Engineering Distributed Shared Memory Middleware for Java

Michele Mazzucco[1,3,⋆], Graham Morgan[2], Fabio Panzieri[3], and Craig Sharp[2]

[1] University of Cyprus, Nicosia, CY 1678, Cyprus
[2] Newcastle University, Newcastle upon Tyne, NE17RU, UK
[3] University of Bologna, Bologna, 40127, Italy

Abstract. This paper describes the design, implementation and initial evaluation of an object-based Distributed Shared Memory (DSM) middleware system for Java. The resulting implementation allows the construction of event-based distributed systems using a simple programming model, allowing applications to be deployed without hardware or communication channel assumptions. Our implementation utilises standard, freely available, Message Oriented Middleware (MOM). This approach eases DSM development as many reliability and scalability issues associated to DSM may be handled by MOM. In addition to an implementation description, we provide performance results of a prototype system on a Local Area Network.

1 Introduction

The evolution of middleware architectures has provided developers with enabling technologies, easing the implementation of large-scale distributed applications deployed in heterogeneous environments. Such middleware identify the remote procedure call (RPC) 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 consider objects as the unit of distribution as this is by far the most popular approach supported in middleware.

Distributed Shared Memory. (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 run-time that transparently intercepts user access attempts to remote memory addresses and translates them into the appropriate messages.

Developing a DSM system for use with object-oriented middleware and providing distributed application deployment in heterogeneous environments would be beneficial.

[⋆] Michele was partly funded by the European Commission under the Seventh Framework Programme through the SEARCHiN project (Marie Curie Action, contract number FP6-042467).

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 531–548, 2009.

Developers would program their applications without the hindrance incurred from using the services required for distributed object implementation. This would 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 (who 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;
- Run-time deployment: to permit evolving software solutions, the addition of software artifacts should be allowed at run-time, and not be restricted by compile-time decisions.

The principal contribution of this paper consists of providing a practical implementation of a DSM protocol in order to support the construction of event-based application systems. In addition, our architecture can use non-reliable communication support (*e.g.*, the Internet), where packets may be lost, or experience unpredictable delays.

2 Design Issues

Before we can clearly identify a suitable approach for the provision of DSM middleware, we must 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 the previous section. We divide this section into three further subsections based on the basic design choices of a DSM:

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

2.1 Implementation Level

A number of alternatives exist for determining the level of abstraction where a DSM implementation is to be deployed: from systems that maintain consistency entirely in hardware to those that exist entirely in software. Considering our requirement of a middleware solution, we cannot guarantee homogeneous hardware support; we focus exclusively on software supported DSM systems. Software DSM systems can be split into three classes: page-based, variable-based, and object-based. In each of these approaches our concern is where and how transparency of remote access is introduced:

1. Page-based implementations use the *memory management unit* (MMU) to trap remote access attempts;
2. Variable-based run-times require custom compilers to add special instructions to program code in order to detect remote access requests;
3. Object-based systems use special programming language features to determine when the memory of a remote machine is to be accessed.

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 solutions may be possible (given that a certain degree of platform independence is provided) the compile time requirements restrict the ability to introduce new types during run-time. This leaves the possibility of object based solutions. Although this approach seems tightly coupled to a particular programming language, a degree of platform independence is afforded beyond that offered by page-based systems. In addition, if the language in question supports the introduction of new types during run-time, then this desirable feature may be incorporated into the DSM middleware.

2.2 Memory Consistency Model

Choosing an appropriate memory consistency model presents a trade-off between minimizing access order constraints and the complexity of the programming model: strict memory models (*e.g.*, sequential [14]) reduce complexity from the programmer's perspective but are achieved at the expense of performance; increased message passing coupled with the locking of resources is required. On the other hand, weak memory models (*e.g.*, release consistency [12]) grant improved performance, but allow the memory to return unexpected values. It is the duty of the programmer, using explicit synchronization techniques, to provide algorithmic semantics equivalent to the sequential model.

