The background of the page features a large, faint, golden seal of the University of Bologna. The seal is circular and contains a central figure of a seated man, likely a scholar or saint, surrounded by various symbols and text. The outer ring of the seal contains the Latin text "UNIVERSITAS BOLOGNENSIS" and "S. PETRUS UBIQUE PATER".

Design and Evaluation of a Wide-Area Distributed Shared Memory Middleware

Michele Mazzucco Graham Morgan Fabio Panzieri

Technical Report UBLCS-2007-19

July 2007

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory ABSTRACTS.

Recent Titles from the UBLCS Technical Report Series

- 2006-26 *FirmNet: The Scope of Firms and the Allocation of Task in a Knowledge-Based Economy*, Mollona, E., Marcozzi, A. November 2006.
- 2006-27 *Behavioral Coalition Structure Generation*, Rossi, G., November 2006.
- 2006-28 *On the Solution of Cooperative Games*, Rossi, G., December 2006.
- 2006-29 *Motifs in Evolving Cooperative Networks Look Like Protein Structure Networks*, Hales, D., Arteconi, S., December 2006.
- 2007-01 *Extending the Choquet Integral*, Rossi, G., January 2007.
- 2007-02 *Towards Cooperative, Self-Organised Replica Management*, Hales, D., Marcozzi, A., Cortese, G., February 2007.
- 2007-03 *A Model and an Algebra for Semi-Structured and Full-Text Queries (PhD Thesis)*, Buratti, G., March 2007.
- 2007-04 *Data and Behavioral Contracts for Web Services (PhD Thesis)*, Carpineti, S., March 2007.
- 2007-05 *Pattern-Based Segmentation of Digital Documents: Model and Implementation (PhD Thesis)*, Di Iorio, A., March 2007.
- 2007-06 *A Communication Infrastructure to Support Knowledge Level Agents on the Web (PhD Thesis)*, Guidi, D., March 2007.
- 2007-07 *Formalizing Languages for Service Oriented Computing (PhD Thesis)*, Guidi, C., March 2007.
- 2007-08 *Secure Gossiping Techniques and Components (PhD Thesis)*, Jesi, G., March 2007.
- 2007-09 *Rich Media Content Adaptation in E-Learning Systems (PhD Thesis)*, Mirri, S., March 2007.
- 2007-10 *User Interaction Widgets for Interactive Theorem Proving (PhD Thesis)*, Zacchiroli, S., March 2007.
- 2007-11 *An Ontology-based Approach to Define and Manage B2B Interoperability (PhD Thesis)*, Gessa, N., March 2007.
- 2007-12 *Decidable and Computational Properties of Cellular Automata (PhD Thesis)*, Di Lena, P., March 2007.
- 2007-13 *Patterns for Descriptive Documents: a Formal Analysis*, Dattolo, A., Di Iorio, A., Duca, S., Feliziani, A. A., Vitali, F., April 2007.
- 2007-14 *BPM + DM = BPDM*, Magnani, M., Montesi, D., May 2007.
- 2007-15 *A Study on Company Name Matching for Database Integration*, Magnani, M., Montesi, D., May 2007.
- 2007-16 *Fault Tolerance for Large Scale Protein 3D Reconstruction from Contact Maps*, Vassura, M., Margara, L., di Lena, P., Medri, F., Fariselli, P., Casadio, R., May 2007.
- 2007-17 *Computing the Cost of BPMN Diagrams*, Magnani, M., Montesi, D., June 2007.
- 2007-18 *Expressing Priorities, External Probabilities and Time in Process Algebra via Mixed Open/Closed Systems*, Bravetti, M., June 2007.

Design and Evaluation of a Wide–Area Distributed Shared Memory Middleware

Michele Mazzucco¹

Graham Morgan¹

Fabio Panzieri²

Technical Report UBLCS-2007-19

July 2007

Abstract

This paper describes the design, implementation and experimental evaluation of an object-based Distributed Shared Memory middleware system. That system is unique among the existing DSMs as it structures the entire distributed system into small, completely independent subsystems; it enables the interaction among nodes belonging to the same subsystem, only, and allows processes to migrate dynamically between subsystems. The resulting implementation allows one to construct event-based distributed systems using a simple programming model; thus, applications can be deployed in any environment as neither hardware or software assumptions are made, nor reliable channels are required.

1. School of Computing Science, The University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU (UK)
2. Dept. of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy)

1 Introduction

Distributed Shared Memory (DSM) systems provide programmers with the abstraction of shared memory on top of message passing hardware. This model offers a low cost solution to the provision of shared memory computing in a distributed system, as it can be constructed out of off-the-shelf hardware and operating system components. In addition, it is easy both to build and to program as, in essence, all that the DSM runtime system has to do is to intercept user access requests to remote memory addresses, transparently, and translate those requests into appropriate messages. The application programmer is thus given the illusion of a large global address space, eliminating the tedious and error prone task of explicitly moving data among the machines that form the distributed system.

Both hardware and software implementations of DSM systems have been developed; in this paper, we describe the design, implementation, and preliminary experimental evaluation of a software-only DSM system.

The principal contribution of this paper consists of the DSM protocol we have developed in order to support effectively the construction of event-based application systems which can be distributed over a large geographical scale. This class of applications includes multiplayer online games, stock exchange applications, Internet auctions, and news distribution channels, for instance. In addition, our architecture can use non-reliable communication support (*e.g.* the Internet), where packets may be lost, or experience unpredictable delays. Finally, the DSM system we describe in this paper makes no assumptions on the hardware and software architectures required to support it, and can be deployed in a set of heterogeneous machines.

The paper is structured as follows. Section 2 introduces the problem we address and describes the core idea of the DSM architecture we propose. Section 3 briefly describes a prototype implementation of the architecture we have developed. Section 4 discusses an experimental evaluation of our prototype implementation. Section 5 examines relevant related work. Finally, Section 6 proposes some concluding remarks.

2 Background

In this section we first outline our motivation, in order to identify the context and the focus of our work. We then provide an overview of the challenges associated to the development of DSM systems in general, referring to well known work in this area. We then present a discussion that considers the most appropriate approach for DSM development given our motivation.

2.1 Motivation

The development of middleware architectures have provided developers with enabling technologies that ease the implementation of large-scale distributed applications deployed in heterogeneous environments. Such middleware identify the remote procedure call (RPC), sometimes called remote method invocation (RMI) in distributed object-oriented technologies, as the mechanism within which transparency of distribution is achieved. This has the result of making the interface, and associated implementation, the unit of distribution across a middleware platform. For clarity, we shall consider objects as the unit of distribution from now on as this is by far the most popular approach supported in middleware.

A developer competent in the implementation and deployment of applications based on distributed objects must consider programming complications not commonly found in non-distributed application development. For example, a developer may be required to use additional services to manage non-local object invocations (*e.g.*, security, concurrency control, reliability, location and discovery).

