communication services as well as in the programming system being targeted.

For optimality the developer of the LDM need to be able to understand and modify either the runtime system or the virtual machine. Although in principle optimal LDMs can be developed, in practice it is very difficult requiring multi-disciplinary knowledge. Furthermore, having surmounted these difficulties the end result is a spaghetti system, interleaving middleware aspects with ordinary programming aspects. Such a system is difficult to maintain, improve and the entangled middleware component is not reusable.

In LIM systems there is a very clear boundary between the programming system and the middleware; this simplifies the task of developing the middleware enormously. But there are still differences between theory and practice; the most popular, CORBA, is still object-oriented, there is very little support for programming languages of other paradigms (e.g. functional). There exist, of course, also low-level middlewares providing messaging support only, e.g. group communication services (MPI [11], Spread [1]), but these all have limited functionality. Also the division between programming system and middleware would seem to preclude many useful distribution strategies. For instance, for objects to be moved between processes and for objects to be replicated, would seem to require co-operation between the middleware and programming system. This is the case with CORBA, and to our knowledge all LIM systems of today.

In this context some questions arise. Is it possible in some way to:

- develop a LIM where the efficiency gap to a LDM is negligible or nearly so?
- develop an architecture of a LIM that is extendable and maximizes reuse?
- get more or less the best of both worlds, the LIM and LDM?
- design a LIM (or LDM) that contains a full repertoire of the necessary distributed algorithms to achieve good performance under all conceivable patterns of use?

1.2 The Distribution Subsystem

In this paper we will describe a middleware component, called the **distribution subsystem (DSS)** that is intended to give transparent distribution support to arbitrary programming languages. The DSS is truly language-independent; in fact, it is paradigm-independent in that it supports languages of all the various paradigms (object-oriented, functional, logic).

In this paper we present not only a DSS architecture but indirectly an implementation. For the purpose of this paper the implementation is a proof of concept. The implementation is made publicly available for study. The focus of the paper is on the architecture.

The DSS, like any other LIM, has certain functionality that is obtained via the use of the DSS interface or API. However, unlike other LIMs, the DSS is not primarily intended to be used by application developers. Rather the LIM is a middleware component intended to be coupled to programming languages/systems or virtual machines. The result of such a coupling is a distributed programming system tailored to the chosen programming language.

We claim that the end result of coupling the DSS to a VM is a distributed programming system roughly comparable in terms of performance to best LDM systems. The claim is, answering the question of the previous subsection, that it is more or less possible to get the best out of both worlds.

An important aspect of the architecture is the division of labor, the boundary, between the responsibilities of the DSS and the VM. The main design goal was to maximize the responsibilities of the DSS and therefore to minimize the responsibility of the VM within the constraints of language-independence and maximal efficiency. We do require certain instrumentation in the VM, but believe that this instrumentation is kept to an absolute minimum. Further minimization can only be achieved at a price of decreased performance or loss of vital functionality.

The DSS provides or incorporates a rich set of distributed algorithms. There are many that provide the same functionality but are geared to optimizing very different use cases. There are, for instance, many that deal with shared state or distributed objects. This richness implies complexity. However, the complexity is greatly reduced by breaking down the choice of algorithm into a number of more or less orthogonal dimensions. This not only simplifies the internal structure of the DSS greatly, but is also reflected in a simpler API.

Compared to the APIs of LIMs that are intended for use by application developers the DSS API and interaction model are more complex; not unreasonably so, but still more complex. The reason for this complexity is 1) that all (or most) programming languages are covered and 2) the richness given the full repertoire of distributed algorithms. If one consider only those aspects that would be needed to implement standard remoting the complexity would be roughly the same.

However, and this cannot be stressed enough, the greater complexity here does not matter and is not the point. The purpose here is to provide a simple programming model after the DSS has been coupled to VM. The

programming API is very simple. This we hope to show.

It is the complexity of the programming API that needs to be minimized. This is used on a daily basis while the DSS API is only needed *once* per programming language.

1.3 Outline

In section 2 we list the requirements that we make on a (or our) DSS. These requirements are necessary to ensure that the resultant system has the wide applicability property, characteristic for LIMs, and the efficiency property, characteristic for LDMs. In section 3 we characterize our target languages and we shall see that we include virtually all high-level programming languages. These requirements are necessary to ensure ease of use (LDM property) and to make the resultant distributed programming system useful. In section 4 we lay the groundwork for understanding the division of labor between the DSS and the virtual machine by introducing a very abstract semantic model that captures only the semantic features of languages that the DSS need to know.

In section 6 we describe the DSS. First we expose the DSS framework and VM interaction model. The framework also defines a tuning space, a space of distribution strategies. Second we give an overview of the internal of the DSS. Finally we illustrate the way in which strategies within the tuning space may be specified.

In section 7 we deal with the virtual machine. First we cover the minimal extensions that are needed to be able to couple a VM to the DSS and then we give an overview of the coupling methodology. Section 8 illustrate the extendibility of the DSS architecture by example and finally in section 9 we evaluate the architecture and benchmark a system that we built along those lines against existing LDMs and LIMs. In section 10 we conclude.

1.4 Terminology

Here we introduce some terminology that we will use in the rest of the paper:

process. An operating system process. Processes are typically, but need not be, on different machines, connected by some kind of network (LAN, WAN, or, for that matter, wireless).

thread. The unit of concurrency within a single OS-process. Note that this means that the fine-grained processes within an OS-process that some systems offer (e.g. Erlang [2]) will in our terminology be called threads rather than processes.

2 Requirements

The purpose of the DSS middleware is to provide a service that realizes distribution of programming languages/systems. The service is best explained on the level of language entities. Objects, integers, strings, arrays, functions, procedures, classes are all examples of language entities. In a centralized system a computation proceeds by performing operations on the language entities, within a single process.

The service that the DSS provides enables operations to be performed on the same entity from multiple processes (machines). The typical example is the shared object that may be invoked from many processes. Put in another way, the DSS is the service that enables entities to be shared between processes.

Our five requirements are listed and briefly described below. Some of the requirements require more motivation and discussion than is conveniently listed. That is covered in the following subsections.

Generality. Ideally the DSS should support all programming languages/systems.

However, even to the extent that it would be possible to support all languages the result is not necessarily practical. So we will qualify this requirement; within the bounds of usefulness and feasibility the DSS should support all languages.

Lest the reader come to the conclusion that the qualification in any way severely restricts the applicability of the DSS solution, we point out now that virtually all general-purpose 'high-level' programming languages lie within the bounds of usefulness and thus are supported.

A full technical characterization of target languages, i.e. languages that lie within the aforementioned bounds, is presented in the next section (3), together with numerous examples.

Network Transparency The operational semantics of language entities is preserved. Put another way the behavior of language entities is the same, irrespective of whether the entity is shared or not, and irrespective of how the entity is shared, modulo timings, failures and memory requirements.

Control and Tunability It must be possible to control (tune) the way in which the DSS supports the sharing of an entity to obtain optimum performance.

Memory Consumption The requirement on the memory consumption is twofold. First, garbage collection of shared entities should function

in the same way as for non-shared entities. When all references to a shared entity are lost the entity should be reclaimed eventually. Additionally if there are references to an entity then the entity should be protected from reclamation. This requirement is, in general, impossible or very impractical to fulfill to 100%, given the current state-of-the-art of distributed garbage collection [30].

Practicality forces us to relax this requirement somewhat. We will not consider this further here, suffice it to say that the DSS must provide for some reasonable approximation(s) and that there is a clear model of the limitations of the offered distributed garbage collection. In section 6.3.5 we briefly describe the distributed garbage collection techniques used in our DSS implementation and their limitations.

The second requirement is that the memory consumption for shared entities is bounded. This should be considered in light of the fact that we cannot require that the total memory consumption in all processes to be the same for a shared entity as for a local entity. This is just not possible in any state-of-the-art system, where memory usage for local entities and references to local entities is optimal. We do however require that the memory consumption for a shared entity in any one process is bound, irrespective of the entity's history. Formally: $smem \leq lmem + k$, where k is a constant. Together with the first memory consumption requirement this means that the total memory consumption for shared entities (summing together all processes) is at most linear in the number of processes sharing the entity.

It goes without saying that there are practical considerations as regards 'eventually' in the first requirement and the constant factor k in the second requirement.

Fault Detection Support The DSS is responsible for maintaining shared entities so that transparency is preserved modulo failure. Failure may cause transparency to break down. The failure of one process may disable operations for a shared entity in other processes. This must be reflected up to the programming level. Actual detection of process, or link, failure lies outside the DSS in the OS-service [5] component. The OS-service component will monitor processes and links to processes.

The DSS is responsible for 1) mapping process and link failures to entity failures and 2) informing the OS-service component about any changes in the set of links that the OS-service component should monitor.

2.1 Transparency

Transparency is a traditional goal for distributed systems. In the context of programming languages we take network transparency to mean that the operational semantics of language entities is preserved. This means that entities behave the same way irrespective of distribution, modulo timings, memory consumption, and failure.

The advantages of transparency are two, simplicity and ease of use. There is only one model of entity semantics, rather than at least two as would be the case for non-transparent systems (one for shared entities and one for non-shared entities). Secondly, transparency means that the application developer can develop, test and debug distributed applications using exactly the same concepts as he/she would use for centralized applications.

Given that there are so many non-transparent distributed programming systems one would suspect that there are some serious disadvantages to transparency as well. Indeed proponents of such systems do argue against transparency. The two arguments are 1) it is not possible and 2) it is too inefficient.

However it has been shown that modulo failure, timings, and memory consumption, transparency is possible for a number of programming languages. Emerald [17], for example, demonstrated transparency for shared objects in the late 80's, and the Mozart Programming System [4] for shared entities in the programming language Oz, e.g. single-assignment variables, objects, procedures, closures. True, complete transparency (including failure, timings and memory) is not possible, but this does not mean that transparency should be abandoned altogether. Leaving aside failure, for the moment, we see that the functional aspects of (most) programming languages can be made transparent. The advantage of being able to develop, test and debug a program's functionally on one machine remain.

Given that complete transparency is not possible, a final consideration is whether a network transparent solution will, in any way, make dealing with failure, timings and memory more difficult than if we would choose a more non-transparent solution. Within our requirements, concerning control, memory and fault-detection, we implicitly deal with this. We believe that with these requirements working with failure, efficiency and memory issues is, in principle, just as easy (or hard) as it would be using any more non-transparent solution.

The second argument against transparency is that it is too inefficient. In our view this is just plain wrong. The argument is based either on deficiencies in the implementation or in the language. In our requirement concerning control and tunability we deal with implementation efficiency issues, so we will not consider those aspects further here, but instead concentrate on language

issues.

Two examples of non-transparency in Java RMI are reentrant locking and the copying of object parameters in remote method invocation. No one, we think, would argue that transparent reentrant locking is not useful; the argument is rather that this would impose an implementation overhead that is prohibitive when re-entrant locking is not needed. This may not be as true as commonly supposed; the induced overhead (keeping track of thread identities across processes) does not seem to be that great, but there is no getting around that some cost is associated with distributed re-entrant locking. However, this may equally well be seen as a deficiency in the language - if the system had provisions for both non re-entrant and re-entrant locking objects on the language level there would be no problem.

When local objects are used as parameters in remote method invocation they are copied. If all the instance variables are final all is well, but if not there are now 2 different objects that may be updated independently. This is as non-transparent as you can get. A transparent solution would be to make the object parameter a remote object on the fly. Clearly this can be done (but not at compile time). Once again the difficulty is that fairly often a programmer knows an object is stateless. Presumably the object became stateless some time after construction completion, but this cannot be expressed. Now imagine an extension to Java that provides the programmer with the ability to cause objects to become immutable at runtime, where attempts to update such an immutable object cause a runtime error. Properly used this distinguishes objects that should become remote when passed as a parameter and those that can be safely (transparently) copied.

The point here is not that the hypothetical extensions to Java that we made are necessarily the best or even good, but that standard Java does not make the necessary distinctions for us to combine transparency with efficiency. Other languages or dialects, existing or future, may make these distinctions so the DSS must clearly cater for the full spectrum of choices.

While the DSS is specifically designed to support transparent distribution for all kinds of programming languages, this does not necessarily mean that the DSS can only be coupled in a way that will result in a transparent distributed programming system. While we would argue that transparency is a good idea even here, it is not necessary. For example, the DSS will provide support for distributed locks of both the reentrant and non-reentrant kind. It is possible to use the support for non-reentrant locks for locks that are in the language defined to be reentrant, if one wishes. In fact, the standard Java RMI solution can be obtained by suitable coupling.

We do want to stress, however, that when we claim that the DSS provides generic distribution support for programming systems we base this claim on

an analysis of the requirements for transparent distribution for a wide spectrum of programming languages/systems. The DSS is, conceptually at least, complete in that all the required support is made available. However, this applies only in the context of transparency. The space for non-transparent solutions is just too large, if at all finite, for a DSS to cover all varieties.

2.2 Control and Tunability

For any software abstraction, or tool, to be useful it must be reasonably efficient. But reasonably efficient compared to what? Often tools are compared to each other but if the technology is immature we run the risk of short expiration date; new developments will quickly render the comparison obsolete.

Another measure is to compare the abstraction or tool with what could be achieved at a lower level. This measure has been often used to evaluate centralized programming abstractions. For example, we might evaluate compilers by looking at the compiled assembler code, and we might evaluate a high-level language like Java by comparing the performance of applications written in it as compared to a low-level language like C.

What is the low-level counterpart to assembler or C in distributed computing? Here we have two factors to consider. The first is the additional computation that is being performed on all the involved machines. The second is the nature of the algorithm or exchange of messages needed.

Take RMI as an example. When a remote method invocation is made the method invocation needs to be trapped and the invoking thread suspended. The method and its parameters need to be serialized and packaged in a message which is given to the operating system. Ultimately this message arrives at the machine where the object resides, the method and its parameters are deserialized and ultimately given to some thread for local execution. The completion of the method will also need to be trapped, the return value serialized, packaged and finally sent to the original invoking site. Upon arrival the return value also need to be deserialized and passed to the original invoking thread.

Clearly, remote method invocation involves considerably more computation than local execution (on both sites). In our architecture some of this computation lies outside the DSS and some within. In particular, serialization and deserialization are intimately connected to the particular language/system we are using and are thus not part of the DSS. In contrast, the messaging service is part of the DSS. Further discussion of the division of labor between the DSS and the target language is deferred until later. At this point we merely note that some, but only some, of the extra computation

that is required in going from method invocation of local objects to shared objects takes place in the DSS.