Given the type of DSM middleware under consideration, the most important design choice, in terms of scalability, is where to physically store memory. If a *single reader/single writer* algorithm is in operation, *i.e.*, if such a storage space is 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.

In order to realise a 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.

2.3 Communication Channel

To ensure availability 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 and practical communication abstraction for developers over such protocols, it would be imprudent not to exploit such 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 an initialised and maintained communication stream between sender and receiver, either throughout a call or for as long as RPC participants hold references to each other (usually sender holding reference to receiver). This tightly coupled approach to communication is satisfactory for small numbers of participants but does not scale to support hundreds or thousands of participants. RPC used in such a manner is not scalable, as the management of connections at both client and server would be a substantial drain on available processing resources.

Middleware developers have tackled this scalability issue by 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.*, the sender does not know who is receiving its messages [8]). In distributed systems such an approach to message exchange is encapsulated in Message Oriented Middleware (MOM) [17] with the Java Message Service (JMS) [28] providing an example implementation in Java.

In MOM, senders publish their messages onto well-known message channels (topics or queues, depending on the model in operation), 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.

2.4 Memory Design

To provide a DSM middleware for use by developers, a suitable approach to implementation would be object-based, while affording run-time introduction of software objects. Although not ideal (given that language dependency persists) 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 widely used by existing RPC middleware solutions. In addition, the reflective qualities of Java coupled with the serializability of object instances, allow the introduction of new object types at run-time.

Once the decision is taken to model the shared memory abstraction as a collection of shared objects, two further choices present themselves: (i) the algorithm to use and (ii) the memory consistency model.

As we are deploying our DSM on multiple machines using copies of objects (with a view to eventual distribution over a wide geographical area), we need to employ replication techniques that minimise the possibility of bottleneck and excessive access delays due to network latencies. This raises a significant challenge in ensuring consistency of data across replicated memory locations; hence an agreement protocol will be required to maintain a degree of consistency. This subject however, presents complexity in distributed systems owing to the unavailability of an absolute global time. The lack of such temporal-synchronisation means 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 preemptive operating systems. Our approach is quite similar to that proposed by [9] such that we exploit the ordering features provided by the communication channel – in particular, given two messages m_1 and m_2, if $m_1 \rightarrow m_2$[1] then all processes will receive m_1 before receiving m_2 – while the run-time system uses an algorithm which allows the update of remote replicas.

Extensive research has been carried out in the area of memory consistency models; for the purposes of our work, we adopt and experiment with (i) the sequential and (ii) the Pipelined RAM (PRAM) consistency models [16]. These models offer a compromise between programming constraints and implementation complexity, allowing us to evaluate the feasibility of our approach at an early stage.

An integral component of any agreement-protocol is support for ordering and reliable message delivery. By choosing a MOM solution for our communication channel we not only provide scalability for message exchange, but the opportunity to use as required, associated services potentially providing ordering and reliability guarantees; this will ease the overall development of the DSM agreement protocol. Having identified Java as a suitable implementation language for our DSM, so we choose JMS as our MOM technology. Consequentially, our system provides the following main features:

1. The combination Java/object-based run-time allows us to deploy our DSM in heterogeneous environments (possessing a Java Virtual Machine);
2. By utilizing existing middleware services in our DSM middleware we aim to provide QoS guarantees (*e.g.*, atomic delivery order of messages, exactly-once semantics);
3. The use of Java and MOM, coupled with our DSM middleware (constructed using existing middleware techniques) means we can grant the developer run-time introduction of new types;
4. Our *multiple reader/multiple writer* algorithm uses both partitioning and replication. Read operations are local (see Figure 1), while nodes interact only with a (dynamic) subset of shared objects, as shown in Figure 2. Furthermore, different processes can operate on the same object concurrently, as processes read/write their own local copies, thus increasing the degree of parallelism;
5. Distributed applications do not interact with the memory via an object's methods (this is by far the most common approach used by object-based DSM implementations). Instead, our system provides the same primitives of page-based systems (*i.e.*, `read` and `write`).