DSM systems attempt to provide a higher level of abstraction to the developer than that found in middleware where developers knowingly incorporate RPCs into their applications. In such systems, transparency of distribution is afforded via the access of shared memory: irrelevant of where a client access occurs, or where the shared resource is located, the developer views such

an access as simply a local access of a local resource within the regular programming style of the implementation language being used. As such, the appropriate utilization of required services (e.g., location and discovery) is handled by the DSM system (ideally, the developer is unaware of remote access).

Developing a DSM system for use with object-oriented middleware to provide large scale distributed application deployment in heterogeneous environments would be beneficial: developers would program their applications without hindrance incurred from using the services required for distributed object implementation. This will abstract the bulk of the required distributed service architecture currently used directly by developers in RPC based middleware into the DSM system itself. We now term such a DSM system *DSM Middleware*.

If DSM middleware is to be successfully deployed and used by distributed application developers that would otherwise use object-oriented middleware, the benefits associated with object oriented middleware must be maintained. These benefits include:

- Platform independence - reliance should not be directly placed on hardware or operating system services, allowing development in heterogeneous environments.
- Ease of programming - like RPC in object-oriented middleware, DSM middleware should not require significant changes in programming style to accommodate distribution.
- Runtime deployment - to allow for evolving software solutions the addition of software artifacts should be allowed at runtime, and not restricted to compile time decisions.
- Scalability - as distributed applications deployed over RPC middleware may be large scale, and may be distributed over a wide geographic area, the DSM middleware should not hinder scalability.

2.2 DSM Challenges

Before we can clearly identify a suitable approach to the provision of DSM middleware, we first explore general approaches to DSM implementation. Within such approaches we identify themes of development that may be suitably tailored, or used "as is" within our own DSM middleware. We consider suitability based on the benefits of object-oriented middleware listed in our motivation section. We divide this section into three further parts based on the basic design choices of a DSM:

- Implementation level - where within existing middleware is it most appropriate to implement DSM.
- Consistency model - how best to afford sufficient consistency of a shared resource without hindering performance.
- Communications - how to enact communications appropriately to provide the propagation of state changes associated with DSM updates.

2.2.1 Implementation Level

As reported in [35] there exists a number of alternatives for determining the level of abstraction where a DSM implementation is to be deployed: from systems that maintain consistency entirely in hardware (e.g. [18]) to those that exist entirely in software (e.g. [27, 5]). Considering our requirement of a middleware solution spanning a heterogeneous environment, we cannot guarantee homogenous hardware support. Therefore, we focus exclusively on software supported DSM systems. Software DSM systems can be split into three classes: page-based; variable-based; object-based. In each of these approaches our concern is where, and how, transparency of remote access is introduced:

- Page-based - uses the *memory management unit* (MMU) to trap remote access attempts (e.g. [19, 3, 31]).

- Variable-based - requires custom compilers to add special instructions to program code in order to detect remote access requests (e.g., [6, 39]).
- Object-based - special programming language features are required to determine when a remote machine's memory is to be accessed (e.g., [30, 4]).

Due to platform dependencies (e.g., operating system), we cannot consider page-based solutions as an adequate approach for a heterogeneous solution to DSM middleware. Although variable-based may be possible as a certain degree of platform independence is provided, the compile time requirements restrict the ability to introduce new types during runtime. This leaves the possibility of object based solutions. Even though this approach appears to be tightly coupled to a programming language, it still affords a degree of platform independence beyond that offered by page-based systems. In addition, if the supported language allows the introduction of new types during runtime, then such a quality may be introduced into the DSM middleware.

2.2.2 Memory Consistency Model

A memory consistency model may be considered a contract between those elements of a system that access memory and the memory itself [1]. Choosing an appropriate memory consistency model is a trade-off between minimizing access order constraints and the complexity of the programming model: assuring strict consistency of the DSM for all accesses is achieved at the expense of performance as increased message passing coupled with the locking of resources is required.

For the type of DSM middleware that we are concerned with the single, most important, design choice affecting scalability is where, physically, to store memory. If such a storage space was consigned to a single location there is a greater potential for memory contention issues to arise, commonly resulting in bottlenecks. Furthermore, data may be geographically separated from an accessing process to such an extent that latency of message exchange may be sufficiently high as to hinder performance.

To afford scalable solution for DSM middleware a compromise must be reached regarding the consistency of memory against the performance incurred from using such memory. One design option would be to replicate shared memory across the DSM middleware, affording local access when appropriate, while seeking to maintain a degree of consistency across replicas to ensure successful application operation. For example, distributed transactions may be employed to ensure state changes of one or more memory replicas are achieved in a consistent manner. Alternatively, group communication protocols offering total ordering and atomic multicast may be used to ensure all state changes are viewed in a mutually consistent way. As these techniques themselves come with a performance cost, they must be used only when needed (i.e., during state change events).

2.2.3 Communication Channel

In a DSM middleware where nodes and processes are separated over some geographic distance the choice of communication medium is limited. In fact, to ensure availability of such a system for the widest audience a developer must rely on standard protocols such as those governing public access network traffic (e.g., TCP/IP for the Internet). As existing middleware provides a convenient, easy to use, communication abstraction for developers over such protocols, it would be folly not to exploit such middleware in DSM middleware.

As RPC is the primary mechanism for enacting communication within existing middleware, one must consider RPC as a suitable communication mechanism on which to construct DSM middleware. Using RPC requires a communication stream between sender and receiver to be initialised and maintained either throughout a call or for as long as RPC participants hold references to each other (usually sender holding reference to receiver). This approach to tightly coupled communication is satisfactory for small numbers of participants but does not scale to support hundreds and thousands of participants. However, in a DSM system it is quite conceivable that hundreds, if not thousands, of clients may at some point in an application's execution

require access to a memory location. Alternatively, a client may require access to many hundreds, if not thousands, of memory locations. RPC used in such a manner is not scalable, as the management of connections at both client and server side would be a substantial drain on available processing resources.

This scalability problem has been tackled by middleware developers via abstracting away the one-to-one communication model of RPC in favour of a many-to-many solution. This is achieved by providing messaging services that decouple sender from receiver (e.g., sender does not know who is receiving its messages [10]). In distributed systems such an approach to message exchange is encapsulated in Message Oriented Middleware (MOM) [21] with the Java Message Service (JMS) [34] providing an example implementation for Java.

In MOM senders publish their messages onto well known message channels while receivers express interest in receiving messages from such channels. The use of MOM allows additional services, such as message ordering, to be abstracted away from the concern of the programmer to the systems level.

2.3 Discussion

To present a DSM middleware for use by developers a suitable approach to implementation would be via an object-based approach, affording runtime introduction of software artefacts. Although not ideal, as language dependency is still present, such an approach would provide a significant degree of platform independence. The Java programming language offers a semi-platform independent solution as the java virtual machine (JVM) is available on most platforms and is in widespread use in existing RPC middleware solutions. In addition, the reflective qualities of Java coupled with the serializability of object instances allow new object types to be introduced at runtime.