To the extent that the DSS is involved in this extra computation we do require it to be reasonably efficient. We claim that our own implementation is reasonably efficient in this aspect, which is supported in the section on evaluation, admittedly by comparisons on the level of competing tools. To us all the work that the DSS does in this very simple scenario is necessary and we doubt any dramatic improvement in performance is possible, at least, say, within 20%. Note that the C++ code in our DSS implementation is also available for inspection.

So far we have only considered the extra computation involved. We now turn our attention to the distributed algorithm aspect. RMI is a, admittedly very simple, distributed algorithm for maintaining the consistency of shared objects. The object is stationary and never replicated and all invocations on the object from other machines involves sending a message to the home site of the object, performing the invocation there and sending back the reply. From the viewpoint of the invoker the time needed for method invocation is increased by two network hops. Depending on the network latency and the granularity of the method the time for method invocation can easily increase by a huge factor.

Given that RMI is based on one particular algorithm for maintaining the consistency of shared objects, the natural question is if there are others. The answer is that there are many other algorithms, involving moving the object or replicating the object. So, is RMI the best algorithm? Given the prevalence of RMI in today's systems one might suspect that the answer would be yes, but the correct answer is sometimes. It all depends on the pattern of use; there is no one best algorithm. There are not only usage patterns where the difference in performance (chiefly the number of hops) is large; the difference may actually be unbounded. For example, (consistent) replication may decrease the network hops from two per invocation to epsilon hops per invocation where epsilon is arbitrarily small.

Our strategy for dealing with this fact is to support all known algorithms or protocols in the DSS. As there is no one best strategy for distribution support for shared objects we offer them all. More generally we want to support all the best strategies for all types of language entities. The goal is that there should never be any motivation for the programmer to move to a lower level and write his own algorithm for anything that is already in the language. (He may, of course, need to write his own distributed algorithms for other things).

By control and tunability we mean that the programmer can choose an appropriate consistency algorithm. Note that whatever choice is made trans-

parency is preserved; the program works, irrespective of whatever choice is made.

Later on when we consider the tuning space in more detail we will see that tuning is not just a matter of performance but also includes some failure and memory aspects. At this point the reader may be thinking that our DSS can only be huge bloated beast, containing everything but the kitchen sink. But this is not the case and this, we believe, is one of our main contributions. In later sections the tuning space is covered in detail and we shall see, among other things, that the complexity of tuning is minimized by division into three more or less orthogonal dimensions.

3 Target Languages

The purpose of the DSS middleware component is to provide a service for a programming system/language, such that the system/language can be extended with transparent distribution. Ideally, one would like to cater to all programming languages and systems. However, though it may be possible to extend all languages/systems, the end result is not necessarily useful or, for that matter, new.

Nevertheless we claim that the DSS can be of great benefit for a wide range of languages. By DSS, here, we include our current implementation, future refinements and other DSS systems built along the lines of what is described in this paper. The DSS has applications in all known programming paradigms; object-oriented, functional, logic and constraint. A very loose characterization of those languages/systems that we exclude is that they are low-level.

We restrict ourselves to programming languages/systems that fulfill the requirements below. The main motivation for the requirements is that without them there would be little point in supporting transparent distribution, as the result would be of very limited use. Each requirement is discussed in more detail in following subsections. In the last subsection we give numerous examples of both suitable and unsuitable programming languages/systems

- Concurrent
- Reference Secure
- Data Structuring
- Distinction between stateless and stateful data structures
- Automatic Memory Management

3.1 Concurrency

We require that the programming system/language is concurrent, i.e. a program or a process is not limited to a single thread of execution. Consider what would transparent distribution mean, for a single-threaded system? In the distributed system we would still on a conceptual level be limited to a single thread; the only difference is that this 'thread' would not necessarily be bound to a single process, but could move around the network. As operations are synchronous in the centralized case they should be synchronous in the distributed case.

An example of such a system would be a Remote Procedure Call [3] enabled network where, at any one time, execution is limited to a single process. This is clearly an uninteresting scenario, and of limited use. In particular, when the single thread of execution has moved away from a process (e.g. a RPC invocation), then that process must become idle. There is no other thread that may be scheduled to take advantage of otherwise unused processing power.

If we look at application trends concurrency seems to be more and more important. Also concurrency and distribution usually go hand in hand. In most client/server applications, for example, servers are typically multi-threaded.

3.2 Reference Security

The middleware transparently makes data available to threads, disregarding their location. The challenges, for the middleware constructor are very different depending on whether or not reference passing between threads is explicit (e.g. Java [10]), or implicit (e.g. C [18] with pointer arithmetic). If reference passing is implicit, i.e. if references can be guessed and forged, then potentially all data is shared. The middleware must then be constructed so as to be able to, on the fly, coordinate data transfer or exchange between arbitrary processes. Clearly this does not work for open distributed computing, where processes are not necessarily known to one another.

There does a middleware exist that can handle guessed and forged references? These systems are distributed shared memory (DSM) systems. A DSS that is unable to leverage reference security would turn out to be very much like DSM. This would not be new. Also, because all data is potentially shareable and potentially stateful DSM must make use of expensive consistency protocols on all data. This is impractical without fast multicast so their use is restricted to LANs.

Explicit reference passing makes it possible for a machine to safely di-

vide data between local data, which is referenced only by threads on that machine, and shared data, which is referenced by threads on other machines as well. It takes an explicit action by the process to move data currently local to that process into the shared area. Irrespective of what is going on in other processes, as long as this action is not taken, other processes can never access the local data. The distinction between local and shared data is often crucial for efficiency. Operations involving only local data can be done locally, and there is no need for any coordination between processes. Furthermore middleware does not even need to know, and should not need to know, anything about the local data.

3.3 Data Structuring

This requirement is superfluous in that it is standard today, and basically met by all languages that meet our other requirements. We mention here it for completeness, and because it, like the other requirements, is a property of a programming language that we leverage in the DSS for performance. The requirement is that the language provides for composite data structures. These data structures are composed out of simpler ones, and though they may be decomposed, they are for most purposes dealt with by the system as if they were a single programming entity. Objects, classes, records (Oz [4]), tuples (Erlang [2]) are all examples of data structures.

3.4 Distinction between Stateless and Stateful

This requirement is, we believe, essential for efficiency. Stateless data is so much simpler than stateful data to deal with. When references to stateless data are passed from one process to another the data may be safely replicated. Maintaining the consistency of stateful (mutable) data is, however accomplished, expensive.

Ideally, data that is stateless should not masquerade as being stateful. For transparency stateful data should never masquerade as being stateless, but the converse does not hold; stateless data can safely masquerade as being stateful in all languages that allow for both stateful and stateless data. More realistically, it should be possible (and hopefully natural) to express that data, or data structures, are or have become stateless and immutable. In Java an object where all instance variables are final is stateless after construction. Thus it is possible to express statelessness in Java. More flexible mechanisms would be desirable on the language level. For instance, it would be advantageous to be able to cause an object to become immutable at any time, not just directly after construction.

3.5 Memory Management

Explicit memory management is a complicated and error-prone task in a single threaded system and even more so in a concurrent system. A common and well accepted solution is to use some sort of automatic memory management, called garbage collection [16]. Since transparent distribution in effect will introduce more concurrency, the argument for automatic memory management is strengthened.

This requirement, unlike the others, is possibly not truly essential. We were much influenced by the fact that the languages we had in mind as targets all have explicit memory management. Throughout our work on the DSS, we assumed automatic memory management. We have not given any serious consideration to the explicate memory management alternative. Presumably it would be possible to support a distributed delete operation, as well as distributed dangling references, but we doubt this would be useful.

3.6 Example languages/systems

The service provided by the DSS is intended exclusively for programming languages/systems that fulfill the aforementioned requirements. Many languages meet the requirements. For instance, Java [10], C# [31], Oz [4], Erlang [2], Haskell [13], Scheme [35], CAML [15], Mercury [26] are all examples of programming systems that could successfully use the DSS.

Examples of languages that are not suitable are C [18], C++ [7], and any kind of assembler. In particular pointer arithmetic in C and C++ make them totally unsuitable.

We claim that all languages/systems that fulfill our requirements can be extended with transparent distribution by coupling the system to the DSS. The result will be a distributed programming system (DPS) tailored to the chosen language.

4 An Abstract Model of Language Entities

The targeted languages represent information in the form of data structures. Each data structure is associated with a type; the type gives the data a semantical meaning in the form of a set of valid operations that can be performed on the data. We will use the term language entity (or just entity) for this data structure. Not only are data structures regarded as entities, but also code; thus procedures, functions, and classes are all entities.

The joint set of entities found in all targeted languages is large. If we further distinguish language entities by semantics this set will grow even

larger. Different languages can have entity types with the same name but with different semantics.

Our approach to transparent distribution is to enable distribution of language entities. The middleware component should expose an interface that makes it simple to share all entities of a language between different processes. Sharing an entity in this case refers to upholding an entity's centralized semantics when shared among multiple processes. From the above discussion we see that the set of semantical entities that have to be supported to be very large. Supporting every type would seem to be impractical.

Instead of supporting every single entity type in all the target languages, we take an approach using abstract entity types. We argue that the semantics of a language entity can be divided in two parts; the larger part, of which need never be exposed to the DSS at all. Only a small part of the semantics of a language entity needs to be exposed to the DSS. This is what we mean by **abstract entity type**. Furthermore the number of different abstract entity types is very small, no more than a dozen or so. Just as concrete operations may be performed on a concrete language entity abstract operations may be performed on abstract entities. Once again this is a many-to-one mapping. One single abstract entity operation may correspond to many different concrete entity operations.

We first present an exhaustive description of the abstract entity types found in our target languages. One of the major tasks when a language is coupled to the DSS is to map concrete language entity types and operations to abstract entity types and abstract operations. This is discussed at the end of this section.

4.1 Synchronous Stateful

Entities with mutable state can be modeled by the **synchronous stateful** abstract entity type. The semantics that we want to preserve is sequential consistency for read and write operations. All operations are synchronous from the perspective of any calling thread; the thread is blocked until the operation is completed. This abstract entity offers the following abstract operations:

update Alters the state of the abstract entity. When the operation returns, the state of has been updated.

access Reads the state of the abstract entity. The value returned is the latest value of the entity's state.

Synchronous stateful entities can be found in almost all programming languages, with the exception of some logical and functional languages. An instance variable of an object in Java is an example of a synchronous stateful entity. Depending on the level of granularity, the whole object can also be seen as a synchronous stateful entity.

4.2 Asynchronous Stateful

The asynchronous stateful abstract entity is similar to the synchronous in that it has a mutable state. However, it differs in that the update operation is asynchronous instead of synchronous. The operations are FIFO with respect to the performing thread (i.e. a form of process consistency). The order of operations originating from two different threads is unspecified.

update The abstract operation alters the state of the entity, eventually. It is not known if the state of the entity has been updated when an update-operation invocation returns. It is known that a reader will observe two consecutive operations from the same thread in the same order.

access The current value of the entity is read.

Message-sending entities can be modeled by the asynchronous stateful abstract entity. The ports of Mozart [4], and the process references in Erlang [2] are examples of two entities that model message sending; both can be modeled by the asynchronous abstract entity type. Note that there are differences between Mozart and Erlang when it comes to how the mailbox is organized (i.e. what exactly the update operations do and how received messages are retrieved) but this difference is abstracted out.

4.3 Monotonic

Entities that make monotonic transitions are semantically depicted by the **monotonic abstract entity**. These entities have three states, untouched, touched, and bound. It can monotonically move from untouched, then to touched and finally to bound. The entity travels through its states as a result of operational manipulation. Multiple instructions to make a certain state transition, will only move the entity into the new state once, the first such operation will win. Latecomers will be dropped. All reference holders will be informed about transitions.

touch Moves the abstract entity from the state *untouched* to *touched*. If the entity is already in the state *touched*, or *bound*, the operation is

discarded. The operation is synchronous, when it returns; the abstract entity has done the state transition.

bind Moves the abstract entity from the states *untouched* or *touched* to *bound*. Similar to the abstract operation *touch*, the operation is discarded if the abstract entity is already in the state *bound*.

Futures, logical variables, and by-need constructs are examples of entities whose semantics are described by the monotonic abstract entity. Futures and logical variables starts in the state *touched*, and will only do the transition to *bound* when their value is determined. The *touch* construct is used by the by-need entities when asking for the value of the entity.

Resolving possible conflicts is abstracted out for the DSS, i.e. is handled locally by the virtual machine. For instance, consider the case of multiple bindings of a single-assignment logical variable. Only one of the bindings will take place. The loser of the race condition will be informed and can then repeat his operation; the end result is the same as if the binding had come from another thread on the same machine. For those not familiar with logical variables and their use in programming we should point out that the race condition is not harmful. Conflicting bindings is a programming error like dividing by zero and hopefully rare. Multiple compatible bindings, on the other hand, are legitimate and with multiple compatible bindings the end result is the same irrespective of order.

4.4 Stateless

This abstract entity represents entities that never alter their state, thus the state can only be read. These entities are commonly refereed to as stateless, or immutable, thus they only have one type of operation.

access Accesses the immutable state of the stateless abstract entity.

Code (classes, procedures, functions) is stateless. Other stateless entities are atoms, integers, floats, records, and lists. Java objects with all instance variables defined as *final* are also stateless, as are objects without instance variables.

4.5 Lock

A lock is an abstract entity type that is different from the others abstract entities in that it imposes control on the threads that performs operations on it. The lock has a state that can be taken by one thread only. A thread

that attempt to take an already taken lock is suspended, until the lock is freed for him (in order). We recognize two subtypes of locks, reentrant and non reentrant. With a reentrant lock the holder of the lock can take the lock multiple times.

take The abstract operation tries to acquire possession of the lock. If the lock is already taken by another thread, the calling thread is suspended, put into a queue and woken up when it is given the lock.

release The release operation release the lock and makes the lock available to other threads or gives the lock to the first queued thread.

Locks in Mozart and synchronous object methods in Java can both be realized by reentrant locks.

4.6 Entity Types and Annotations

The semantics of a concrete language entity type will determine to which abstract entity type it is mapped to. When previously local language entities are about to become shared between processes, the DSS will need to be informed about the entity's abstract entity type.

In order to cater for the tuning, as described in section 2.2, the DSS needs additional information. We call this extra information the **entity annotation**. The rationale behind the name is that the most flexible arrangement for tuning, as section, is via annotations of single entities. This lets the programmer tune on the level of single entities; entities of the same type may then have different annotations. An example for objects would be a language where annotations are passed as parameters to the constructor.