3 Implementation

In our system, interactions between actual shared object instances are not the concern of the application developer; rather, he or she interacts with "wrapper" objects, which provide the abstraction of DSM as depicted in Figure 2. This approach originates from the technique used by page–based implementations. In order to recreate similar behaviour, the shared memory abstraction is based on two elements; namely, wrapper objects and memory addresses. The wrapper object is the DSM coherence unit while a memory address unequivocally locates a wrapper (given a replicated wrapper object o, all replicas

[1] The → symbol indicates the "happened before" relation as defined in [13].

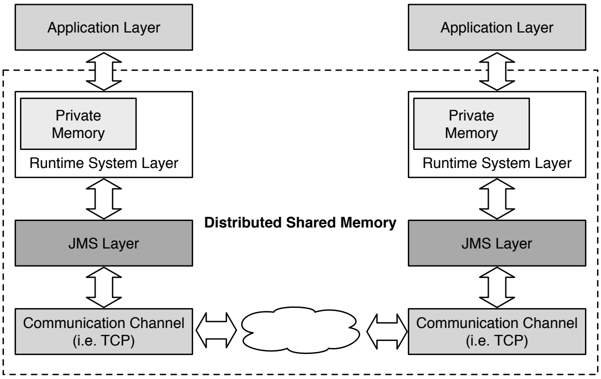


Fig. 1. DSM architecture. The cache allows for local reads

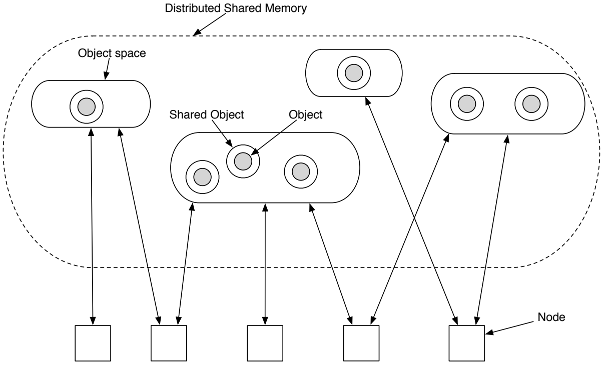


Fig. 2. The DSM uses both replication and partitioning to reduce the number of exchanged messages

share the same address $Addr(o)$). The main advantage of this scheme is that it supplies a single system image; all processes reading (or writing) the memory address x read (or write) the same item.

3.1 Local Memory

Even though the proposed scheme is object-based, it uses some techniques adopted by page-based protocols. The main difference is that the Java programming language does not allow the programmer to directly manipulate the physical memory. This is a shortcoming in the sense that it adds overhead, but also an advantage in terms of providing

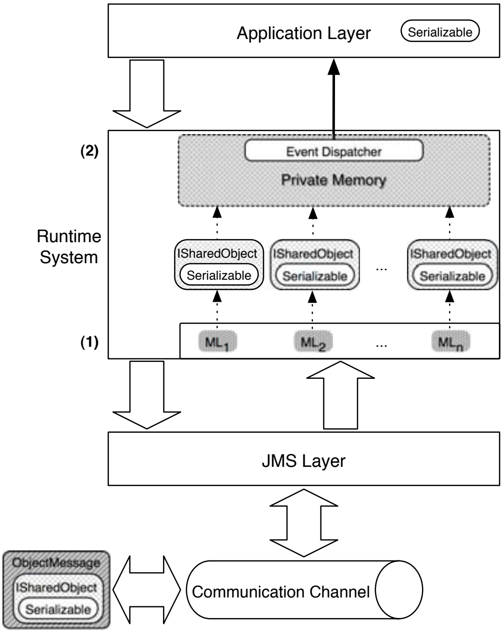


Fig. 3. Updates management

platform independence. In order to solve this limitation we decided to implement the local memory abstraction through two hash tables, $\langle k, v \rangle$:

1. `memory`, acting as local cache where the keys are memory addresses and the values are `ISharedObject` instances, specifically wrapper objects. As the objects stored in the DSM 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 value is an object containing all JMS objects needed during communication phases. Hence a "topic" in this discussion, represents a list of update-messages having relevance to specific shared-object replicas.