As we are deploying our DSM using multiple machines over a wide geographic area, we need to employ replication techniques to minimise the possibilities of bottlenecks and excessive access delays due to network latencies. This will raise a significant challenge in ensuring consistency of data across replicated memory locations. Therefore, an agreement protocol will be required to maintain a degree of consistency to enable the DSM to satisfy the requirements of applications. This matter, however, is very complicated in distributed systems owing to the unavailability of an absolute global time. The main consequence of this is that it is not always possible to determine the order in which events occurred due to the asynchronous nature of the network (message delays are bounded but unknown) coupled with the non-determinism of multi-threaded process execution on pre-emptive operating systems [15]. Extensive work has been carried out in this area; for the purposes of our work, we adopt the sequential [16], and the Pipelined RAM (PRAM) consistency models [20]. These models offer a compromise between programming constraints and implementation complexity.

An integral part to any agreement protocol is the ability to support ordering and reliability of message delivery. By choosing a MOM solution for our communication channel we not only provide a scalable solution for message exchange, but can make use of associated services that may provide ordering and reliability guarantees when required. This will ease the overall development of the DSM agreement protocol. As we have identified Java as a suitable implementation language for our DSM so we choose JMS as our MOM technology.

2.4 Paper Contribution

The contributions of the paper may be summarized as follows:

- The combination Java/object-based runtime allows us to deploy our DSM in heterogeneous environments;
- The use of a MOM system allows us to provide a scalable architecture;
- By utilising existing middleware services in our DSM middleware we aim to provide QoS guarantees (e.g., atomic delivery order of messages, exactly-once semantic);

- The use of Java and MOM coupled with building our DSM middleware using existing middleware techniques means we can afford the developer runtime introduction of new types.

Our approach is to utilise the notion of independent subsystems to manage consistency for scalability while utilising the benefits that existing middleware platforms provide.

3 Implementation

This section summarizes the most interesting issues we have addressed during the prototype implementation developed using the Java programming language and the Java Message Service technology. Before we describe the individual components of our system, we first describe an overview of the system as a whole.

In our system, the application developer does not have to be concerned with interactions with the shared object instances themselves; rather, he/she interacts with wrapper objects which provide the abstraction of DSM, as depicted in Fig. 1. This approach originates from the technique used by page-based implementations. In order to obtain a similar behaviour, the shared memory abstraction is based on two elements; namely, wrapper objects and memory addresses. In other object-based implementations, *e.g.* Orca [4], processes communicate via object methods. The wrapper object is the DSM coherence unit while a memory address univocally locates a wrapper: given a replicated wrapper object o , all replicas have the same address $A(o)$. The main advantage of this scheme is that it supplies a single system image, that is all processes reading (writing) the memory address x read (write) the same item.

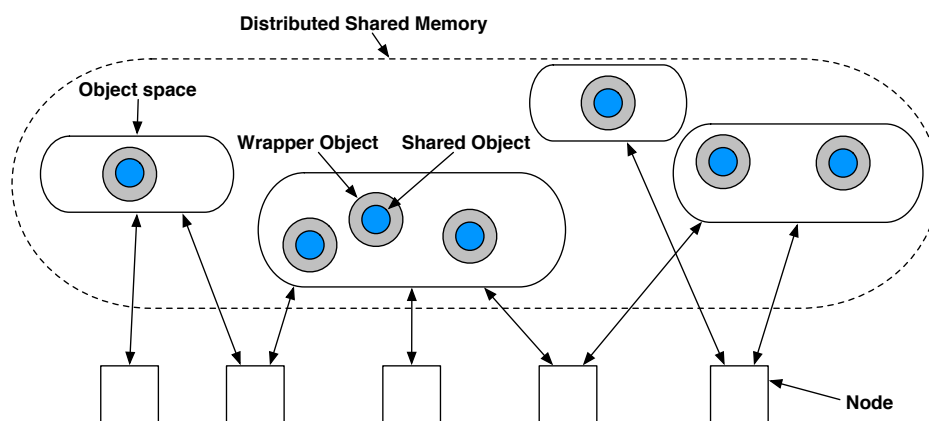


Figure 1. Nodes communicate through the memory.

3.1 Local Memory

As the proposed scheme is object-based it uses some techniques adopted by page-based protocols: the main difference is that the Java programming language doesn't allow manipulating the physical memory. This matter can be viewed as a shortcoming (since it adds some overhead) as well as an advantage, since it is platform independent. In order to solve this limitation we decided to implement the local memory abstraction through two hash tables:

1. *memory*, acting as local cache: keys map to memory addresses while values map to `ISharedObject` instances, that is wrapper objects (since shared objects are transmitted over the network we require the content to be `Serializable`);
2. *subscriptions*, storing subscribed topics. The key is the topic name while the second item is an object containing all JMS objects needed during communication phases.

Since the system is asynchronous, for what concerns the shared memory management we faced the following two challenges: how to update the local cache, and how to notify the application when updates happen. In order to solve these two challenges, the runtime system is structured as depicted in Fig. 2. This runtime system consists of (1) a `MessageListener` object for every subscribed topic, and (2) the *event dispatcher*. The *event dispatcher* is based on the *Observer* design pattern [12], which guarantees that every time the subject is updated all observers are automatically and transparently notified.

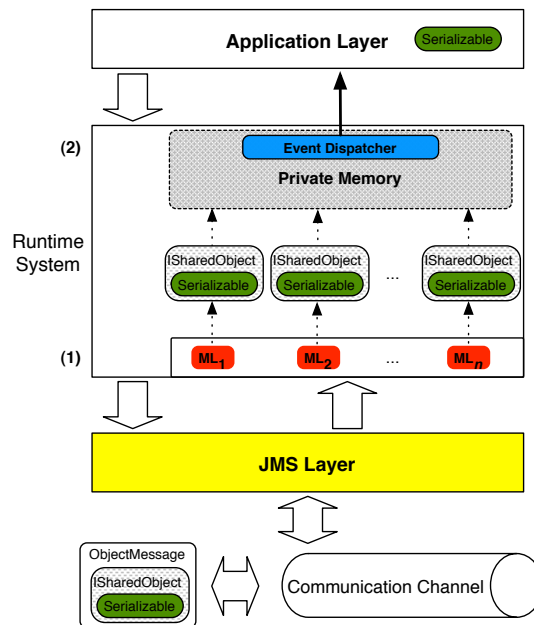


Figure 2. Updates management.

As consequence, the shared memory API is composed by three methods:

- `void write(ISharedObject)`: writes a wrapper object to the shared memory. The address to write is contained into the argument.
- `void read(Address)`: reads the specified shared memory address. Because of the introduction of the event dispatcher this method returns no value.
- `void deleteLocal(Address)`: locally deletes the memory zone bound with the specified memory address. As result the topic specified by the function argument is unsubscribed.