Less flexible arrangements are also possible. Another mechanism is to annotate by type on the language level. In Java this might be achieved by interface inheritance. Less flexible still, and not recommended as then no tuning will be possible, is to use the same annotation for all entities of the same abstract entity type. As the provisioning for annotation is done either on the language level or in the 'glue' code between the DSS and the virtual machine all possibilities are catered for.

4.7 Classifying Language Entities

We have noted previously that semantically there is quite some freedom when classifying entities, even without relaxing transparency. An entity may always be classified as having stronger consistency guarantees than it actually needs. A stateless entity may be classified as being monotonic or stateful,

and a monotonic entity as a being stateful. We also said that, in general, this was not a good idea. In particular, that it is undesirable that stateless entities masquerade as being stateful.

But this is not the whole story. Up till now we have viewed language entities as monolithic self-contained entities without structure. Looking at entities on the language level an entity may contain references to other entities. That stateful entities may reference other entities is of no concern here they are part of the state of the entity. Of interest is that stateless entities may reference stateful entities, possibly via some other intermediary stateless nodes. The stateless entity may be seen as a wrapper for the stateful entities.

Let us first consider the case where the stateless entity wraps exactly one single stateful entity. We can represent this faithfully to the DSS or use the semantic freedom and classify the wrapper as being stateful as well. The major difference between the two is that the stateless wrapper together with the stateful entity will now from the DSS point-of-view be a single stateful entity so it will no longer be possible to access solely the stateless wrapper without involving a state consistency algorithm. In general this is not desirable, though if accessing the wrapper by itself is rare then the two models are more or less equivalent.

The interesting case is when the stateless wrapper indirectly or directly references more than one stateful entity. Here there are major differences. If the stateless wrapper is classified as being stateful then all the contained references will belong to a single stateful entity, rather than many different stateful entities. We exemplify with an array.

Consider an array of integer values. The array has a fixed length or fixed bounds and is composed of a vector of mutable cells that contain integer values. The stateless wrapper might contain information as to its type and length or bounds. The values are accessed using indexing, where each index refers to a particular cell in the array. Note that while there may be operations on arrays that only need to access the stateless wrapper they are rare. For instance, the array may be asked about its length or bounds.

Depending on the granularity that we want, the array can either be classified as one stateless construct referencing a set of stateful cells or as one monolithic stateful construct. In the first case the array indexing is then done on the stateless part of the array, returning a stateful cell. It is the cell that is actually read or written. In the second case the access is done on the array as a whole.

In the first case the state consistency algorithms will be run on the various cells independently. In the second case there will be one state consistency algorithm running on the array as a whole. If the array is stationary there is little difference in performance, but if the more sophisticated replication or

movable state schemes are used the differences can be arbitrarily large.

Once again there is no one optimal strategy; it all depends on the application. The potential advantage of modeling the array as individual stateful cells is that working with different indices in different processes need not interfere with each other at all. The disadvantage is the when many or most of the indices are required at one process this will require the action of many consistency algorithms, i.e. many messages (even though the number of hops might be the same).

The DSS makes no assumptions on how arrays are modeled. Potentially the language may offer both kinds of arrays via entity notations. Furthermore, it is also possible, though this requires more instrumentation on the virtual machine level to decompose an array into suitable sized blocks (e.g. dividing an array of 1000 into 10 blocks of 100 elements each).

5 The OS Service Component

The DSS is designed to be platform and network environment independent. By platform independent we mean that the DSS can be deployed on any machine, irrespective of its operating system. By network environment we mean that the DSS does not rely on any constraints on the topology of the underlying network.

All functionality that can interfere with the two above independences have been lifted out of the DSS and located in a separate module, the OS-service component. The environment where a DPS is deployed is presented to the DSS in a uniform way over a well defined interface.

The OS-service component has three primary tasks:

1. Access to communication channels in the form of an abstract UNIX socket. The socket, or channel as it is called, offers non-blocking asynchronous communication and callback interfaces, that allows the DSS to be informed when data can be read or written from a virtual socket.
2. The OS-service component is responsible for opening channels to remote processes and to accept incoming connection requests from remote processes. This involves both address resolution and connection establishment. Both of these are operating system dependent.

Every process has a description of how to establish a connection to it, called a virtual address. When trying to establish a connection to a process, the virtual address of the target process is needed. In its simplest form a virtual address is nothing more than an IP-address

and port. More sophisticated versions of virtual addresses cater for process mobility, authentication, etc.

3. The OS-service component is responsible for fault detection on the level of processes (and links) and supplies the information that the DSS needs to detect faults on the level of language entities. The motivation of putting this aspect of fault detection in this component rather than the DSS is twofold. First, there are aspects of fault detection that are OS-dependent or transport protocol dependent. But there are also aspects that are environment dependent. As an example of the last, consider what a connectivity loss for a few seconds might indicate in different environments. For two computers on the Internet a two second loss is not unusual; in a cluster such a loss indicates a serious problem. The role of the OS-service component in fault-detection will become clearer when we present fault detection in the DSS in section 6.2.6.. We will also see that the DSS does provide some useful functionality that can aid the OS-service component to detect faults.

It is probably advantageous to include in the interface between the VM and the OS-service component methods to instrument fault-detection. This will allow applications some control over fault-detection. For instance, given that the round-trip time has been 10ms for some time, a sudden delay of 100ms is for some applications an indication of a problem, but perfectly normal for others.

In addition to the above the OS-service component can, of course, be used to hide non-communication related differences in operating system services, e.g. file services.

6 The Distribution Subsystem

The most important functionality of the DSS is the support for sharing entities between processes, or between threads on different machines. A central design criterion was that this support should realize full network transparency for such shared language entities. The DSS supports the sharing of entities on the level of abstract language entities (see section 4). The mapping between language entity types to abstract entity types is done when the DSS is coupled to the VM to create a Distributed Programming System (DPS). This can be done once and for all for any one particular programming language or VM. There is some freedom in this classification but not much. If language entities that are stateful are classified as being stateless, then irrespective

of all other instrumentation of the DSS, the resultant DPS will become a semantic mess.

Nevertheless, within the constraints given by the abstract entity type, there are many possible strategies that the DSS may take to maintain consistency of shared entities. Leaving aside obviously poor strategies, many strategies remain, where each is best under certain patterns of entity usage. Our view is that therefore the DSS should supply them all. To each entity the application developer may choose his sharing strategy. Choosing sharing strategy may be done on an entity for entity, application for application, basis. Different entities of the same type, abstract or concrete, may use different strategies.

In this section we present the DSS. We begin with the general framework that the DSS uses to maintain entity consistency. Some of the framework is fixed and some merely defines the limits within which various system components work. This will lay the ground for a description of sharing strategies toward the end of the section. In between we give an overview of the internals of the DSS.

6.1 The DSS Framework and VM interaction model

6.1.1 Sharing Entities

Within a VM references to an entity are typically represented as pointers, pointing at one unique entity structure. The only exceptions are simple primitive data types. Threads (within one VM or process) can then share entities by passing references or pointers between them. More exactly, the reference is duplicated or copied and the copies must point or refer to the same entity structure as the original. All the references, the original as well as all the copies, are the same, none is especially privileged. In particular, the cost of operations by threads on entities does not depend upon which particular copy of the reference that the thread obtained.

Now we turn our attention to the distributed case, when entities are shared between VMs or processes. Clearly intra-VM pointers will no longer work. Providing for reference sharing over VM or process boundaries is our first challenge on the journey toward transparent distribution of entities. As stated in the requirements, references may be shared between threads, irrespectively of the current location of the threads, i.e. irrespectively of which VM they are currently residing in. We now turn our attention to how this is achieved.

The most straightforward approach for non-stateless entities and one that is taken by most middleware systems is to distribution-enable the references.

References from one VM to another are remote references, and such references are no longer simple pointers, but have additional logic at *both* ends of the pointer in order to be able to interact with the entity remotely.

One property of this model is that references are inherently unequal. References from the same VM where the entity structure is located are more privileged than remote references, and operations using the privileged references are inherently faster. In this model there is only one entity structure and its location cannot change, it remains located in the VM that created the entity to begin with.

An alternative model is replication. Not only is the reference replicated but also the entity structure. Most middleware systems use this mechanism when dealing with stateless entities [25] [31]. An example would be when a data structure (modeled as an object) is passed as an argument in a remote method invocation. This approach is problematic when the entity is not stateless due to consistency requirements. Note that in this case that the references will in the end reference local copies and all references become equal with essentially the same rights.

6.1.2 Guarded Replication

In our model we also replicate the entity structures. A crucial addition is that a guard is placed in front of the entity structure. Guards are not only placed in front of the replicated entity structures, but also the original entity structure. We should point that the guard is more on the conceptual level; it does not necessarily represent a wrapper object, and can be implemented at low cost. The guard together with the entity structure is the local representative of a shared entity, in a VM. Thus a shared entity has from the viewpoint of the VM the same local representative in all VMs. Conceptually the local representative of a shared entity is in each VM composed of the pair, the guard and the entity structure. Note that the entity structure of a shared entity has the same form in memory of the VM as it does when the entity is strictly local. The strength of this model is that the VM performs operations on both local and, when allowed, shared entities in exactly the same way. No new types are introduced in the VM.

The purpose of the guard is to ensure consistency and network transparency. When threads attempt to do operations on a shared entity the guard may or may not let the thread perform the operation. The guard is under the control of the DSS and will always block thread operations on an entity until such a time that the operation may be done without violating network transparency. The division of the labor between the VM and the DSS is exactly this. The DSS controls when and where the operation is to

be performed while the VM performs the actual operations.

When a thread attempts to do an operation on a guarded entity it must ask the DSS for permission. This is an essential feature of the interface between the VM and the DSS. The DSS may respond in one of three ways:

proceed The DSS may tell the thread to *proceed*, in which case the thread executes the operation exactly in the same way that it would had the entity been local. The DSS is designed to only give this go-ahead when operational semantics are preserved.

skip The response is typically used when the operation will be performed in another VM, e.g. remoting.

suspend The response is typically used when performing the operation would violate consistency. Later, when the situation has changed, through the actions of one or more DSS systems, the thread can be woken and the operation resumed.

The standard remote reference model is one of many possible instantiations of our general model. In this case the guard of the original VM, hosting the entity, will always return *proceed*, while all other guards (in other VMs) will always return *suspend*. Later, by a mechanism that we have not yet covered, the operation will actually be performed at the original VM. Later, still, after the DSS with the suspended threads receives notification that the operation has been performed, the suspended thread can be woken. This time the DSS will respond with *skip*. We will return to this scenario later after we have laid some additional groundwork. We will then describe in detail exactly how remote operations are performed; for now note that in this scenario the guard will never let the operation be performed locally (except in the hosting process).

The straight replication model is another instantiation of our model. In this case guards trivially return *proceed* on all operations. The two models, remote referencing and straight replication, are at the two extremes. In one case the guard always lets the operation be done locally, in the other never (except in the home VM). Later, we show that there are other interesting and useful instantiations of our general model where the DSS of one process will sometimes let the operation be performed locally and sometimes not.

Our model, guarded replication, might seem to imply certain inefficiencies. After all, why should we replicate the entity structure if we want to implement standard remoting where the entity structure behind the guard will never be needed. However, and this will be covered in more detail in the next section, avoiding copying the entity structure unnecessarily may be seen

as an optimization. The principle behind the optimization, which has many more applications than in standard remoting, is that the entity structure does not need to be complete. Depending on factors of entity type, protocol, and timing it may be absent, out-of-date, or only partially instantiated - as long as it is then protected by the guard from actually being used.

6.1.3 Passing References

In the centralized VM threads may acquire references to entities that they previously did not have. Apart from bootstrapping (see 6.1.11) this is achieved by operations on previously shared entities. An example is when an instance variable of a shared object is updated to reference a new entity. All threads that have references to the shared object can now access the instance variable, and thus acquire the new reference.

We can distinguish the exposing thread from the importing threads. In our example the thread that updated the instance variable is the exposing thread, i.e. it exposes a new reference (or more precisely reference instance) to other threads. The reference in the instance variable is now potentially made available to other threads, the importing threads. We make three observations:

1. Upon exposure the new reference instance is instantly available to the importing threads.
2. The reference in the instance variable need not be a new reference from the point of view of an importing thread - all threads that share the object may already have a reference to the same entity acquired previously.
3. The importing threads might never access the instance variable, or might only access it after an additional update in which case the importing thread will never need to use or see the reference or corresponding entity.

In a distributed setting, references may now be passed between threads residing in different VMs. More precisely a thread in one VM exposes a new reference instance to threads residing in other VMs. Consider figure 1. An entity C is shared between two VMs. The entity C is a data structure that in turn contains a reference to another shared entity C2. In addition in VM A there is a thread that references both C and some other entity A1. This thread performs an update on entity C (which is clearly not stateless) so that C now references A1 instead of C2. Our guarded replication model requires that information needs to be passed from VM A to VM B.

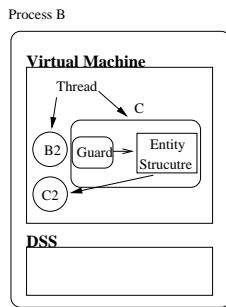
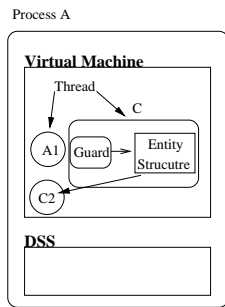


Figure 1a

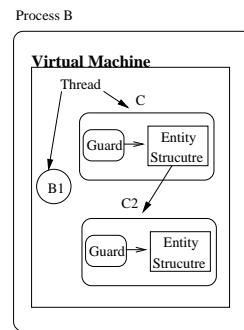


Figure 1c

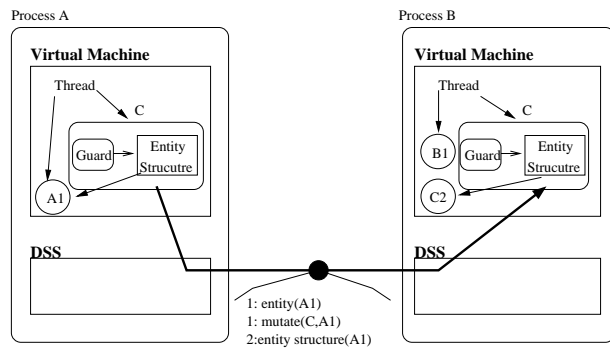


Figure 1b

Figure 1: Guarded and controlled replication

The sending of the necessary information is done under the supervision of the two DSS systems. The information, that in general, is needed may be divided into 1) the identity of the shared entity A1, 2) the update of the entity structure of C, and 3) the entity structure A1. With this information process B can ensure that its local representative of C and A1 are properly and correctly replicated.