Since the system is asynchronous in nature, shared memory management challenges comprised of (i) how to update the local cache, and (ii) how to notify the application when updates happen. The run-time system we propose, depicted in Figure 3, consists of a `MessageListener` object for every subscribed topic and the *event dispatcher*. The event dispatcher is based on the *Observer* design pattern [11], which guarantees that every time the subject is updated, all observers are notified automatically and transparently. Consequently, the shared memory API is composed of three methods:

1. `void write(ISharedObject)`: writes a wrapper object to the shared memory. The address to write is contained in the argument;
2. `void read(Address)`: reads the specified shared memory address. In using the event dispatcher, this method need not return a value, as the application will be

notified as soon as the data becomes available (averting the possibility of reading an address that has not yet been written);

3. `void deleteLocal(Address)`: locally deletes the memory zone bound with the specified memory address, resulting in the topic specified by the function argument to become un-subscribed.

3.2 Remote Data

Our approach allows the run-time system to distinguish between local and remote access attempts. However, due to the introduction of the Observer design pattern, access attempts use this notification system whether local or remote reads are taking place. Remote access requests are satisfied using a three-step algorithm: Transmission request to the topic T is bound with the memory address; Local memory synchronization, *i.e.*, creation of a new replica; Subscription of the topic T.

Since memory is physically distributed, interaction between system components occurs only through messages. During the step number 3.2 the synchronization protocol 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. For the purpose of our discussion we state the consensus problem in the following terms: given a collection of processes $p_1, \ldots, p_n$ $(n > 1)$ communicating via message passing, every process begins in the undecided state 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 $V = \{v_1, \ldots, v_n\}$. If every correct process proposes a value then an algorithm is a consensus protocol only if it satisfies the following three properties:

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

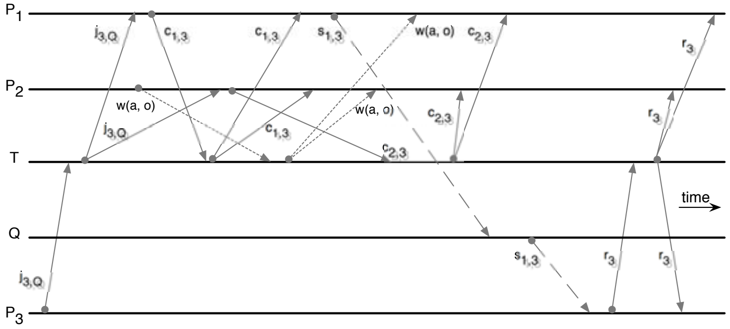


Fig. 4. Naive agreement protocol

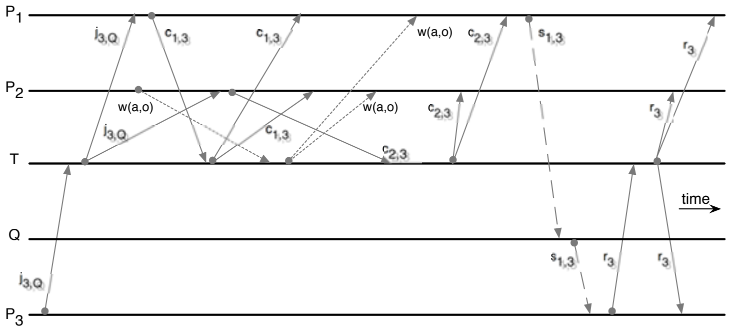


Fig. 5. The agreement protocol. The synchronization message $s_{1,3}$ is sent only after all proposals are received.