3.2 Remote Data

Our approach allows the runtime system to distinguish between local and remote access attempts; local reads are useless because of the introduction of the Observer design pattern, while remote access requests are satisfied with a three-step algorithm:

1. Transmission request to the topic T bound with the memory address;
2. Local memory synchronization, that is creation of a new replica;
3. Subscription of the topic T .

Since memories are physically distributed, interaction between system components happen only through messages: during the step number 2 we must guarantee that the requesting node will receive only one (correct) reply in order to maintain the consistency among replicated data. The solution to this issue is the solution to the consensus problem.

3.3 Agreement Protocol

Several definitions of the consensus problem can be found in literature [8, 17, 23]. For the purpose of our discussion we state the consensus problem in the following terms: given a collection of processes p_1, \dots, p_n ($n > 1$) communicating by message passing, every process begins in the undecided status and proposes a single value. Following a deterministic protocol, at some point during its computation a process must irreversibly decide on a single value v_i drawn from a set $\mathcal{V} = \{v_1, \dots, v_n\}$.

If every correct process proposes a value then an algorithm is a consensus protocol only if it satisfies the following three properties:

- *Termination*: every correct process eventually decides a value;
- *Agreement*: all correct processes decide the same value;
- *Integrity*: if the correct process p_j decides v_i then some correct process has proposed that value.

A JMS message is an object composed by three fields, *Header*, *Properties* and *Body*. Our architecture employs only two of the six available classes: namely, `TextMessages` sent during elections together with message properties, and `ObjectMessages` used to maintain memory coherence. While messages containing `ISharedObjects` are handled by the *event dispatcher*, election messages are handled by the message listener.

A protocol run starts when a process p_i would like to join a topic T (e.g. when a remote memory access attempt is made): p_i creates a temporary queue Q needed to receive the synchronization message and publishes to T a message containing its identifier and Q 's identifier (message $j_{3,Q}$ in Fig. 3).

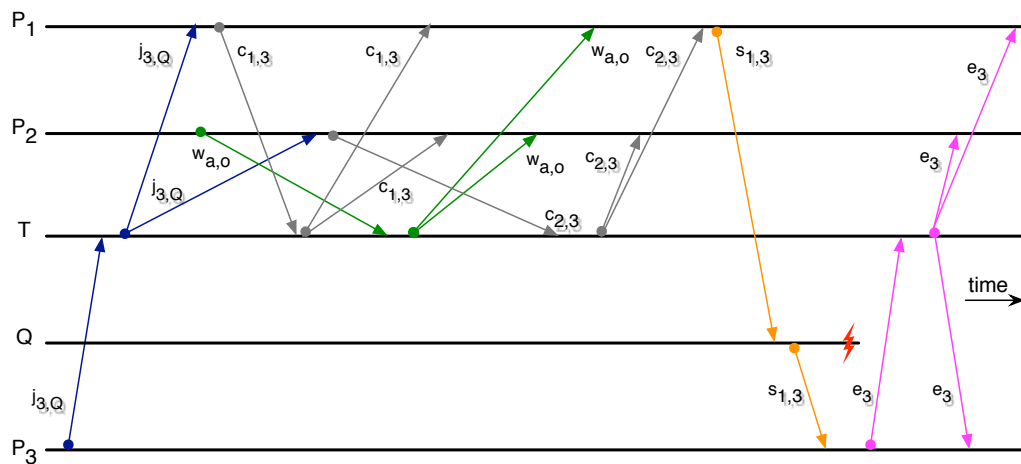


Figure 3. The agreement protocol.

When a process p_j ($\forall j \in T$) falling into T receives a request, it stops immediately outgoing memory communications to the T 's channel (further requests will be buffered), and then publishes to the topic its own proposal (messages $c_{1,3}$ and $c_{2,3}$). To find an agreement, our protocol exploits the delivery order warranty provided by JMS: since all nodes belonging to the same subsystem receive messages in the same order, the leader is the sender of the first received message.

The elected process continues by sending a synchronization message to the queue Q containing all `ISharedObjects` bound to \mathcal{T} .

Fig. 4 illustrates the following coherence problem. After the receipt of the message e_3 , P_3 's cache differs from the one of both P_1 and P_2 because the leader P_1 sent the synchronization message before receiving P_2 's update.

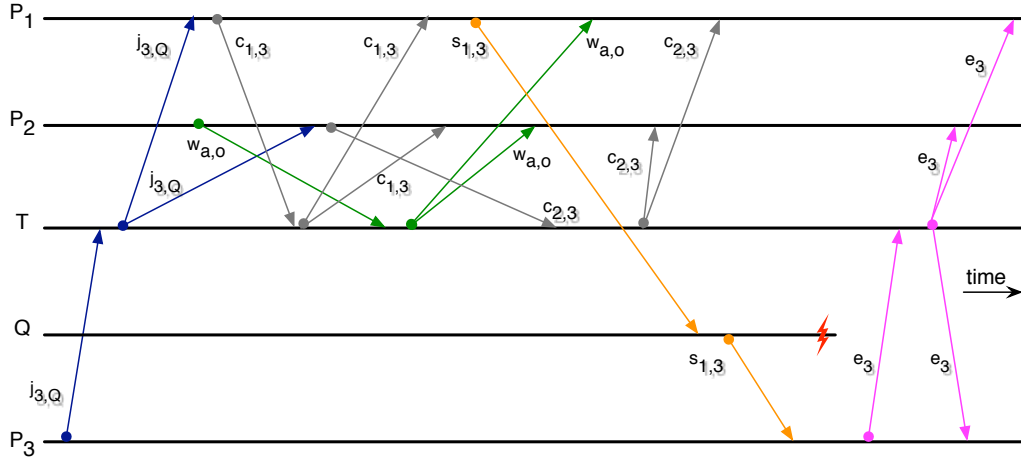


Figure 4. Naive agreement protocol.

In order to avoid coherence problems such as that illustrated in Fig. 4, the synchronization message (arrow $s_{1,3}$) is sent by the designated coordinator only when all proposals are received. As just described, since processes stop outgoing memory communications when they receive a join request, the causal delivery order guarantees that *all* updates (marked as $w_{address,object}$) are propagated *before* the leader receives the last proposal.

When the initiator node receives the synchronization message, it updates its own memory, deletes the queue Q , subscribes the topic \mathcal{T} , and finally publishes a message ending the protocol, marked as e_3 . When processes receive an e message from the channel \mathcal{T} they restart outgoing memory communications to \mathcal{T} .

The proposed scheme has however a serious shortcoming: since the coordinator process must wait until all proposals are received, it has to know how many processes belong to \mathcal{T} ; hence, because of the FLP theorem [11], this protocol allows no crash.

If a process is no longer interested in a topic \mathcal{T} , a protocol allowing that node to leave \mathcal{T} is used. This algorithm is very similar to the previous one and thus is not described; the main difference is that here the number of available nodes is decremented.

3.4 Object Updates

To distinguish between new objects and updates the `ISharedObject` data type is extended by two interfaces, `INewData` and `IUpdate`.