Clearly when and if the importing thread attempts to perform operations on the entity A1 and is permitted to do so by its guard, then the entity structure must be fully instantiated, consistent and well-formed. The required information has been gathered at process A, sent to process B, that process B has received the information, and finally that process B has had the time to update and build the appropriate entity structures. This is under the control of the DSS systems working cooperatively. The DSS of process B controls the guards in its VM to ensure that operations on the entities are not performed until the necessary steps have been taken.

Returning to our first observation on sharing within a single VM, we note that exposure no longer goes hand in hand with availability - the update of entity C in the figure in VM A does not immediately make A1 accessible to threads in VM B. In fact, to begin with VM B knows nothing about the update. Usually, as would be the case if entity C is a shared object, consistency requires that the guard on entity C on VM B is already initialized to block operations before the update operation can be performed at VM A. Further discussion on this point is delayed until the various protocol options are considered.

6.1.4 The At Most One Copy Property

We turn our attention to our second observation from above, that the so called new reference might not be really an unknown reference to the importing thread. The same observation may be made on the level of the virtual machine. When a thread in one VM passes a reference to a thread in another VM the corresponding entity may already be known to some thread in the same VM or it may not.

An important consideration upon repeated imports of the same reference is whether to allow multiple copies of the same entity structure within the same VM. It is clear that in general this is not a good idea, indeed, taking an object-oriented view where the only sharing between threads or VMs is on the level of shared objects, this is given. Without the one copy per VM property there would be consistency problems with replicated objects even within one VM. Aside from consistency issued there are two other strong arguments for the at most one copy per VM property. Firstly, duplication of entity

structures within a VM would consume more memory, and secondly, it would complicate equality tests (token equality) for those VMs that implement this by address comparison.

Upholding the at-most-one-copy-per-VM property requires shared entities to have unique ids. Naturally we do support this property in our model. The details of the naming scheme that provides this is described in 6.2.2. However, a case may be made for not supporting the at most one copy per VM property for certain types of stateless language entities. With stateless entities you do not have a consistency problem, and if the size of the corresponding entity structure in memory is small avoiding the overhead of naming and reimportation determination may be considerably more important than increased memory usage. Once again, this may be seen as a special case optimization of our general at-most-one-copy-per-VM rule.

6.1.5 Lazy Replication

Our final observation from sharing between threads within one VM was that it is quite possible and may even be common that the importing threads actually never make use of the exposed reference. It is clear that in such case that our distributed system would be sending messages, some of which are potentially large, unnecessarily. This brings us to the question of introducing laziness in our replication model. The two extremes are 1) propagating updates in entity structures as quickly as possible and 2) making propagation demand driven so that the information is only propagated when some importing thread actually attempts to use the entity structure. There are clearly some tradeoffs here. The most obvious consideration is that laziness increases latency but may decrease bandwidth use. In addition, laziness may also increase memory usage and failure sensitivity. In practice the most important factor in practice will usually be the probability of unnecessary propagation, which is partly application dependent and best discussed in the context of sharing strategies. For now, we merely note that the DSS should (and hopefully does) offer a full selection of meaningful levels of laziness.

Referring again to figure 1 laziness may be reflected in two ways. We may, as we have seen, delay sending all the information associated with the update. In addition we may delay sending the information needed to construct entity structure A1. At the importing process we may use the partial information, and update the entity structure of entity C. Entity A1 cannot then be constructed in its entirety. Conceptually entity A1 will then be strictly guarded; the guard will not pass any threads. The guard may be seen as either referencing nothing or, alternatively, a partially instantiated shell containing some minimal information as to the entity. An example

would be when the shell knows only its type. In real systems the actual representation depends on what the VM can offer. The laziness option is reflected in the numbering. The first two pieces of information (numbered 1) may be sent between the processes in figure 1, long before the third piece (numbered 2).

6.1.6 Completing the Framework

Figure 1 represents those situations where operations are first (and possibly only) performed in the VM of the thread that initiated the operation. The other possibility, is for the operation to be performed remotely first (and possibly only there). Such a case would be a slight variation on figure 1b. The only difference would be that there would be no actual update in entity C in process A. The same message(s) between process A and process B would be sent.

Referring again to figure 1b there are two more important observations that we need to make in setting the stage for a more detailed description of the DSS, and how it works. The entity A1 that was exposed by the operation at VM A might be local or already be shared. In the local case the DSS will, in general, need to initialize various structures that it needs in order to be able to handle the shared entity. This is called globalization and is dealt with in the next section.

There were 3 pieces of information in figure 1b that was sent from process A to B. The first piece, (entity(A1)) can be seen as lying solely within the province of the DSS, containing information about the identity of the entity A1. The second and third pieces are mainly within the province of the VM. Only the VM knows how to update entity structures and construct new ones. The DSS receives the messages, interprets them, and using the interface provided by the VM instructs the VM. This illustrates that creating and interpreting messages between processes is, in general, a co-operative activity involving both the DSS and the VM.

We have now laid the framework for understanding the interaction between the VM and the DSS. The model could be characterized as sometimes tweaked, sometime lazy, controlled guarded replication. Replication is controlled by co-operating DSSs. The property of at-most-one-copy per VM is upheld throughout, excepting one useful optimization.

6.1.7 Globalization and Coordination

Globalization is the name for the various initializations that a DSS must take when a local entity is first made sharable. This may take place whenever a

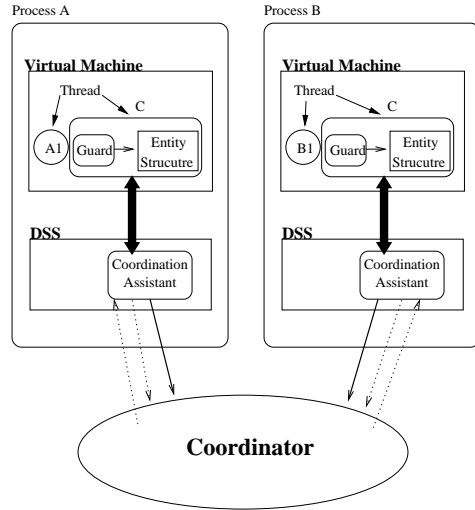


Figure 2: Coordination Network

local entity is exposed to a thread residing in a different VM. As was pointed out in the last section, laziness is allowed in some protocols, and globalization may be delayed, or even avoided altogether. An example of avoidance would be when an instance variable in a shared object is updated to reference a local entity, but before there is a need to globalize the entity, the instance variable is updated again, and the entity is no longer accessible to threads in other VMs.

During globalization the following takes place:

1. A guard is placed on the globalized entity
2. A coordination assistant is initialized in the DSS
3. A coordinator is initialized
4. A connection between the entity and coordination assistant is established

For each shared entity there is exactly one coordination assistant per VM that references the entity. Conceptually there is only one coordinator per entity, but the coordinator may be replicated for fault-tolerance. One of the basic choices that DSS provides, a choice that may be made on a per entity basis, is the type of coordinator. We recognize at least three different types of coordinators, these are covered in section 6.1.9. Coordination assistants always know their coordinator(s). Coordinators, depending on the particular

protocol, may know zero, one, some, or all of their coordination assistants. This is illustrated in figure 2.

Coordination assistants control the guard of the local representative that they are linked to. The responses to VM-initiated operations on guarded entities (*proceed*, *skip* or *suspend*) are made by the coordination assistant. The response *proceed* would typically indicate that the coordination assistant is in such a state that it knows that operations can safely be done on the entity. The response *suspend* would typically indicate that the coordination assistant must contacts its coordinator, either to get the operation done or to initiate a protocol that will eventually put the coordination assistant (and the VM) in such a state to let the operation be performed.

Point 4) is crucial but tricky. It represents the fact that the DSS must be able to interact with the VM and pass back to the VM something that the VM can use to find the particular entity that the DSS is addressing. Similarly the engine must be able to interact with the DSS and address the coordination assistant that is coupled to a specific entity. We will consider this in detail in next section, when the VM and coupling is presented in detail. There are a number of possible variations in how coupling can or is best achieved, and there are some (minor, not major) trade-offs between efficiency and ease of coupling. For now we will take a more abstract view and call these seemingly encapsulating breaking pointers **virtual pointers**.

6.1.8 Memory Management in the DSS

The typical life of an entity is that it is first created in some VM or process. The entity is then local. Later on the entity is shared between thread on different VMs and is globalized. The entity may exist in this shared state for some time, with references being passed back and forth between different processes, and with references going out of scope within a VM. Both the number of VMs, and also their identities, referencing the entity is, in general, changing continuously. Eventually, there will be zero references to the entity left. But before that there will be the situation where there is only 1 VM reference remaining.

It is desirable that the memory management facility of the DSS should recognize both of these situations. Recognizing 0 references is recognizing garbage in a distributed system. Algorithms that guarantee the garbage is always correctly detected may either be unpractical or unavailable. Memory management is considered in more detail in section 6.3.5. For the system as a whole to be practical the memory management unit should definitely detect garbage in most cases, and probably all garbage in most applications.

Correctly detecting that only 1 VM reference remains, is probably less

critical than garbage detection, but often comes for free, or almost for free, on top of garbage collection algorithms. Upon such detection, the DSS may, if this is included in the sharing strategy, reverse the globalization process. This is called **localization**. The coordinator, and the one remaining coordination assistant are deleted, and the guard on the entity structure is removed.

Localization is often useful. There are three possible developments when 1 VM reference is detected. 1) The entity is about to become garbage - the last VM referencing the entity has just not detected it yet. The effect here is that reclamation of memory in the DSS is done earlier than it otherwise would be. 2) The entity will be used again by the local VM, and then become garbage. The effect here is to reclaim memory in the DSS early, and very slightly speed up the remaining operations on the entity; these can be performed unguarded. 3) The entity will be shared again. In this case localization may be disadvantageous, at least if the entity becomes shared again quickly, as new coordination units will need to be constructed shortly after the old ones were discarded.

6.1.9 Coordination and Protocols

The coordinator(s) together with all of its linked coordination assistants may be viewed as a small distributed system linking together a set of processes. Often the **coordination set** is very small. In particular, the coordination set is usually much smaller than the set of processes involved in the application, i.e. the union of all such coordination sets. The coordination set is not static but is continually changing as references are passed between processes, and references go out of scope.

The coordination set together with the links between coordinators and coordination assistants form the **coordination network**. The coordination network may be even more volatile than the coordination set, as links between coordinators and coordination assistants are made and removed through the action of protocols.

The coordinator is responsible for maintaining the consistency of the shared entity that it is coordinating. Depending on entity type, operation, and sharing strategy the coordination assistant in one VM may need to consult the coordinator on every operation, some operations, or even no operations. The coordinator is also responsible for memory management, i.e. detecting the fact that there are only 0 or 1 VM references remaining.

The simplest coordinator is a software component that is initialized and stationary in the DSS of the process where the entity was globalized. Coordinators may also be more or less mobile, or may be replicated. This is reflected in the choice of coordination strategy. This choice is one of the

many factors when choosing the sharing strategy for an entity. Coordination strategies are covered in section 6.3.3.

When references are passed between sites the associated message must contain enough information to build a coordination assistant in the importing DSS. In addition there must be enough information in the message to enable the coordination assistant to find the coordinator. Note that the message contains an embryonic coordination assistant; as soon as the message arrives the coordination assistant will be built. The coordinator must take this into account. Even in those cases where the coordinator needs to know the identity of all its assistants eventually, this cannot be done atomically; there may be coordination assistants, embryonic or real, that the coordinator does not yet know about.

Consistency of a shared entity is managed by a protocol or distributed algorithm that runs over the entity's coordination network. The DSS supports many different protocols. Why? To begin with, the consistency constraints for different abstract entity types are different, and in general the more dynamic the entity state is the heavier the needed protocol is. It is, for instance much easier in terms of the number of messages and latency to manage the consistency of a logical variable than an object. A logical variable can only change its state once, while an object can change its state any number of types.

But this is not the end of the story. There are a number of different known protocols or consistency algorithms for the same abstract entity type and none is best under all circumstances. The best algorithm depends on both the usage pattern of the entity and in some cases its size (or more exactly, the size of the serialized representation). This is why the DSS offers a full, we think, choice of protocols, so that the application developer may choose the best one for his application.

6.1.10 Cooperative Memory Management

Memory management in the VM of globalized entities cannot solely be the responsibility of the VM; rather it is a co-operative activity, involving both the DSS and the VM. In particular part of the root set for garbage-collection lies in the DSS, in the virtual pointers that the DSS holds. For example, as described previously, coordination assistants have virtual pointers to VM entity structures. Note that the fact that the DSS holds a virtual pointer to an entity does not necessarily make that entity a root (but might), it all depends on the sharing strategy and the state of protocol.

The DSS can also inform the VM that an entity may be localized as described in section 6.1.8. Finally when the VM finds that a shared entity

that was not classified by a root, has no more local references it informs the DSS, so that the coordination assistant may be dismantled. In addition, this often triggers some action by the DSS according to the requirements of distributed garbage collection.

6.1.11 Bootstrapping

In a centralized system threads may be created in a common address space. Thus a newly born thread may be initialized having a number of references that are held by other existing threads. In a distributed system there is no general counterpart. A thread created in one machine cannot be initialized with references that are held only by threads in other processes.

Without a common address space we need some other mechanism to bootstrap threads and/or processes. One example of a bootstrapping service is naming services. But note that then the process/thread will need to know the naming service, ad infinitum. Ultimately bootstrapping mechanisms need to be able to extract references from one process in some form and then piggybacking on other communication services be able to inject the reference into another process. Such a reference extraction and injection service is provided in the DSS.

6.2 DSS Internals

6.2.1 Protocol Machine

The main responsibility of the DSS is to maintain consistency of shared entities. In other words, the DSS maintains the functional properties of shared entities. It also maintains non-functional entity properties, such as memory management and fault-detection.

Maintenance of these properties are achieved by protocols or distributed algorithms. The protocols are realized by coordinated message sending. This is done on a per entity basis and for each shared entity there are a number of different protocols that are actively working over that entity's coordination network. The DSS can be viewed as a massively concurrent protocol machine. Not only are there several protocols running for each shared entity, a typical DSS is at any one time engaged in supporting many shared entities.

Clearly, one of the main tasks of the DSS is messaging.