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_3 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 Figure 4).

When a process p_j ($\forall j \in T$) falling into T receives a request, it immediately ceases outgoing memory communications to T's channel (further requests will be buffered) and 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 T.

At first glance, the protocol described above looks correct. However, if used, it would cause coherence problems under certain circumstances. As illustrated in Figure 4 after the receipt of the message $s_{1,3}$, P_3's cache differs from those of P_1 and P_2, because the leader P_1 sent the synchronization message before receiving P_2's update.

The solution requires that the coordinator send the synchronization message only when all proposals are received (message $s_{1,3}$ in Figure 5). Since processes stop outgoing memory communications as soon as 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 to the topic T, and finally publishes a message ending the protocol, marked as r_3. When processes receive an r message from the channel T, they recommence outgoing memory communications to T.

Finally, if a process is no longer interested in a topic T, a protocol allowing that node to leave T is used. As this algorithm is very similar to the previous one, its details have been omitted for the sake of brevity. The main difference with the leaving protocol however is that the number of available nodes is decremented.

This proposed scheme does have one serious shortcoming however; since the coordinator process must wait until all proposals are received, it has to know how many processes belong to T. Hence, due to the findings of the FLP theorem [10], this protocol cannot cope with crashes (this may be solved however with group membership protocols like JGroups [5] or Project Shoal from Glassfish [1]).

3.4 Memory Updates

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

Within synchronization messages (see figure 5), 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.

During updates matters become more complex; the proposed scheme exploits the observation that when operations are commutative they can be reordered without affecting the final state [20, 30]. If the received message contains an "entire" object then the content is stored in 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 using *reflection* techniques.

In the case of updates, the DSM low-level behavior differs according to the memory consistency model in operation. If the sequential consistency model is used, then the update is sent over the communication channel and the local memory is updated only when the message is received (a topic subscription could be required). The PRAM model requires instead that the new value be written to the local memory before updating remote copies; in order to avoid coherence problems, the following actions are required: The old value is saved so that in the case of communication problems it can be restored; The node which updates the shared memory cannot receive its own message (as is the case in the sequential model), otherwise the memory would be updated twice for each `write` invocation.

4 Experimental Results

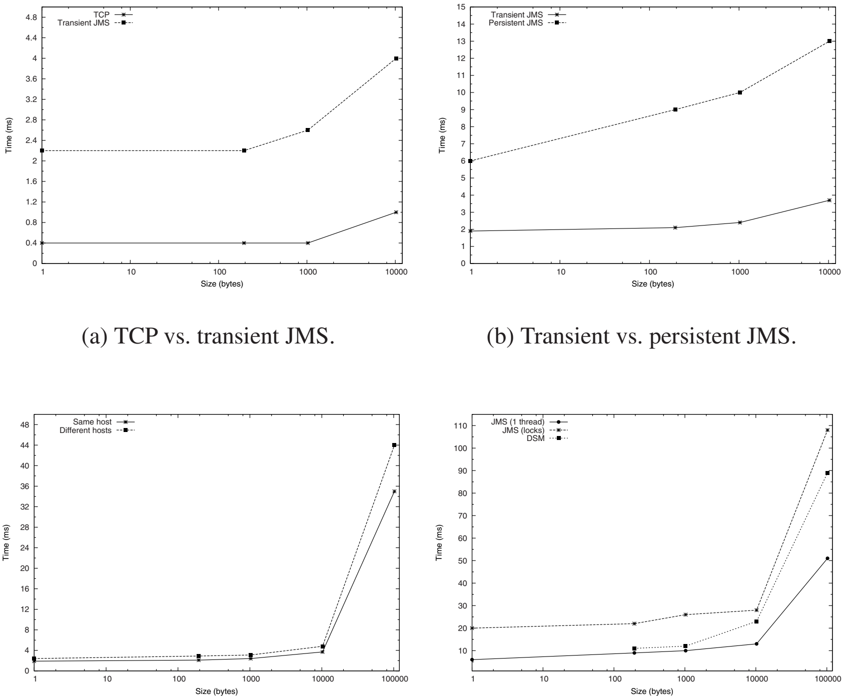
A number of performance tests were conducted on our prototype architecture. This section presents the preliminary results of those tests.