During synchronizations, the requesting process receives a hash table containing only `INewData` objects. Thus, the message content is stored into the memory hash table as it is; every time the local cache is updated, registered observers are notified.

When updates happen the matter is much more complicated and the proposed scheme exploits the observation that when operations are commutative they can be reordered without affecting the final state [25, 37]. More in detail, if the received message contains an “entire” object the content is stored into the local cache (this could mean, for example, that operations are not commutative) while if the message content is an update then the original object is modified through Reflection.

4 Experiments and Results

In this section we present the preliminary results of the tests we conducted on our prototype architecture.

Our testbed environment consists of a cluster composed by Pentium 4 PCs running Linux 2.6.10 and JVM Sun 1.5.0.02 and connected by a 100 Mbit Fast Ethernet LAN. More in detail, the used JMS provider was Joram 4.3.1 [26] with each node deployed on a Pentium 4 3.0 GHz with 1 GB of RAM while clients were deployed on Pentium 4 2.4 GHz with 512 MB of RAM.

Before we discuss the test results it is important to outline that the network environment as well as CPU resources were shared with other users.

A benchmark application suite has been developed to determine the components overhead, the agreement protocol cost, the update cost and protocol behaviour compared against a client-server architecture. A detailed description of the benchmarks is given during the tests discussion.

4.1 Components Overhead

The aim of this test was to measure the overhead added by each component. More in detail, we measured the performance differences between (i) TCP and (transient) JMS, (ii) transient and persistent JMS, (iii) “local” and “remote” connections, and (iv) JMS and our DSM.

The overhead has been calculated as roundtrip time depending on the message size: tests were conducted on messages of size 1 Byte, 194 Bytes, 1 Kbyte, 10 Kbytes, 100 Kbytes and 1 MByte while shown results are the mean values of 1000 measurements.

TCP vs. JMS Fig. 5(a) shows the overhead introduced by JMS: while TCP provides with an *at-most-once* semantic with FIFO order, this JMS configuration (single server, transient) guarantees the same semantic but a causal atomic delivery order. Measurements show that the overhead is about the same in absolute terms and thus its impact decreases when the message size increases: while publishing a single byte with JMS is about 11 times more expensive than TCP, the difference when a 1 MByte message is sent drops to about 7%.

The second interesting thing to note is the difference between 1 Kbyte and 10 Kbytes messages: since the network MTU is 1500 bytes the first message is contained into one IP packet while the second one is split into seven fragments. In these conditions the cost of a TCP send operation grows of a factor of 2.5 while the JMS one grows of only 1.5 times.

Transient vs. Persistent mode This test measures the overhead needed to guarantee an *exactly-once* delivery semantic: when the persistent mode is used in transit messages are not lost due to a JMS provider failure.

In this scenario the JMS provider is distributed among three nodes: Fig. 5(b) shows that the overhead goes down while the message size grows, however a difference of about 58% lasts when a 1 Mbyte message is published.

Transparency cost The third experiment measures what happens when a JMS client connects to a server (usually JMS clients create only one connection) but sends messages to a topic deployed to another server. The scenario depicted in Fig. 6 shows three clients (c_1 , c_2 and c_3), a distributed JMS architecture composed by three servers (s_1 , s_2 and s_3) and two topics, t_2 and t_3 , deployed respectively on server s_2 and s_3 . Both c_1 and c_2 communicate to c_3 (that is subscriber of both topics), but the interaction is different: c_1 is connected to s_1 but publishes messages to t_3 while c_2 is connected to s_2 and sends messages to t_2 . It’s obvious that the message exchange between c_2 and c_3 is faster than the one between c_1 and c_3 : our goal is to quantify this difference.

This test was carried out by running a distributed JMS configuration composed by three nodes configured in transient mode. Fig. 5(c) shows an off-beat result: the difference remains the same (26%) until the message size reaches 100 Kbytes, then it grows until about 37% for 1 Mbyte messages. In order to explain this behaviour Fig. 6 is needed: assuming that only one publisher and one subscriber exist, if the topic and the connections are on the same server two messages are

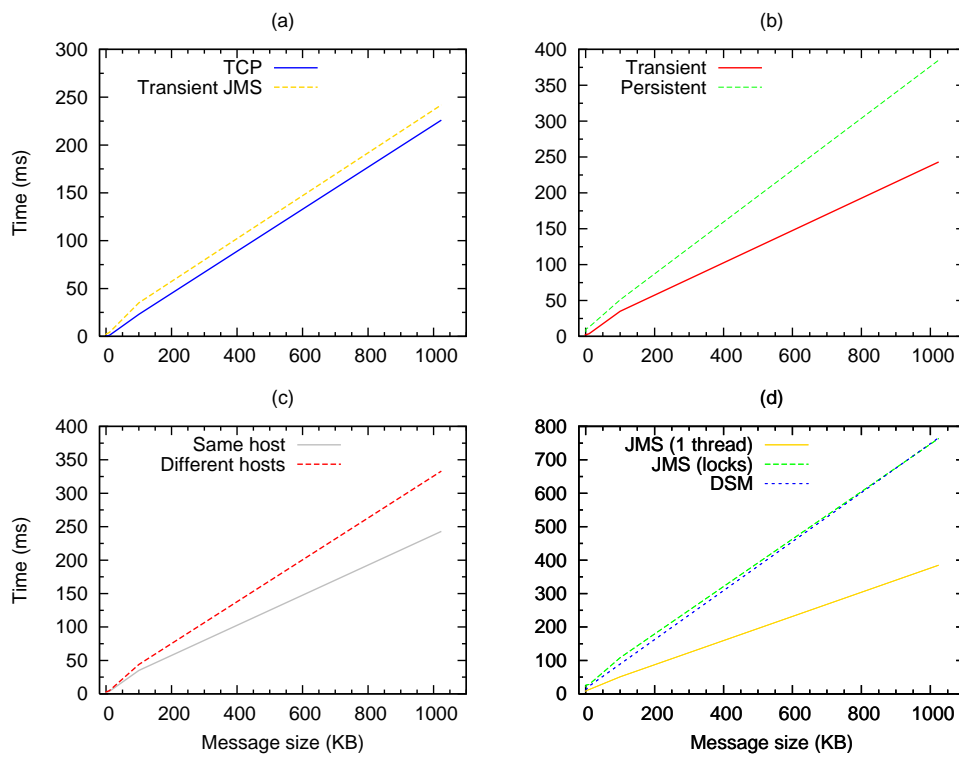


Figure 5. Components cost.

sufficient ($c_2 \rightarrow s_2, s_2 \rightarrow c_3$), else at least three messages are needed; as shown in Fig 7(a) this justifies the high overhead introduced by the runtime system when large messages are handled.

Unfortunately there is no immediate solution to this problem: the use of a distributed architecture requires some form of synchronization, and it happens by messages exchange. The problem, instead, is related with the JNDI: since it is fully transparent, clients looking for an object don't know where physically it has been deployed.