6.2.2 Identifiers and Sites

Shared entities in the DSS have unique identifiers. The identifiers are guaranteed to be unique over time and unforgeable. The identity scheme used

in the DSS is based on the triplet: *ProcessId*, *CreationNo*, *Large Random Number*. The process identifier uniquely identifies the process where the entity was first created, while the entity identifier identifies the entity within the process. The large random number prevents the forgery of references. More precisely the random number makes guessing the identity of an entity unlikely; exactly how unlikely, or how large the random number is a configurable parameter of the DSS.

Central in the DSS is the notion of remote processes. Conceptually there exist a link to each process that the DSS knows of. The DSS knows about other process indirectly. Coordinator assistants always know their coordinator(s), so the DSS will know the process(es) holding the coordinator(s). Coordinator assistants sometimes, but rarely, know other coordinator assistants within the same coordination set. And depending on protocol choice and other factors, coordinators know zero, one, some, or all of their assistants.

Inside the DSS the remote process is represented in a data structure, called a site. Sites serve two purposes, first they are used as identifiers for processes and second they are used for establishing physical communication channels using the OS-service module.

The DSS upholds the property of at most one site data structure per known process for a number of reasons. First, it is quite possible for a DSS to know a large number of sites/processes (without necessarily communicating with them). Second the size of a site data structure may be fairly large (especially the security-related fields). Finally, in much the same way that DSS may repeatedly import references to the same entity, a DSS may repeatedly import the same process id. In fact, repeated import of the same process id will be even more likely than repeated import of the same entity.

The site data structure is composed of two fields, the process id, and a virtual address. The virtual address is never used directly by the DSS; it lies within the domain of the OS-service module. The virtual address is passed to the OS-service module when and if the DSS determines the need for establishing a physical communication channel between two processes.

6.2.3 Message Contents

Conceptually messages are sent between coordinators and coordination assistants. In the implementation messages are sent between processes. Most messages therefore contain the identity of the entity associated with the coordination network. At the receiving end the DSS directs the message to the appropriate coordinator or coordination assistant.

Messages typically contain DSS-specific as well as VM-specific information. The VM-specific information is marshaled in one VM, and routed via

its DSS partner to some other DSS, and finally to the associated VM. Each of the components is responsible for marshaling or serializing its part in the same buffer.

The DSS recognized two cases when sending a process id. In one case, the invariants of the protocol or coordination network ensure that the recipient process already knows the process id. An example is when a coordinator sends to one of its coordination assistants. In this case the recipient already has an appropriate site data structure built, and in particular, already knows the virtual address associated with the process id. (Note that the process id must still be included in the message so that the receiving DSS can direct the message to the right coordination assistant). The other case, is when the sender is not sure; the recipient may or may not already know the identity of the process. In this case the virtual address must be included in the message also.

Marshaling and unmarshaling of the virtual address is done by OS-service component, so that, referring back to section 6.1.3 and figure 1, when a reference is passed between two processes, the message is partially constructed by the VM, partly by the DSS, and partly by the OS-service component.

6.2.4 Channels

We distinguish between logical channels and physical channels. From the viewpoint of DSS A there exists a logical channel between itself and DSS B if one the two following conditions holds: 1) there is exists a link in some coordination network between A and B or 2): B previously connected to A, indicating its need for a logical channel between A and B and has not yet announced that it has no further need for the channel. The logical channel between two processes is kept open until neither site has a need for it, by criteria no 1. There is a 1:1 mapping between logical channels and site data structures.

There is very little overhead associated with maintaining a logical channel by itself. A logical channel merely indicates that the DSS is ready to send and receive messages to the process it is connected to. Logical channels are divided into logical sub channels to cater for dealing with messages of different priority. This is used to minimize the impact of messages produced by background activities.

There are three important priorities, 1) control messages with high priority, 2) protocol messages with medium priority and 3) maintenance message with low priority. Control messages are used by the DSS to set up and take down physical channels between processes. They are used by one DSS in its negotiation with other DSSs. These are importance as one process may

be under extreme heavy load and another under light load. Maintenance messages deal with non-critical system aspects where delays do not block threads. For example, messages dealing with memory management are in this category.

Then there are physical channels. The DSS controls the opening and closing of such channels, but the actual connection resides in the OS-service module. The DSS respects the resource limitations that the OS-service module stipulates. For example, if the OS-service module makes use of TCP/IP on standard operating systems the number of connections that can be open at one time is limited. The DSS will then time-share the limited resources, opening and closing physical channels on its own initiative.

The DSS will maintain at most one physical channel for every logical channel that it has. All messages destined to the same process will thus use the same physical connection, and if the OS-service layer user TCP/IP the same connection.

6.2.5 Messaging

Messages are generated by coordination assistants and coordinators according to some protocol. Messages do not need to be, and in general are not, sent at once. Rather the messages are scheduled for delivery. Messages are queued in the appropriate virtual channel, and in due course will be delivered.

When messages are scheduled, they are placed in data structures called message containers. Most of the information in the message container is DSS specific information, having to do with the message type, recipient id(coordination assistant or coordinator), etc. The container may also contain virtual pointers (to entity structures). Note that nothing is marshaled or serialized when messages are queued in the DSS.

The careful distinction between scheduling a message on the one hand and marshaling and sending a message on the other is important for two reasons, granularity control and memory usage. Entities in message containers, i.e. in unsent messages, are represented as pointers to message mediators which in turn reference the entity in the VM directly. The indirection allows for relocation of the entity by the VM as a result of local garbage collection.

At the receiving end messages are also placed in message containers and scheduled for processing. Construction of protocols is greatly simplified by the fact that there are no provisions for the special case when messages are sent between coordination assistants and the coordinator when the both happen to reside in the same process. Almost all of the potential optimization of this can be done by shortcutting on the level of message containers. A message container being scheduled for sending will in the special case realize

that the message is destined to itself and place the container in the receive queue. This avoids the two most computationally expensive tasks, marshaling and communication.

6.2.6 Fault Detection

The DSS implements fault detection on a per entity basis. Thus, if an abstract entity becomes non-usable due to a network problem, or a process crash, this is reported to the VM so that it can take appropriate action. We say that the entity has failed.

A necessary condition for entity failure is that there is a problem in some with some link or some process in that entity's coordination network. However, this is not a sufficient condition; from the viewpoint of any one process it is quite possible that one, some or all other processes and/or links may be down without failure. It all depends not only which protocol (or sharing strategy) the entity is using to maintain consistency but also the state of the protocol.

It is an important criteria in designing the distributed algorithms that are the basis for all various sharing strategies and consistency protocols in particular - that the ability to detect entity failure is maximal. By maximal here we mean within the bounds of algorithmic complexity, network hops etc.

Note that fault detection on the level of processes (and indirectly links) is done by the OS-service component. The job of the DSS is to take this information and to map this to entity failure. The OS-service component reflects faults to the DSS abstractly - there are only two fault states (and one normal state). The fault states are permanently lost **perm** and temporary lost **temp**.

The monotonic fault state *perm*, represents a process that has crashed or failed, and will or can never be revived. No communication with a perm process is possible or can become possible. *temp* is a transient indicating that the process is currently unavailable, i.e. it is right now not possible to communicate with the process, but it might (and might not) eventually be possible to communicate. Note that the temp fault state is truly temporary; the same process may switch between the normal and the *temp* states any number of times. On the other hand, once *perm* it is always *perm*.

The process fault states will be mapped to zero, one, or more entity failures by the DSS. Entity failures may like process faults be classified as being *temp* or *perm*. The DSS will report any changes in an entity's failure status to the VM.

As is well known, permanent faults (*perm*) or crash failures can be very

hard or impossible to detect in asynchronous distributed systems. It is partly for this reason that detection of permanent process failures is outside of the DSS. We have placed it in the OS-service component. This is natural as process fault detection is often tied to the actual transport medium used (e.g. TCP). Also, as was previous described, the translation from virtual addresses to physical ones was also placed in the OS-service component. This is only natural as if processes are persistent or can change their IP-address this must be taken into account in fault detection. Finally, the OS-service component can be made open to application-specific instrumentation.

We also placed detection of temporary faults in the OS-service component, for many of the same reasons. However, most temporary fault detection is based on timeouts and round-trip time measurement. It is useful and efficient to let the DSS be responsible for heartbeats, as well as monitoring and measuring round-trip time at the instruction of the OS-service component. This can reduce the network traffic, for instance, if two processes are anyway communicating frequently due to protocol action round-trip messages are piggybacked on normal messages.

6.3 Sharing Strategy

In section 6.1 we described a framework for supporting the sharing of entities between processes. Within the bounds of the framework we described a space of possibilities for realizing sharing. We claim that for any particular entity type, even without violating network transparency, there is not one but many reasonable choices within the space of possible strategies. A reasonable choice is a strategy that is optimal for at least one pattern of use. The entity type by itself does not provide sufficient constraints to be able to choose one best strategy; best strategy depends on the application and the pattern of use.

Our view is that in order to deal with this, that the DSS should support all reasonable strategies. The space of possibilities becomes then a tuning space. The choice of location within this tuning space is what we call a sharing strategy. We also believe that the choice of sharing strategy should be made by the application developer, possibly with the aid of tools.

Note that this does mean that for every single shared entity that the pattern of use must be carefully analyzed so that an appropriate sharing strategy can be chosen. The fact that all parts of a program may be tuned for optimality does not mean that they need to be. Just as for centralized programming careful tuning might not be needed at all, or alternatively can be made late in the development cycle (e.g. after profiling). It is also reasonable that a DPS built using the DSS, chooses a default strategy per entity type. Only when needed is the default strategy overridden through

programmer intervention.

In our work on sharing strategies we believe that we have identified the reasonable choices within the space of possibilities. We have also worked on simplifying navigation within the resultant tuning space. While the tuning space is fairly large the complexity is greatly reduced as tuning can be done in three, more or less, orthogonal dimensions.

We believe that we have mapped the tuning space thoroughly and that the strategies that we present in this section are complete in the sense that all reasonable choices can be specified (within the bounds of transparency). Our DSS implementation is, currently, not complete. Some of the strategies that are presented here are not yet implemented. However, we have implemented, very many of them. Our implementation offers tuning capabilities that are richer than other systems by an order or magnitude. We also claim, that our implementation covers a sufficient number of strategies to be proof of concept. It is possible and practical to aim for complete tunability.

6.3.1 State of the Art

We will see that the gains in performance that can be made with complete tunability are arbitrarily large, i.e. on the algorithmic level. That tuning is useful is also supported by features that existing middleware offer. Taking Java remoting as an example, numerous books and articles address the issue of how to tune a remote object system. Furthermore, numerous versions of Java RMI([9], [19], [21]) exist with different implementations to improve performance in remoting. For instance, asynchronous RMI where the invoker need no longer wait for method completion at the remote site will greatly speed up some applications at the price of reduced transparency. (In our view there's nothing wrong with having asynchronous operations in a distributed programming system, just the opposite they're useful, but preferable would be to be able to perform asynchronous operations both within one process and between processes).

Tuning any DPS can essentially be done at two levels. First, the messaging system, including messaging and marshaling, can be targeted. This can even be done without endangering transparency. For instance, consider the openness of the serialize interface in Java. A class can implement a specialized serializing strategy that can be optimized both in respect to the byte representation and performance.

Second, the protocols can be targeted. Asynchronous RMI is one example. From our point of view, this does not really represent different tunings, but different language constructs - so different that they are mapped to different abstract entity types (synchronous stateful versus asynchronous stateful).

More generally, CORBA [28] offers the possibility to intercept remote method invocations, Microsoft's .Net [31] architecture exposes a similar intercepting interface. Both systems give the programmer an interface to work on the protocol level. In principle, it is then possible to program at the application level many of the same consistency protocols that we have in our DSS (in the case of CORBA the language must be the same at both ends). But this is no longer tuning, but rather reimplementing of DSS functionality without language independence and at lower efficiency (the interception is rather expensive).

6.3.2 The Three Dimensions of Sharing Strategy

We divide sharing strategy into three, more or less, orthogonal dimensions. To understand this we need to consider the coordination network and what we are trying to achieve with it.

For each entity the coordination set forms a small dynamic distributed system. The set is continually changing. Processes join as a result of passing references. Nodes leave as a result of references passing out of scope (garbage collection). Consistency and other important properties of shared entities are realized by various distributed algorithms working over this distributed system.

Many distributed algorithms require an arbitrator, leader, or coordinator. Members of the system will, when triggered by the action of the VM, need to consult the coordinator, i.e. all members of the coordination set need to be able to find the coordinator. The first dimension of sharing strategy is concerned with the characteristics of the coordinator. We call this the **coordination strategy**. Important considerations when choosing coordination strategy are provisions for coordinator mobility and/or redundancy.

The second dimension of sharing strategy is the **consistency strategy**. This is where the specific consistency protocol or consistency algorithm is chosen. Unlike coordination strategy this choice is dependent on abstract entity type. There is still often a choice, but the entity type limits the choice. For instance, we cannot choose one of the protocols geared to stateless entities when we are dealing with stateful entities.

The third dimension of sharing strategy is the **memory management strategy**. A coordination set is first created upon globalization. At this point the coordination set will consist of a coordinator(s) and two coordination assistants (to begin with one of the coordination assistants is in embryonic form, i.e. in a message). Thereafter the members of the coordination set will be continually changing with processes leaving and joining. The purpose of the memory management strategy is to recognize the situation when there

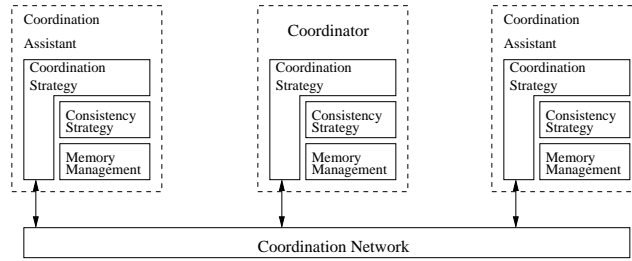


Figure 3: A coordination set of two coordination assistants and one coordinator. The figure depicts the relationship between the three subcomponents of coordination, the coordination strategy, the consistency protocol, and the memory management strategy.

is only one or no coordination assistants remaining. It is at this point the coordination network, the coordinator(s) may be removed.

6.3.3 Coordination Strategy

Central to the functionality of a coordination network is its arbitrating unit, the coordinator(s). The coordination strategy defines the number of coordinator in a coordination set and their mobility properties. Furthermore, it defines how the coordination assistants communicate with each other and the coordinator.

We recognize the following coordination strategies:

Stationary The stationary coordination strategy has one single coordinator, located at the process where the entity was first globalized. This is the simplest coordination strategy available and has a small memory footprint. However, its strong dependency on the creation process makes it vulnerable to failures. Even if it is known that the creation process is to be taken down, nothing can be done.