Our testbed environment consisted of a cluster composing Pentium 4 PCs running Linux 2.6.10 and JVM Sun 1.5.0_02 and connected by a 100 Mbit Fast Ethernet LAN. The JMS provider we used was Joram 4.3.1 [21] 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 discussing the results of the tests, it must be stressed that they were carried out on a shared network with shared servers. Therefore the occurrence of unpredictable delays due to unrelated network traffic was a possibility, however this did present the opportunity to test the DSM system performance in a realistic environment.

A "benchmark" application suite was developed to determine the overhead of components, the agreement protocol cost and the cost of updates. 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 incurred by each component. 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.



(a) TCP vs. transient JMS.

(b) Transient vs. persistent JMS.

(c) Local vs. remote connections.

(d) Persistent JMS vs. DSM.

Fig. 6. Components cost

The overhead was calculated as round-trip time depending on the message size. The results show the mean values of 1000 measurements.

TCP vs. JMS. Figure 6(a) shows the overhead introduced by JMS. While TCP provides *at-most-once* semantics 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 approximately the same in absolute terms and thus its impact decreases as the message size increases. JMS is 11 times more expensive than TCP when a single byte is sent and four times more expensive when the message is 10 KB large.

An additional point of interest regarding the difference between 1 KB and 10 KB messages is that since the network MTU is 1500 bytes, the first message is contained within one IP packet while the second is split into seven fragments. In these conditions the cost of a TCP send operation grows by a factor of 2.5 while the JMS grows by only 1.5.

Transient vs. Persistent mode. This test measured the overhead needed to guarantee an *exactly-once* delivery semantic. When 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: Figure 6(b) shows that the overhead is indeed quite noticeable; the persistent mode is approximately four times more expensive than the transient, with an increasing trend.

Transparency cost. The aim of this third experiment was to determine the outcome of JMS clients connecting to a server, then sending messages to a topic deployed on another server (usually JMS clients create only one connection). This test was carried out by running a distributed JMS configuration composed of three nodes configured in transient mode. Figure 6(c) shows that the difference is approximately 26%, with an increasing trend; as shown in Figure 7. This difference explains the high overhead introduced by the run-time 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 this happens by message exchange. This is price for providing location transparency, and is related to the naming service (JNDI).

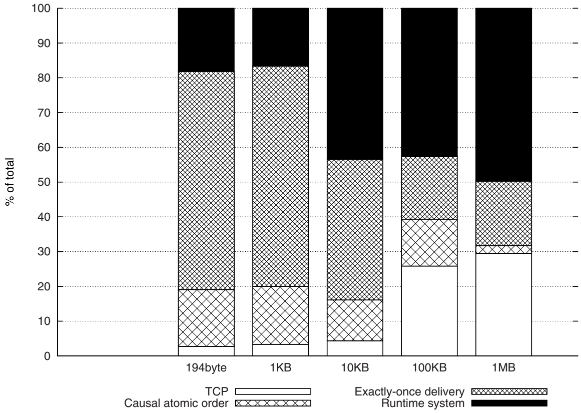


Fig. 7. Components overhead for different message sizes

JMS vs. DSM The last experiment in this set of tests measures the overhead introduced by the run-time system. The communication channel is provided by three Joram nodes, configured in persistent mode. The results shown in Fig. 6(d) include the following scenarios: (i) a JMS client using a single thread to receive and publish messages, (ii) a multi-threaded client using one thread as publisher and another thread as subscriber, and (iii) a client using the facilities provided by the run-time 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 for its receipt and thus the single-thread version performs better; unfortunately in a real scenario the two operations are handled individually. The dual-threaded application can be modelled in the form of a “producer-consumer” with a bounded buffer of one-item capacity; the overhead needed to synchronize the two threads is very high (111% for 100 KB messages). Finally the DSM guarantees the coherence of replicated data as well as a synchronization mechanism. Results show that the run-time system performs well only when messages are sufficiently small (1 KB).