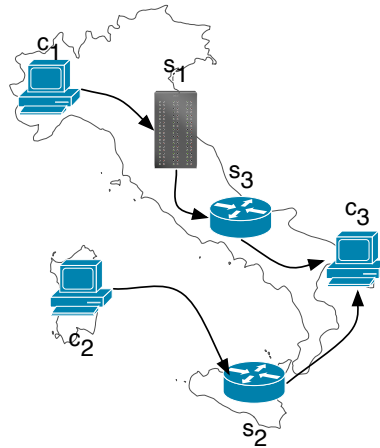


Figure 6. Third test scenario.

JMS vs. DSM The last of this set of tests measures the overhead introduced by the runtime system. The communication channel is the one provided by three Joram nodes, configured as persistent. Results shown in Fig. 5(d) include the following three scenarios: (i) a JMS client using a single thread to receive as well as to publish messages, (ii) a multithreading client using one thread as publisher and the other one as subscriber, and (iii) a client using the facilities provided by the runtime system. Messages are sent by the `write()` primitive while incoming messages are handled by the event dispatcher.

In this experiment once the message has been published the client waits its receipt and thus the single-thread version performs better: the problem, however, is that in a real scenario the two operations are handled individually.

The two-thread application can be modelled by the producer-consumer problem with a bounded buffer of one-item capacity: the overhead needed to synchronize the two threads is very high (98% for 1 Mbyte messages).

Finally the DSM guarantees the coherence of replicated data as well as a synchronization mechanism: results show that the runtime system performs quite well only until messages are small (1 Kbyte). Further analyses are needed to explain this phenomenon.

4.2 Agreement Protocol

The second test evaluates the cost of shared memory synchronizations. Assuming a system composed by n nodes of which k fall into the considered subunit ($k \leq n$) all nodes receive $k + 2$ messages and publish only one message (except for the coordinator, that sends two messages).

Experiments were repeated five times: average results, depicted in Fig. 7(b), show that external factors are much more important than the number of nodes involved into the election. As already discussed both network and computation resources were shared during tests: this explains why 1371 *ms* are needed to allow the admission of the node number 28 while the protocol requires only 350 *ms* when nodes are 31.

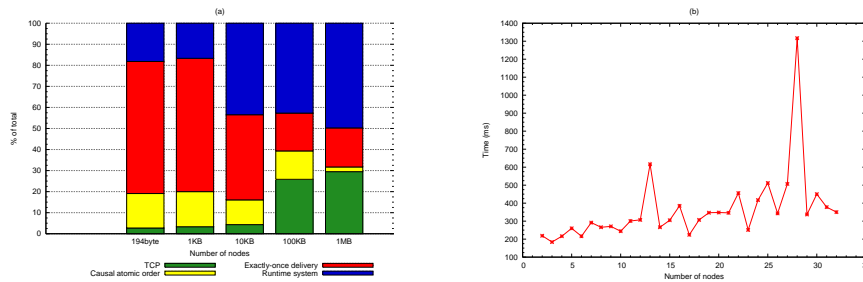


Figure 7. (a) Components overhead and (b) Agreement protocol.

4.3 Updates

In this section we show update test results related with the sequential consistency model. The JMS provider is composed by three persistent nodes while DSM processes interact through a clustered topic. The clustered topic abstraction supplies a fault tolerant mechanism but does not provide any form of load balancing. The discussion focuses on three points:

1. What happens by varying the number of nodes (1, 4, 8, 16 and 32);
2. How the system reacts when the amount of requests per node ranges from 125 to 500, using increments of 125;
3. What is the system behaviour depending on the system configuration (25%, 50%, 75% and 100% of nodes falling into the same subunit).

Average cost depending on the number of nodes This test is not very revealing because nodes repeatedly try to update the shared memory as fast as possible. This is equivalent to the following code:

```
while (updates < toDo)
    publish(msg);
    update++;
```

The problem is that this chunk of code clashes with the provider ability to handle requests. The big difference shown in Fig. 8 is related with the topic architecture and thus adding other machines to the provider only partially mitigates the issue.

Average cost depending on the number of requests Once again results correspond to the attended ones. When the number of clients is four there is no substantial difference, but if the number of nodes grows any more the provider is no longer able to satisfy requests on the flight.

For what concerns spurious values, when the number of nodes is high they can be attributed to external factors while when it is low (and thus the throughput is high) it becomes fundamental where the connection is created.

Average cost depending on the system configuration As expected the configuration where all nodes belong to the same subunit performs always worst because its semantic is equivalent to broadcast communications. For what concerns other configurations, the one providing 50% of nodes falling into the same subsystem performs very well, sometimes even better than the one where only 25% of nodes belong to the same subunit. This is because the 25% test was carried out together with

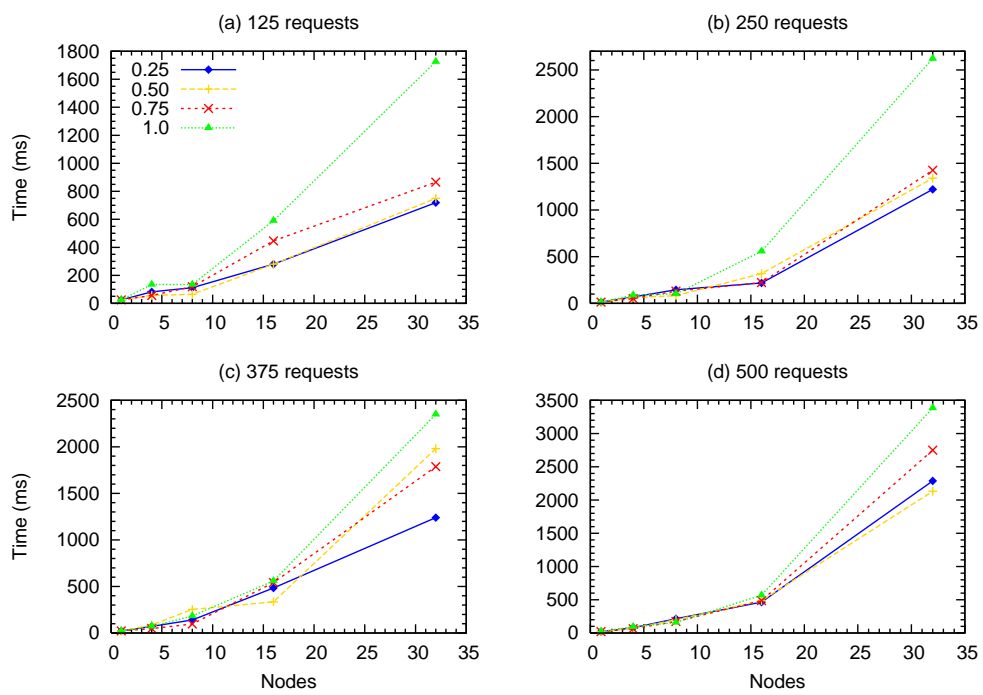


Figure 8. Updates.

the 75% one. Now, since the JMS provider knows the number of both publishers and subscribers it can allocate more resources where they are needed.