Migratory The migratory coordination strategy has one single coordinator with the possibility of migration between processes. Within the migratory coordination strategy two questions arise.

First, how should the coordinator assistants find their coordinator? Since the location of the coordinator is not static, the coordination assistants cannot rely on static information of the location of their coordinator. Instead a mechanism is necessary for enabling contact from coordination assistants to their coordinator despite eventual moves of

the coordinator. The spectrum of possible solutions ranges from simple forward chaining algorithms to solutions where the coordinator has full knowledge about all its assistants.

Second, what triggers migration? Migration can be triggered either externally from the EVM or internally from the DSS. External triggering allows for active control from the EVM, an example of use would be to move a coordinator from a process that is to be taken down. Internal triggering can migrate the coordinator away from heavily loaded processes to lower loaded ones (in the same coordination set).

Replicated Coordination is not realized by one single coordinator, but by a set of coordinators, running an internal consistency protocol of their own between them. The set of coordinators is not visible to the coordination assistants, from their point of view of the assistants the set looks like a single coordinator. Loss of the one and only coordinator in the two previous protocols fails the abstract entity that the coordination supports. Replication allows for loss of any number of coordinators as long as at least one is still alive. The advantage of resilience to faults is not for free. The internal consistency protocol that keeps the set of coordinators consistent implies extra messages and delays.

Our DSS implementation currently supports stationary and migratory coordination strategies.

6.3.4 Consistency Strategy

The consistency strategy is the choice of consistency protocol or algorithm. A consistency protocol exposes a set of protocol specific operations. For example, a channel (message-sending) protocol would expose just one operation, send. Other protocols expose multiple operations. It is the responsibility of the DSS (coordination assistant) to translate abstract entity operations into the corresponding protocol operations, valid for the consistency protocol it is using.

The protocols that we recognize are summarized here. They are currently available in our DSS implementation unless otherwise explicitly stated.

For entities of the synchronous stateful abstract entity type one of the following may be chosen:

Remote Exec The protocol assigns one of the coordinator assistants as representing the home of the protocol, commonly, the first coordinator assistant created. The protocol operation *execute* passes an abstract

operation to the home process, executes it there, and returns the result to the initiator of the operation. Operations performed at the home process can be performed locally. The protocol is a variant of the RPC or RMI protocol.

Migratory The protocol implements state as a token that is passed around in the coordination set. The coordination assistant that holds the token has sole access to the state of the abstract entity. A coordination assistant can attract the token using the protocol operation **access**. Migration of the token is controlled by the coordinator. The base protocol as described in [32], was refined for better fault-detection.

Read Write Invalidation A token based protocol with two kinds of tokens, read tokens and a write tokens. Coordination assistants can acquire either a read or a write token, using the operations *read* and *write*. Possession of a read token grants only read access to the state whereas the write token gives sole access to the state, thus allowing for both read and write. Before a write token can be granted, all read tokens must be invalidated. This protocol is well-suited for entities where the frequency of write operations is low compared to read operations. We recognize a number of different subvarieties of this protocol differing in the eagerness of update propagation. In our DSS implementation we have only one (eager) variety.

For entities of the asynchronous stateful and monotonic abstract entity types there is only one suitable protocol.

Channel This protocol is suitable for the asynchronous stateful abstract entity type. The channel protocol assigns a home coordination assistant similar to the Remote Exec protocol. The operation *send* is asynchronous and no confirmation of completeness is returned to the initiator of the operation.

Monotonic This protocol is used for logical (single-assignment) variables, futures, and lazy evaluation. The protocol is a generalization of the distributed variable (or unification) protocol first described in [12]. We will not describe the generalization in more detail, and limit ourselves to single-assignment variables. Such an entity may be seen as being both stateful, and constrained by the fact that the variable may be bound only once. Threads that need the value suspend (data-flow) until the value is available. The task of the coordinator is to make sure that the variable is bound only once, and to propagate the information to all processes that reference the variable. Clearly, this involves

less synchronization (for the reader) than true state. Also when the binding becomes available in some process the coordination assistant is clearly no longer needed and may be reclaimed immediately. Thus, the monotonic protocol seen as variation on stateful consistency protocol takes an advantage of the single-assignment constraint to optimize the protocol. Given the choice between modeling a shared entity as stateful or monotonic, monotonic is always preferable. The protocol is the only protocol available for the monotonic abstract entity type.

For entities of the stateless abstract entity type there are 3 different protocols available, most with two subvarieties. The 2 subvarieties are **named** and **unnamed**. The difference between the two varieties is that DSS does not attempt to support the at-most-one-copy-per-VM for unnamed stateless entities.

Immediate Replication Entities are replicated as eagerly as possible. There is no abstract operation associated with this strategy. There is no element of laziness at all. In section 7.1.5 we will see that the unnamed variety of immediate replication defines a strategy that makes exposure of the entity to the DSS unnecessary. In the named variety the entity needs to be exposed to the DSS. Coordination in this case is used to guarantee the at-most-one-reference-per-process property.

Lazy Replication This strategy is maximally lazy. The stateless entity is explicitly fetched when the VM needs it. The corresponding abstract operation is *fetch*. For named entities using this strategy there may or may not be a delay (due to latency) between the need for the entity and acquisition. It may be that when the entity is first imported there already exists a copy in which case this copy will be linked in. With unnamed entities this will never happen and the first access will always be delayed.

Eager Replication This strategy is intermediate in laziness compared to the first two. Here a request for the entity structure is immediately made upon import when it is discovered that no copy of the entity exists in the process. Clearly if one does exist then the copy is immediately linked in instead. This strategy is useful when the entity structure is large and there is a fair chance that a copy already exists but the latency of lazy replication is unacceptable. In many applications, code (e.g. classes, procedures) is of this category. This strategy is only sensible for named entities, as otherwise even if there is an existent copy at the importing site the DSS cannot recognize it, so nothing is gained at the price of two extra messages and some delay.

6.3.5 Memory Management Strategy

Coordination can be seen as a contract between the EVM and the DSS. The contract starts when an entity is referred from more than one processes, i.e. it is globalized. The contract is terminated when an entity is again referred to be a single process or no process at all. The role of the memory management strategy is to detect termination of this contract. This functionality is achieved by properly packaged distributed garbage collection algorithms. Note that memory management must take into account that there may be references in transit between processes.

There are variety of distributed garbage collection algorithms with different properties as regards speed of detection, resilience to failures, how many messages the algorithm produces when a reference are passed through the network, and the memory footprint of the algorithm instances.

The following distributed memory management algorithms are implemented in our DSS.

Reference Counting A simple distributed reference counting algorithm, proposed by (Lermen et al[20]), which is based on the regular reference counting technique, adapted to the distributed environment. The algorithm has a small memory footprint, but requires messages to be sent to the coordinator when new references are acquired. Also, it cannot handle loss of a process that holds a reference.

Time Lease By keeping an expiration time we achieve both failure transparency and can minimize communication. The price is usually that reclamation is slower. This algorithm, found in Java RMI [25], unlike the others, is not safe in the sense that live entities may under some conditions be reclaimed.

Fractional Weighted Reference Counting A fast reference counting algorithm with the property that references can be passed between processes without the need for any third party communication. The algorithm is an extension to the classic weighted reference counting algorithm[36], but overcomes the problem when running out of weight. The algorithm has the similar shortcomings when it comes to failure as ordinary reference counting. The algorithm is described in [8].

Reference Listing In this algorithm the coordinator will maintain a list of processes holding references. Several different algorithms is based on this technique ([?] [33]). The algorithm has a disadvantage in that the memory footprint grows linear to the number of referencing processes.

Also the message traffic can increase considerably, and this scheme introduces latency in reference passing.

Persistent The entity is defined as being persistent. Such an entity will never be reclaimed but avoids all the overhead (messages and memory) associated with distributed garbage collection. This is useful for entities that will be accessible over the whole lifetime of an application. distributed garbage collection algorithm for such an entity is just a waste of messages, memory, and processor cycles.

Unlike the other two strategies, more than one memory management may be chosen (except for persistent which cannot be combined). The most interesting is a combination of time lease with one of the reference counting schemes. Such a scheme can reclaim faster than with only time-lease (via the reference counting component) but is more robust than only reference counting.

An orthogonal component of a memory management strategy is to turn off localization as mentioned in 6.1.8.

6.3.6 Discussion

We claim that the model for sharing strategies as described caters for both expressivity and extendibility.

The model for sharing strategies is expressive in that by combining the three substrategies in various ways we cover a very wide range of sharing strategies.

New strategies in one dimension can be added and immediately used together with existing strategies in the other dimensions. This is illustrated with an example from our development work in section 7 where we describe how we extended our implementation with one version of the read/write invalidation protocol.

One might think that this separation of functionality in separate modules would imply a notable overhead in execution time. However, it seems from comparison (see section 9) with existing distributed systems, both LIMs and LDMs, that the expressive model does not impose any great extra cost. This is true even without any sophisticated compile time optimizations to collapse the three subcomponents in one component.

7 Extending and Coupling a Virtual Machine

The design for a DPS presented in this article consist out of two connected components (three if the small OS-service component is include), the distribution subsystem (DSS) and an extended virtual machine(EVM). The virtual machine is extended with new abilities and interfaces needed to be able to cooperate with the DSS. Note that references to VMs in preceding sections are actually references to EVMs.

The job of extending a VM may be divided into two tasks. A convenient way to view the two tasks, is that the first extends the VM and opens it up for both outside manipulation and for delegation. This can be reflected in a EVM API or interface. The second task then consists of mapping the two interfaces or APIs, that from the DSS and that from the EVM, together.

The first task we call **core extensions**. These extensions greatly affect core VM functionality. Realizing the extensions will most likely require expertise in the internals of a VM. Hence, this task is primary intended for an expert in the design and implementation of the target VM.

The second task is related to coupling the EVM to the DSS. Here concrete language entities are mapped to abstract entities. This requires good understanding of the DSS API and the semantics of the language that the VM supports, but does not require knowledge of VM internals.

The second task is the actual **coupling**. Having taken a VM and extended it to a EVM the coupling to the DSS now needs to be performed. We will describe coupling on a fairly abstract level. In practice there are a number of low-level implementation details that need to be worked out. These will depend on the language interface between the EVM implementation and the DSS implementation. (Our DSS implementation was developed in C++)

7.1 Core Extensions

7.1.1 Thread handling

An essential core extension of the EVM is that it must be able to conform to our thread operation model. Operations on a distributed entity might or might not be permitted to execute immediatly. A thread invoking an operation on a shared (guarded) entity must be prepared to handle the following return codes *skip*, *proceed* or *suspend*. The calling thread is thus under direct control of the DSS. The return codes are:

skip The DSS will perform the operation for the entity, the EVM should continue with the next instruction.

proceed The operation can be performed locally, i.e in the same way as if the entity was not guarded. The EVM should perform the operation immediately and continue.

suspend The operation can not be performed now, suspend until further notice, i.e until woken up by the DSS.

If the return code is *suspend* the DSS will eventually resume the calling thread with one of the following instructions:

skip The DSS has carried out the operation and it is now safe to continue. Consistency is guaranteed.

proceed The DSS has retrieved the state of the entity, thus it is possible to carry out the operation locally. The EVM should redo the original operation.

Note that the EVM must provide the DSS with handles to threads. The DSS needs upon suspension store these handles so that later it can pass them back to EVM to wake them up. The thread model of any system using the DSS must thus be able to suspend operations and later to redo them.

7.1.2 Guards

Another core extension is the guarding of shared entities. To adapt the VM to the language model described in section 6.1.2 an entity structure must be extended with a conceptual guard, whose purpose is to force the EVM to cooperate with the DSS when operations on guarded entities is attempted. An entity needs to become guarded during globalization and upon import. Note that during globalization previously local and unguarded entities become guarded.

The VM should also be able to do its part during localization, i.e. removing guards.

7.1.3 Operation Tokens and Callbacks

For all operations that might be suspended by the DSS the VM needs to be able to create operation tokens. The VM also needs to be able to accept operations tokens (together with operands) and perform the corresponding operation. By an operation here we mean a concrete language operation. Note that the operation token may, and usually is, given to a VM different from the one in which it was created; hence the name callback. An example that illustrates this need is standard remoting. On remote method invocation

the invoking thread is suspended and an operation token is passed to the DSS. The DSS transports it to the home process of the entity and passes it to the EVM of that process for execution.

There are two different cases. The callback operation may be atomic in which case the EVM can immediately return and inform the DSS of the result. If the callback operation cannot be done atomically the EVM will need to create a new thread to perform the operation. This thread unlike a normal thread is artificial in the sense that it is not visible from the application level. If we consider remoting this artificial thread is conceptually the same thread that originally invoked the thread by transparency.

The EVM will need to notify the DSS on completion. In addition the EVM may need to supply the DSS with a handle to the thread so that the DSS may terminate the thread for those languages that provide for this type of thread control (e.g. where threads may kill other threads).

7.1.4 Memory management

The automatic memory management routines of the VM must be extended to cooperate with memory management of the DSS. The EVM must respect a new kind of root, the DSS root. In addition the EVM should inform the DSS when non-root guarded entities are found to be garbage so that the DSS may perform its own memory reclamation.

A complication is that many VMs need to be able to relocate entity structures in memory. This complication will manifest itself when we consider coupling (we were working with a relocating VM). In addition we have previously described the usefulness of incomplete entity structures for lazy replication (see section 6.1.5). Note that while the guard prevents the entity structure from being used in normal execution, it does not necessarily do so during garbage collection. If the VM relocates entity structures during garbage collection then the garbage collection must be instrumented to be able to deal with incomplete entity structures.

An additional memory management feature that may be useful is for the DSS to inform the engine that it is now allowed to convert a previously complete entity structure to an incomplete one. An example is the invalidation of the currently held state under the action of migratory and invalidation consistency protocols. Note that this only allows the EVM to compactify the memory representation of the entity. There is no other effect as the guard will prevent the entity structure from actually being used even without this memory optimization.

7.1.5 Marshaling

By marshaling we mean controlled iterative serialization of language entities or more generally of a language graph. If we take a given language entity as the root then in general this entity will reference other entities, as so on. Following all references, transitively, we expose a language graph. Some entity types are always leaves, e.g. integers. Others are from this point of view non-leaf, e.g. a procedure in which calls to other procedures are embedded.

The result of marshaling is a linear (byte) sequence that may be sent between processes. At the receiving end the sequence may be used to construct a replica of the original entity or entities. The reverse of marshaling is unmarshaling, controlled iterative construction of language entities from a linear sequence produced by marshaling in another but functionally identical EVM.