4.4 Comparison Analysis

In this Subsection, we compare and contrast our DSM architecture with a synchronous, client-server system. That system consists of a centralised server providing its clients with the shared memory abstraction. The server is deployed on a Pentium IV 3.0 GHz with 1 GByte of main memory, clients are deployed on Pentium IV 2.4 GHz with 512 Mbyte of RAM.

Clients interact with the shared memory via a clustered queue by using `write` and `read` synchronous primitives. As the centralized server serializes all requests, the read call returns the last written value. The clustered queue provides the clients with a load balancing mechanism; however, it guarantees no fault tolerance.

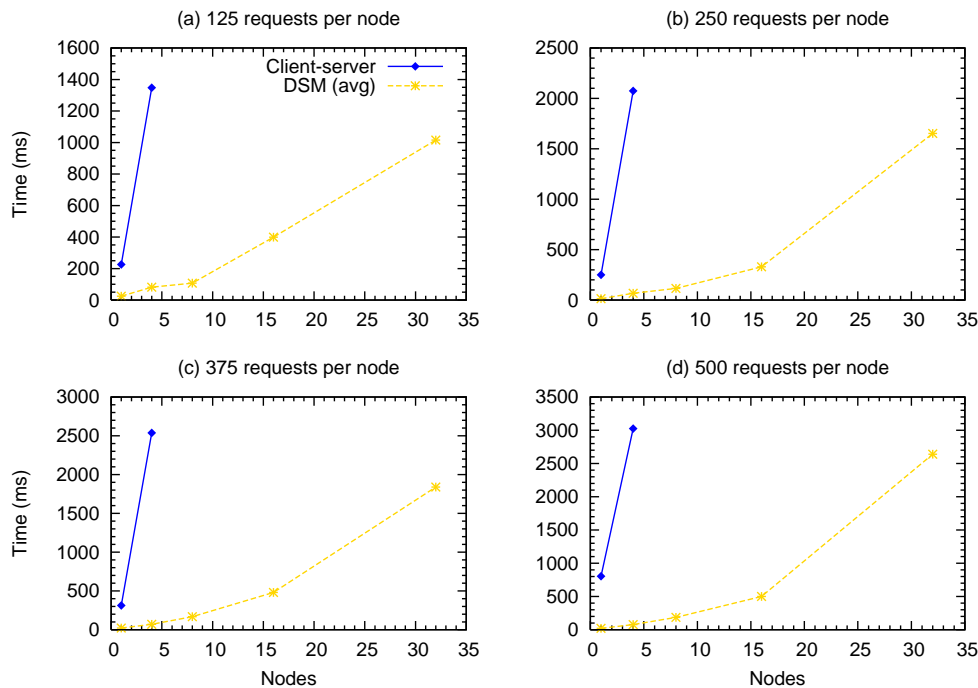


Figure 9. Comparison between the DSM and a client-server model.

4.4.1 Test Result

Fig. 9 shows the comparison results where values for the DSM are the average values of the previous test. Our system outperforms the client-server system as the latter quickly becomes unresponsive, as the client load augments. Furthermore, while the DSM problem comes from the communication channel, the client-server shortcoming is an architecture matter: if replicated servers are used the problem of replica control arises.

Client-server values are available only for one and four nodes because we were unable to complete the test when more clients were used, owing to the crash of the server where the clustered queue was deployed.

5 Related Work

In this section we compare our DSM middleware system with other DSM systems that have been designed using a design approach similar to ours.

Previous systems usually don't address problems related with the communication channel and in particular several implementations assume that packets are delivered in the right order while messages cannot be lost; TreadMarks [3] and IVY [19], for instance, use a combination of UDP and timeouts to maintain coherence among replicas. While this shortcoming is not such a hinderence in LAN environments (where message latency is much lower than in the Internet), this design choice is much more critical since we don't make any assumption about the communication channel.

By far the most popular DSM systems have been built with a predefined combination of operating system/hardware architecture combinations (e.g., TreadMarks, IVY, Brazos [31], Cashmere [14]) or rely on modified compilers (e.g., Munin [6], Shasta [27], Orca [4], Jackal [36]). This approach also extends into approaches based on the Java language with modifications required within the JVM (e.g., Java/DSM [38]).

The way memory coherence is maintained in our DSM middleware is similar to the solution adopted by TreadMarks; however, the two systems differ in several ways. For example, at the implementation level, our DSM system is object-based while TreadMarks is page-based. Tence, TreadMarks may be affected by false sharing and fragmentation problems which require additional protocols, such as the adaptive protocols proposed by [2], in order to be minimised. In addition, at the communication level, TreadMarks is based on the exchange of UDP messages while our solution relies on JMS.

Scalability is rarely confronted in existing systems similar to ours: TreadMarks, for example, propagates updates to all nodes while our solution "hits" interested nodes only; JDSM [30] uses a centralized node (the ClusterManager) to intercept requests which becomes a bottleneck as the number of nodes increases.

The use of group communication is not a new idea. Orca [5] uses group communications to implement a sequentially consistent object-based DSM. Orca, like the system described in this paper, relies on an a communication channel feature (reliable, total ordered broadcast) to implement transparent replication as well as to maintain data objects consistent. Brazos uses two main system threads (under this point of view our system is an evolution of Brazos since it uses two threads for every subscribed topic) to reduce communication overhead and uses a selective multicast to reduce communication traffic [32]. Finally, the work described in [29] is an object-based DSM designed as an extension of the .NET Framework. It provides a causally consistency memory model where the causal relationship is achieved trough vector logical clocks, that are sent within every message.

Java has already been used to implement a DSM system. Java/DSM [38] solves problems arising when a heterogeneous set of nodes are used, however it modifies the memory management of the JVM and thus it is not portable. JDSM, like our system, doesn't modify the JVM, however it is quite different from the solution we propose. First, it requires shared objects to be declared during the initialization step. Second, its architecture, composed by three elements (ClusterManager, Server and Client) requires that requests be sent from the Client to the ClusterManager and then forwarded to one of the available Servers. This may be viewed as hindering scalability as the ClusterManager may be viewed as a bottleneck. Finally, interaction between nodes can use three different communication protocols, TCP/IP socket, PM and VIA. Again, the lack of MOM in a middleware setting makes unavailable to the DSM middleware services that can be incorporate at little cost (e.g., transactions)

Java Past Set (JPS) [24] differs from our system primarily because shared objects are not replicated (and thus for the reasons explained in section § 2.2.3 scalability problems arise when the number of accessing clients increase). Data are located thanks to *element object descriptors* and can be distributed according to several policies (*i.e.* place the object to the first node that requested it or try to distribute the same number of objects to every node). Thus, in JPS, data distribution can happen in several ways while we replicate data on demand and remove replicas when they are

no longer needed.

One thing JPS and our DSM have in common is the data update phase, defined as *user re-definable memory semantics*, allowing the application programmer to define the memory operation semantic.

6 Conclusions

In this paper we have presented a wide-area distributed shared memory middleware. Compared with existing approaches for heterogeneous computing, this platform provides greater transparency to the application programmer: it has been pointed out that the proposed protocol does not need any hardware or software architecture assumption (differences are hidden by Java). Second, the runtime system transparently handles the message passing details such as data replication. Third, it is also able to use non reliable channels, where packets can be lost, and delays are of arbitrary length. Finally, we have shown the applicability of group communications, replication, caching and interest management techniques to support the construction of event-oriented distributed systems.

Some open problems still remain, such as fault tolerance, where the use of more realistic assumptions about the communication channel as well as the use of randomized algorithms is left unexplored, or the problem related with the client connection to the JMS provider, which is responsible for the poor performances achievable in some circumstances. Solutions to these problems are being investigated as part of our current research activity.

References

- [1] ADVE, S. V., AND HILL, M. D. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)* (May 1990), IEEE, pp. 2–14.
- [2] AMZA, C., COX, A. L., DWARKADAS, S., JIN, L.-J., RAJAMANI, K., AND ZWAENEPOEL, W. Adaptive Protocols for Software Distributed Shared Memory. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory Systems 87*, 3 (March 1999), 467–475.
- [3] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer 29*, 2 (February 1996), 18–28.
- [4] BAL, H. E. Orca: A Language for Distributed Programming. *ACM SIGPLAN Notices 25*, 5 (1990), 17–24.
- [5] BAL, H. E., BHOEDJANG, R., HOFMAN, R., JACOBS, C., LANGENDOEN, K., RÜLH, T., AND KAASHOEK, M. F. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems 16*, 1 (1998), 1–40.
- [6] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)* (1991), ACM Press, pp. 152–164.
- [7] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems 13*, 3 (August 1995), 205–243.
- [8] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM (JACM) 34*, 1 (January 1987), 77–97.
- [9] DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. A. Synchronization, Coherence, and Event Ordering in Multiprocessors. *IEEE Computer 21*, 2 (February 1988), 9–21.
- [10] EUGSTER, P. T., FELBER, P. A., GUERRAOU, R., AND KERMARREC, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR) 35*, 2 (June 2003), 114–131.
- [11] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM) 32*, 2 (April 1985), 374–382.
- [12] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [13] IFTODE, L., SINGH, J. P., AND LI, K. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)* (New York, NY, USA, 1996), ACM Press, pp. 122–133.

REFERENCES

- [14] L. KONTOTHANASSIS, R. STETS, G. HUNT, U. RENCUZOGULLARI, G. ALTEKAR, S. DWARKADAS, AND M. L. SCOTT. Shared memory computing on clusters with symmetric multiprocessors and system area networks. *ACM Transactions on Computer Systems*, 23(3):301–335, August 2005.
- [15] LAMPOR, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21, 7 (July 1978), 558–565.
- [16] LAMPOR, L. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* C-28, 9 (September 1979), 690–691.
- [17] LAMPOR, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.
- [18] LENOSKI, D., LAUDON, J., JOE, T., NAKAHIRA, D., STEVENS, L., GUPTA, A., AND HENNESSY, J. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)* (New York, NY, USA, 1992), ACM Press, pp. 92–103.
- [19] LI, K., AND HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7, 4 (1989), 321–359.
- [20] LIPTON, R. J., AND SANDBERG, J. S. PRAM: A Scalable Shared Memory. Tech. Rep. CS-TR-180-88, Princeton University, September 1988.
- [21] MENASCE, D. A. MOM vs. RPC: Communication Models for Distributed Applications. *IEEE Internet Computing* 9, 2 (March/April 2005), 90–93.
- [22] MOSBERGER, D. Memory Consistency Models. Tech. Rep. TR 93/11, University of Arizona, 1993.
- [23] PEASE, M., SHOSTAK, R., AND LAMPOR, L. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)* 27, 2 (April 1980), 228–234.
- [24] PEDERSEN, K. S., AND VINTER, B. Java PastSet: A Structured Distributed Shared Memory System. *IEEE Proceedings – Software* 150, 2 (April 2003), 147–153.
- [25] PU, C., AND LEFF, A. Replica Control in Distributed Systems: An Asynchronous Approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD '91)* (New York, NY, USA, 1991), ACM Press, pp. 377–386.
- [26] SCALAGENT DISTRIBUTED TECHNOLOGIES. JORAM: Java Open Reliable Asynchronous Messaging, 2005. <http://joram.objectweb.org>.
- [27] SCALES, D. J., GHARACHORLOO, K., AND THEKKATH, C. A. Shasta: a Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (New York, NY, USA, 1996), ACM Press, pp. 174–185.
- [28] SCHÖTTNER, M., WENDE, M., GÖCKELMANN, R., BINDHAMMER, T., SCHMID, U., AND SCHULTHESS, P. A Gaming Framework for a Transactional DSM System. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)* (Tokyo, Japan, May 2003), IEEE Computer Society, pp. 502–509.
- [29] SEIDMANN, T. Distributed Shared Memory Using The .NET Framework. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)* (Tokyo, Japan, May 2003), IEEE Computer Society, pp. 457–462.
- [30] SOHDA, Y., NAKADA, H., AND MATSUOKA, S. Implementation of a Portable Software DSM in Java. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI '01)* (June 2001), ACM Press, pp. 163–172.
- [31] SPEIGHT, E., AND BENNETT, J. K. Brazos: A Third Generation DSM System. In *Proceedings of the First Usenix Windows NT Symposium* (August 1997), pp. 95–106.
- [32] SPEIGHT, E., AND BENNETT, J. K. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture* (February 1998), IEEE Computer Society, pp. 312–322.
- [33] STUMM, M., AND ZHOUMUNIN, S. Algorithms Implementing Distributed Shared Memory. *IEEE Computer* 23, 5 (May 1990), 54–64.
- [34] SUN. *Java Message Service*, 1.1 ed. Sun Microsystems, Inc, April 2002.
- [35] TANENBAUM, A. S. *Distributed Operating Systems*. Prentice Hall, 1995.

REFERENCES

- [36] VELDEMA, R., HOFMAN, R. F. H., BHOEDJANG, R. A. F., AND BAL, H. E. Runtime optimizations for a Java DSM implementation. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande (JGI'01)* (New York, NY, USA, 2001), ACM Press, pp. 153–162.
- [37] WUU, G. T., AND BERNSTEIN, A. J. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC '84)* (New York, NY, USA, 1984), ACM Press, pp. 233–242.
- [38] YU, W., AND COX, A. L. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience* 9, 11 (1997), 1213–1224.
- [39] ZEKAUSKAS, M. J., SAWDON, W. A., AND BERSHAD, B. N. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI'94)* (November 1994), pp. 87–100.