There are three aspects of marshaling and unmarshaling. First, the base ability to serialize, secondly the provisions for control, and thirdly the provision for iterative marshaling/unmarshaling.

We begin with serialization. In principle the extended VM must be able to take a language graph and serialize it in its entirety.

We now consider control. When the marshaler, working its way through a language graph, reaches a new node it must, in principle, cooperate with the DSS. The EVM asks the DSS if the node (or entity) should be serialized as a **skeleton** or **complete**. If the DSS responds with *complete* the EVM completes serialization of the entity and will then proceed with serialization of other entities (children) referenced by the entity. If the DSS responds with *complete* only then the entire language graph of the original root entity will be serialized.

If the DSS, on the other hand, indicates that the entity is to be marshaled in skeletal form things are quite different. Firstly, the entity itself, may be serialized incompletely. When unmarshaled the receiving EVM will build an incomplete entity structure. Secondly, the marshaler will not serialize the children of the entity at all. From the viewpoint of the marshaler the entity is treated as if it were a leaf of the language graph.

The marshaler is also iterative. This is more important at the unmarshaling end than the marshaling end. We first consider the iterative nature of the marshaler in the context of incomplete or skeletal marshaling. At some point an entity A is marshaled in process P1 in skeletal form. An incomplete entity structure is then later built in process P2. This would be the case, for example, when the lazy replication protocol is chosen for a stateless abstract entity. Later on, the entity A is marshaled again, but this time the subgraph

rooted at A will also be marshaled. Clearly there may have been some information as to the structure of A that was included in the skeletal serialization that now need not be reserialized. As said, this is not so important. More important is that when process P2 receive the marshaled subgraph rooted at A, this needs to be amalgamated or merged with the skeletal A that is currently being held in P2.

There is one, and only one, kind of entity node that does not need to be controlled upon marshaling, i.e. the marshaler need not consult the DSS as the response is guaranteed to be *complete*. This may, nevertheless, be a fairly common case. The kind we are referring to are stateless nodes with no laziness provisioning (i.e. immediate) and unnamed. This allows the EVM to marshal large stateless structures in one shot (e.g. large chunks of code or large data structures). The price is that the at-most-one-copy-per-VM property is not upheld.

Aside from the stateless immediates mentioned above marshaling is actually a cooperative activity. The EVM-marshaler is given a bytearray to fill by the DSS. Whenever the EVM-marshaler consults with the DSS it reports on how much of the buffer space it has used since the last consultation. The DSS will usually at this point also need to serialize some DSS-specific data. Control will then pass back to the EVM-marshaler. The serialized format will interleave DSS- and EVM-specific information. Needless to say the DSS-specific information is invisible to the EVM-marshaler.

7.1.6 Annotation Service

To take full advantage of the DSS the EVM should allow for the annotation of entities. The annotation specifies the sharing strategies that are to be used when and if the entity becomes shared. As noted before, this can be seen as optional, if each entity type is given a default sharing strategy. In our model the annotation must be made when the entity is still local as it is, in general, very difficult, to provide for switching from one consistency algorithm to another under use. This means that the EVM must be able to associate an annotation with an entity, so that when and if the entity becomes globalized to supply the DSS with this information.

A less ambitious EVM might annotate by type exclusively which can be done during coupling.

7.2 Coupling

There are two important issues when an EVM is to be coupled to the DSS.

The first is how to establish the 1:1 correspondence between entities or entity structures in the EVM space and coordination assistants and similar structures in the DSS. In figure 2 the double-headed arrow represents the correspondence between an entity structure and a coordination assistant. An event in the DSS may require the DSS to communicate with the EVM and ask for actions to be performed on the corresponding entity structure. The DSS needs to be able to specify to the EVM which entity is referring to and will therefore use the upwards link. In addition the EVM may attempt to perform an operation on the guarded entity and need to communicate with the DSS. As only the coordination assistant knows what to do the EVM needs to be specify the appropriate coordination assistant. The EVM thus uses the downward link.

The second important coupling issue is to ensure that the EVM supplies correct information as to abstract entity types, abstract entity operations, and sharing strategies to the DSS upon need. Note that this includes the task of classifying concrete entity types and operations to abstract entity types and operations.

There are also some memory management aspects to coupling.

7.2.1 Cross Component References

In section 6.1.7 we introduced virtual pointers. These were the upward links in figure 2, pointers from the DSS into EVM space. However these virtual pointers are implemented, true pointers, tokens, entity ids, etc. the EVM must be able to take such a virtual pointer and find and work with the corresponding entity. Depending on the target language the virtual pointer may need to contain type information. There are four kinds of virtual pointers that can be classified by the DSS data structure type that they are located in:

Coordination Assistants Covered in section 6.1.7.

Message Container Covered in section 6.2.3.

Thread Container Contains a pointer to a thread

Named Entity These pointers are used for maintaining the at-most- one-copy-per-VM property for stateless entities.

The downward link of the double-headed arrow in figure 2 is a special case of the more general ability to given a virtual pointer determine the identity of the corresponding DSS structure if one exists. One possible implementation

of this would be a hash table in the DSS that hashes on virtual pointers. Such a table could have been implemented within the DSS.

We deliberately left this out of the DSS to allow for the optimization that the EVM might provide space for a DSS-pointer in the entity itself. Rather the DSS will export DSS-pointers to the EVM, which the EVM is expected to pass back to the DSS in the appropriate interface functions. For example, when entities are globalized the DSS creates a coordination assistant and passed a pointer to this assistant to the EVM. Later when operations are performed on this entity this pointer should be passed back to the DSS. If storing DSS-pointers in entities entity is not possible then some small additional code for hashing will need to be added when the two components, the EVM and the DSS are coupled together.

7.2.2 Mapping the Concrete to the Abstract

When entities are globalized during marshaling the EVM must supply the abstract entity type and entity annotation (sharing strategy) to the DSS so that the DSS can initialize the appropriate coordination network and structures.

When operations are performed on guarded entities the EVM must supply the abstract operation type (as well as the concrete operation token).

7.2.3 Memory Management

In order to allow for memory relocation in the EVM the DSS supplies an interface that allows the EVM to inform the DSS when it updates a virtual pointer.

7.3 Some Notes on Our Implementation

In the coupling that we made we coupled a EVM written in C++ with our DSS implementation, also written in C++. This was done in a considerably more object-oriented manner than described here. The virtual pointers described in section 7.2.1 where all represented as objects, their interface classes where defined in the DSS. This cater for simple interaction between the DSS and the EVM, as communication between a coordinator assitant and its entity counterpart was directed over a mediating object. Furthermore, the translation between the concrete and abstract operations where naturally located in the mediating objects. This makes the code for each distributed entity type self-contained.

A question that we have not yet considered is the price of distribution on local operations. After all, most operations will probably still be local and if this is appreciably slowed then it probably does not matter how good the distribution support is.

In our observations the only feature that had any impact on local performance was the price of guards. In our implementation guards were implemented by a single bit flag in entity structures. Local entities could then be transformed into guarded entities by setting the flag. This very simple mechanism has the drawback that every operation on an entity that is not restricted by type to be guaranteed local needs to perform a flag test. This slows ordinary centralized execution. However the slowdown was limited to 0.5% in our DSS-EVM system. In systems with JIT, this may be larger. Other schemes are possible that trade increase globalization overhead but lowers or eliminates centralized execution slowdown.

8 Extendibility of the DSS

A prime example of the modular structure of the DSS is the simplicity of adding a consistency protocol to the DSS. Taking the design of an algorithm and transforming it to a DSS protocol can be realized without the tedious work of learning how other internal components behave or what side effects to consider.

The DSS is built using C++ so an algorithm designer must be familiar or at least able to formalize his model using standard C++ primitives as well as using our primitives to implement the algorithm part. The protocols will thus be defined as automatas which react on invoked methods and uses a well-defined interface. A protocol is divided into node (process) instances and an all-embracing arbitration facility. The node instances are parts of the Coordinator Assistants (CA) and they must all implement the interface *CAProtocol*. The arbitration is performed by the Coordinator, implementing the interface *CoordinatorProtocol*. The use of interfaces is a matter of course in the object oriented domain and natural in the modular design of the DSS.

As the protocol is divided into Coordinator Assistants, being protocol executers - one for each node in the coordination set, the designer must design his protocols in terms of what each CA knows of the rest of the protocol. The actual primitives to communicate and represent a protocol node is very simple, recollect that there exists at most one copy of an entity on a specific process *REF*, hence for representation of another CA it is enough to store that as a process *REF*, the communication primitives will then automatically find the correct CA.

The message component is a simple sending- or receiving message container which allows us to send/receive three things: operation arguments or entity structures, an integer or a process representation. The container models queue behavior thus the first thing added to it is also the first that will come out. The conceptual queue model was a design decision taken due to the possibility for a message to be created in several steps by different parts of the DSS.

The interfaces for a protocol are divided in three parts, one is the primitives exposed to the protocols, one from the coordinator and one from the assistant. A protocol need not to use/implement all interfaces if they have no meaning for the protocol, instead they should be regarded as a toolbox in the DSS environment. The DSS exposed interfaces are:

- Interface for receiving a protocol callback operation. Used when performing abstract entity operations.
- Messaging interfaces to create and send a message to a coordinator, found by the CLS, or to a CA on another process.
- Resource management methods in case a protocol wants to protect a remote process-object from removal. A typical case is when the protocol is representing another CA as a process id.

The other two parts are the specific interfaces implemented by the coordinator part and/or the assistant part. The interfaces those protocol parts may implement are listed below:

- Interface for receiving a protocol operation (CA only). This is necessary for any protocol and exposed to the EVM.
- Messaging interface for receiving of a message from the coordination set. The interface also informs about which the process id of the sender thus that information need not to be included in the message. This interface is the same for both the coordinator and the assistant parts.
- Means to prevent the CA from being removed during memory management in case we need to stay resident. An example is if we have entity-critical data, which would be lost if the particular CA is removed before rescuing the data to another location. The coordinator is automatically as long as there are references to the entity.
- A method informing that a remote process has changed its fault state [REF upwards], the protocols can then deduce if it is affected by this or not.

- Both the coordinator and the assistant can implement serializing/deserializing methods to cope with migration.

The above lists describes the toolbox used in the interaction between a protocol and the DSS. They are, as mentioned, not needed for every protocol but offers the tools to describe a protocol in terms of messages and states, as well as covering other protocol depending operations of the DSS which might affect a protocol.

A basic protocol for synchronous communication is realized using these primitives in around 80 lines of code. The complex logical variable is constructed in around 250 lines of code giving an estimate of the complexity when formalizing an algorithm. Naturally a designer also has to hook in into the entity module, specifically into globalization functions where examination of the chosen protocol is done as well as deserializing-of-a-reference functions. These code changes are both minor and trivial though. Below is an example of the code needed to construct the stationary state protocol. First is the methods implemented by the coordinator. Note that it only implements those methods needed by this protocol, also note the *coord*-reference which is the compound object for all coordinator modules, offering interfaces between the modules and also to the rest of the DSS. This setup is similar for CA:s.

```

1   void ProtSC_Coord::msgReceived(MsgContainer *msg, Site* sender){
2       int ao      = msg->getIntVal();
3       int call_id = msg->getIntVal();
4       MMin_Interface* in = msg->getMMVal();
5
6       // Perform the operation on the local entity structure
7       MMax_Interface* ans;
8       ans = ((ProtSC_C*) coord->getLocalCA()->getProt()->statOP(ao,in);
9
10      SendMsgContainer *msgC = coord->createCAmsg(M_CA_PROTOCOL);
11      msgC->putIntVal(call_id);
12      msgC->putMMVal(ans);
13      coordinator->sendCA(msgC, sender);
14  }
15
16  void ProtSC_Coord::sendMigrateInfo(SendMsgContainer* msg){
17      MMax_Interface* load = coord->m_getLocalCA()->extractState();
18      ((ProtSC_C*) coord->getLocalCA()->getProt()->statChange(REMOTE);
19      msg->putMMVal(load);
20  }

```

```

21
22 void ProtSC_Coord::instantiateMigrateInfo(MsgContainer* msg){
23     MMin_Interface *load = msg->getMMVal();
24     coord->m_getLocalCA()->installState(load);
25     ((ProtSC_C*) coord->getLocalCA()->getProt())->m_statChange(LOCAL);
26 }

```

The above code describes what can happen to a stationary synchronous coordinator. It can receive an operations, and also has to act during migration, when conceptually moving the local proxy to the new place. Below is the code for the CA. It shows how an operation is handled and what interfaces are implemented as well as the two specific methods for this protocol (lines three and seven, `statOP` and `statChange`).

```

27 ProtSC_CA::ProtSC_CA(int st):ProtocolCA(PN_SYNC_CHANEL){state = st; }
28
29 MMin_Interface* ProtSC_CA::m_statOP(int absop, MMin_Interface* b){
30     return ca->doe(absop,b,NULL);
31 }
32
33 void ProtSC_CA::statChange(int s){ state = s; }
34
35 void ProtSC_CA::msgReceived(MsgContainer *msg, DSite*){
36     int th = msg->getIntVal();
37     MMin_Interface* load = msg->getMMVal();
38     DssThreadId *id = reinterpret_cast<DssThreadId*>(th);
39     ca->accessMediator()->resumeFunctionalThread(id,load);
40 }
41
42 OpRetVal ProtSC_CA::protocolOperation(ProtOp op, MMin_Interface* load,
43                                     AbsOp aop, DssThreadId* th_id){
44     if (state == LOCAL) return DSS_PROCEED;
45     switch(op){
46     case SEND:{
47         SendMsgContainer *msgC = ca->m_createCoordMsg(M_COORD_PROTOCOL);
48         msgC->putIntVal(aop);
49         msgC->putIntVal(reinterpret_cast<int>(th_id));
50         msgC->putMMVal(load);
51         ca->sendCoord(msgC);
52         return (DSS_SUSPEND);
53     }

```

```
54     }
55     return DSS_INTERNAL_ERROR_NO_OP;
56 }
57
58 bool ProtSC_CA::isRoot(){ return (state == LOCAL) };
```

9 Performance comparison

One of the main features of the DSS is its language independence, given a programming languages which meet our stated requirements(see section 2) the DSS can be incorporated into it, extending the language into a distributed programming system.

Given that the integrator, or VM designers, provides an optimal solution for the coupling to the DSS, the performance impact with respect to regular operations should be minimal. Most programming systems of today offer some means to achieve distribution though, and it is thus important to detect whether the incorporation of the DSS will degrade the distributed operations. Note here that to be able to do fair performance estimates the DSS must be natively coupled to the VM, or at least on a low implementation level. The purpose of these tests is to show that the DSS is efficiently performing previously used distributed operations, thus including the DSS instead of another language dependant middleware might have a small performance penalty but the programmer is offered the possibility to distribution enable a greater variety of entity types as well as possibly optimize other constructs. The tests will also show that the DSS only costs in direct distributed operations, independantly of the size of the argument for the operation.

As previously deduced the popular model of distribution has been to offer remote procedure calls, or remote method invocation, as the only distribution primitives. To give a fairly good estimate of how efficient the DSS model is we have used simple test programs, performing remote object invocations, to benchmark the DSS in comparison to other middlewares. These estimates are presented here as a proof-of-concept that despite some inevitable overhead of using the DSS, being language independent and general in design, it is still a viable option.

The systems targeted in these performance evaluations are: Mozart [4], Oz (based on the Mozart implementation) coupled with the DSS, Java using RMI [25] Java using CORBA [28], Erlang [2] and .Net Remoting [31]. Thus the results from this test show how well the DSS performs with respect to the system it is derived from.

The test program used was a small client-server implementation using

the distribution primitives offered by each evaluated system. The server side of the test program creates a data containing object and a distributed object containing the former. The client side of the program starts with establishing a connection to the server, then it invokes the remote object method several times to trigger any upstart cost in communication and runtime optimizations for those systems supporting it, as well as stabilization of the server. After this is completed the system total time in milliseconds is read and the remote method is then invoked ten thousand times (10000). As these remote method invocations are synchronous and thus sequential, the total time may then be read out directly after the last invocation, as it is assured that every call has been completed. The remote object invocation returns the data containing object as the result of the invocation.

The test program was executed with two different sized data structures in the transmitted object. In one case the object contained thirty integers, referred to as the *medium* object, and in the other only one integer, referred to as the *small* object. Thirty integers is a small enough data set to not exaggerate the marshalling part of the program. The object implemented a serializable interface on those systems requiring that. The decision to have a different sized data sets thus showing how a programmer should handle data in objects for the respective systems. The test programs files are available at <http://dss.sics.se/files/test.tar.gz> in a gzipped tar archive.

The test program measured the total time to execute a remote execution session. The time thus contains a couple of different components: How the programming systems handle I/O, i.e. how often does it check if something is to be sent/received. How effective the messaging service is. How well coupled the middleware is to the rest of the system, i.e. is there a big overhead of doing distributed operations. How well is marshaling implemented, is there a noticeable difference when sending small objects compared to larger, which will be shown by the different data sizes. The programming system also performs garbage collections in different ways, this might also be included in the tests. What the test thus shows is the overall execution cost in terms of time, when doing distributed object invocations. The different systems will show how well they handle this task in overall, not how costly each component is.

The test programs, both the client and the server part, were executed on an Intel Pentium 4 machine, 2400Mhz, 533FSB, using the i845GE chip set with 512 MB of DDR SDRAM. The machine ran RedHat Linux release 7.3, using kernel 2.4.20-pre10. The compiler used to compile the systems, when needed, was GCC version 2.96, included in RedHat Linux release 7.3. The operating system was not under any additional load except core system processes. For the .Net-Remoting tests Windows XP professional SP1 was

used, on the same machine. The different versions of Java and the other system specifications are listed below:

- SUN Java2. The Standard edition SDK, version 1.4.1_01 for Linux, in a pre-compiled rpm package downloadable from suns' java homepage [24]. This implementation will be referred to as Sun in this chapter.
- IBM:s Java2. The JDK for Linux 32-bit xSeries (Intel Compat.) version 1.4.0, available for download in a pre-compiled rpm package, from the IBM homepage [14]. This java implementation will be referred to as IBM for the rest of this chapter.
- Mozart developers version 1.3.0 [5]. Downloaded from CVS repository 2002-06-20. Compiled from source code using standard comilation options.
- Mozart developers version 1.3.0 as the Oz base with the DSS version 1.0 fully coupled. The complete source is not yet publicly available.
- Erlang version OTP R9B-0 [29]. Compiled from source code using standard compiler options.
- .Net Remoting Release version of .Net framework [22].

The system-specific versions of the test program were compiled using the standard compilation optimizations, such as an "-O" optimization flag.

The choice of two versions of Java was to give an estimate of the differences in distribution behavior between different implementations. The intention of including both Java RMI and Java CORBA was to show differences between LDMs versus the general DSS as well as LIMs versus the general DSS. The CORBA facilities used, such as the ORB and the idl-to-java generator, were those included in each Java distribution, with the implementaion of the test program following the POA model. The Oz/DSS and Mozart realized remote object methods using ports, Erlang using built-in rpc primitives.

The test program sessions were executed 20 times and an average was calculated. The results are presented below, they are normalized against the Oz/DSS times for the one integer test and gives an estime of the order of magnitude in difference between distributed operations on different platforms, using different types of middleware:

From these results there are a couple of interesting, and surprising, conclusions to make; The notable difference between CORBA provided from by IBM and CORBA provided by SUN, in Table 1 is unexpectably large

Language dependant middlewares (RMI) versus General DSS.

	Mozart	Oz/DSS	Erlang	Sun RMI	IBM RMI	.Net Remoting
<i>small</i>	0.86	1	1	4.30	3.17	5.94
<i>medium</i>	0.92	1.06	1.06	5.38	4.18	7.05

Table 1: Test times for each tested system normalized against the time for Oz/DSS. The test compares different language dependent middlewares against the integrated general

Language independant middlewares (CORBA) versus General DSS.

	Oz/DSS RMI	Sun CORBA	IBM CORBA
<i>small</i>	1	6.58	1.68
<i>medium</i>	1.06	6.77	1.84

Table 2: Test times for each tested system, doing remote object invocations, normalized against the time for Oz/DSS. The test compares language independent systems versus integrated general.

but the interesting thing is that RMI on IBM is worse (performance-wise) than CORBA on IBM (Table 1 vs Table 2), i.e. the language independent middleware outperforms the language dependent, something unexpected.

One thing we can conclude from Table 1 is that languages designed for distribution also perform simple remote object operations better than languages coupled with middleware designed to do only this. The assumption when creating the DSS was that when efficiently coupled to the EVM, the DSS would be almost on par with a specialized language dependant middleware, or within 10 to 20 percent. Our Oz/DSS implementation is slower than the original Mozart implementation, a fact we contribute the total language independence and absence of optimizations possible due to generality. The difference is not that big, the benefits with the new middleware is justified against a penalty within 15 percent, given the potential of including more specialized or complex algorithms.

All systems seems to handle larger data structures fairly well except for RMI on either java implementation and also .Net Remoting. From the figures in both tables [Table 1 and Table 2] we can conclude that the marshaling of data structures, which are replicates, has a minor impact on the total execution time. The DSS is within the same range as the Mozart system thus marshalling with the DSS is entirely language dependant.

10 Conclusion

We have presented a novel architecture for a language-independent middleware component. This component, the distributed subsystem DSS, can be coupled to virtually all high-level programming languages to create powerful distributed programming systems. These distributed programming systems can then offer the programmer an extremely simple and powerful distributed programming model.

The key to simplicity for the programmer is transparency, the centralized programming constructs are maximally extended to distribution. All the centralized language entities, objects, classes, procedures, futures, etc. can be shared between processes with no language semantic changes. This enables the programmer to develop a wide range of distributed applications without using a single additional concept as compared to the centralized programming. They behave the same way in all distributed settings as they would in a concurrent centralized setting (modulo timings and failure).

The key to the power of the distributed programming model is completeness as regards control and tunability. We have clearly shown this for some of the most important non-functional properties of programming, i.e. performance and memory usage. The middleware contains all known and in this context useful algorithms and mechanisms for achieving transparency. Optimality as regards the aforementioned non-functional properties can be achieved for all applications and patterns of use. The recommended programming API here is also maximally simple and expressive. In our architecture we present a three-dimensional model for tuning annotations on language entities. Entities may be tuned individually for optimality. Tuning does not impact transparency whatsoever, and may be seen as an option that might or might not need to be taken.

The core VM extensions necessary to make use of the DSS are carefully described. These need to be done by the skilled virtual machine developer. One aspect of the model is that these extensions are fairly few. We also provide a model and guidelines for coupling VMs to the DSS.

The architecture of the DSS promotes extendibility. Appropriate distributed algorithms can appropriately packaged be added to a DSS. We illustrated with the addition of one form of consistent object replication.

As proof of concept we also have an implementation of a DSS that has most of the functionality that is described in this paper. To complete that which is lacking is merely a matter of packaging the appropriate distributed algorithms, along the extendibility guidelines.

In the evaluation section we measure the performance of our DSS coupled to one particular VM. The results indicate that we are at worst reasonably

close in performance (within 10%) and often better than the most efficient existing language-dependent middleware today.

Our conclusion is that the DSS approach combines most of the conceptual benefits of both language-dependent and language-independent middleware. Distributed programming systems built using this approach can easily be constructed. They are then just as easy to use as language-dependent systems; they have a programming model that differs minimally from the centralized programming model.

Furthermore such DPS are very close in efficiency to what can be achieved using a language-dependent approach. At the same time, as the DSS which contains most of the necessary instrumentation for distribution is language-independent, the approach makes for wide applicability.

In practice we expect further benefits. The DSS architecture promotes extendibility. The internal features of the DSS architecture demonstrate separation of concerns; the architecture simplifies developing the middleware component itself, even if it were to be used internally within some language-dependent middleware.

11 Future Work

In our DSS implementation we plan to add the in the implementation missing functionality as described in this paper.

In our work we have not yet addressed security issues. We expect that security issues, just like with performance and memory management issues we will find a space of possibilities. We would then hope to map this space to locate the reasonable choices and to develop appropriate sharing strategies to deal with this issue. There is obviously a clear need to tune for security as the need for stringent security varies between applications and security (e.g. encryption) costs.

In our work we have only partly addressed fault-tolerance. In general, it is unclear to what extent algorithms for fault-tolerance should be implemented on the level of the DSS and what extent on the level of the VM. Fault-tolerance might be achieved through an additional layer between the distributed programming system and the application making use of the fault-detection mechanisms of the DSS. It seemed reasonable to us to provide for redundant coordinators as unlike the entities themselves coordinators and coordination processes are mostly hidden from the programming level. We plan to investigate this further.

Finally, we need to point out that our architecture leverages via the goal of transparency the semantics of existing centralized programming systems.

There may be, and probably are, all kinds of useful distributed programming constructs that cannot be obtained in this way; there are a number of semantic variations that do not exist in any centralized programming language. They are only of interest in distributed systems because they trade consistency for performance (latency). As an example, consider all the various ways in which objects may be replicated with weaker or ad-hoc (i.e. approximate) consistency models.

It is quite possible that successful distributed programming systems of the future will offer new constructs, constructs without centralized programming counterparts. Transparency will probably still be of interest. It is still useful to be able to develop, test, and debug programs on a single concurrent system. However in this case one would expect the development to proceed backwards, i.e. that the construct after having being proved in a distributed system is reflected back into a centralized system. What we have done is to take the proven centralized programming constructs and reflect them into a distributed setting.

In our architecture we were careful to place the functionality that translates between virtual addresses to physical addresses outside of the DSS. This puts all kinds of interesting services completely outside the DSS. Examples are process mobility and naming services. This is more a reflection of the current status of our work than any foundational design decision. In ongoing peer-to-peer projects we are investing these kinds of services. Possibly such services also belong in the DSS. If so, we hope to be able to provide them.

References

- [1] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, 1998.
- [2] Joe Armstrong, Robert Viriding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
- [3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.
- [4] Mozart Consortium. <http://www.mozart-oz.org>, December 2002.
- [5] Mozart Consortium. Mozart developers version 1.3.0, December 2002. Available for download at <http://www.mozart-oz.org/download/>.

- [6] Robin Cover. The xml cover pages. WWW page, 2002. <http://www.oasis-open.org/cover/xml.html>.
- [7] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [8] Per Brand Erik Klintskog, Anna Neiderud and Seif Haridi. Fractional weighted reference counting. In *LNCS 2150*, August 2001.
- [9] K. E. Kerry Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072, 1999.
- [10] Mark Grand. *Java Language Reference*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, January 1997.
- [11] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference. Volume 2, The MPI-2 Extensions*. Scientific and Engineering Computation. MIT Press, Cambridge, Massachusetts, USA, 1998.
- [12] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, 1999.
- [13] Haskell. Haskell language resources, December 2002. <http://www.haskell.org>.
- [14] IBM. Java developers kit version 1.4.0, December 2002. Available for download at <http://www-106.ibm.com/developerworks/java/jdk/>.
- [15] INRIA. The caml language, December 2002. The Caml language resources. <http://caml.inria.fr>.
- [16] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.
- [17] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, 1988.
- [18] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

- [19] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java remote method invocation (RMI). pages 19–36.
- [20] C.-W. Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, pages 343–350, Cambridge, MA, August 1986. ACM Press.
- [21] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Cerial J. H. Jacobs, and Rutger F. H. Hofman. Efficient java RMI for parallel programming. *Programming Languages and Systems*, 23(6):747–775, 2001.
- [22] Microsoft. Microsoft .net sdk with service pack 2, December 2002. .NET Remoting Release version available through Microsoft Developers Network.
- [23] S. Microsystems. Java remote method invocation specification, 1998.
- [24] Sun Microsystems. Java2 standard edition version 1.4.1, December 2002. Available for download at <http://java.sun.com/j2se/1.4.1/>.
- [25] Sun Microsystems. The remote method invocation specification, December 2002. Available from <http://java.sun.com>.
- [26] The University of Melbourne. The mercury project, December 2002. The Mercury Project. <http://www.cs.mu.oz.au/research/mercury>.
- [27] OMG., December 2002. The Object Management Group.
- [28] OMG., December 2002. The CORBA specifications, <http://www.omg.org>.
- [29] Erlang OTP. Erlang otp version r9b-0 source code, December 2002. Available for download at <http://www.erlang.org/download.html>.
- [30] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), September 1995.
- [31] Ingo Rammer. *Advanced .NET Remoting*. APress, april 2002.

- [32] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.
- [33] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. *Rapports de Recherche 1799*, Institut National de la Recherche en Informatique et Automatique, November 1992. Also available as Broadcast Technical Report 1.
- [34] Recursion Software., December 2002. Recursion Software Voyager ORB. <http://www.recursionsw.com/products/voyager/voyager.asp>.
- [35] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, MIT, December 1975.
- [36] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE Parallel Architectures and Languages Europe*, June 1987.