4.2 Agreement Protocol

The second test evaluates the cost of shared memory synchronization. Assuming a system composed of n nodes in which k form part of the sub-set ($k <= n$), all nodes receive $k+2$ messages and publish only one message (except for the coordinator, which sends two messages). These experiments were repeated five times. Average results, as depicted in Figure 8, show that external factors are much more important than the number of nodes involved in the election. As already mentioned, both network and computation resources were shared during tests, explaining 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.

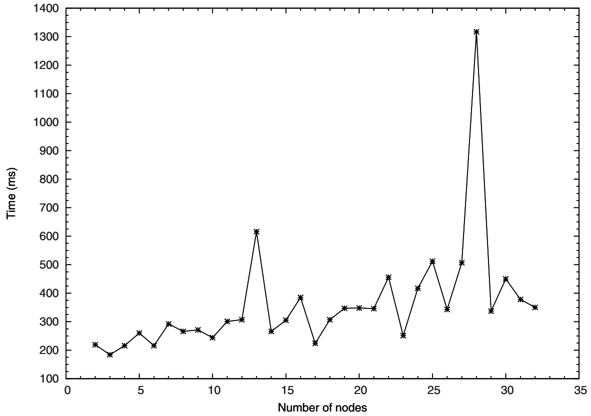


Fig. 8. Cost of memory synchronizations for different network sizes

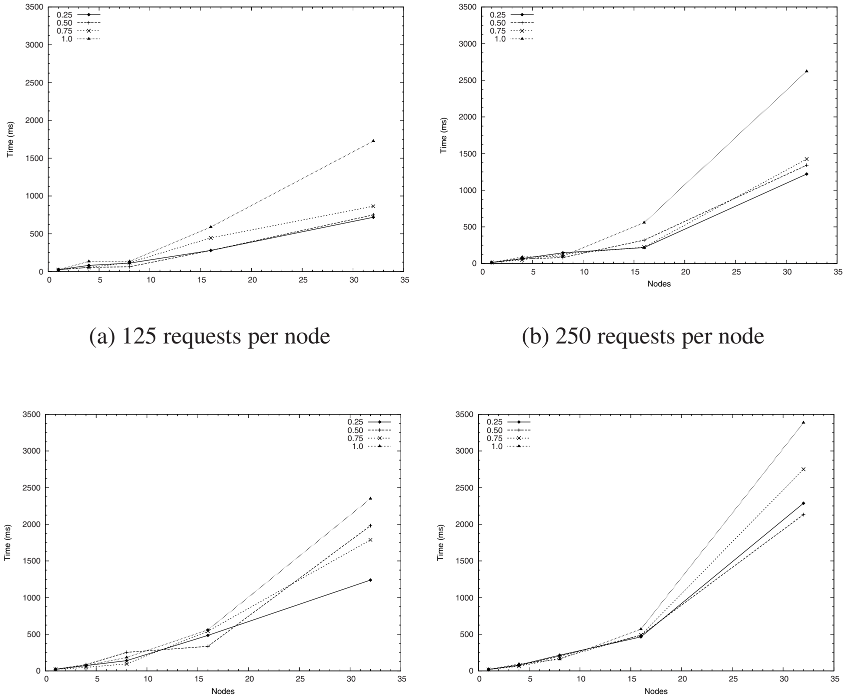
4.3 Updates

This section shows the update test results related to the sequential consistency model for objects of 194 bytes. The performance of the PRAM is not shown because local memory is updated with no communication overhead.

The JMS provider is composed of 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. This set of experiments aims to answer to the following questions:

1. What is the effect of varying the number of nodes (1, 4, 8, 16 and 32)?
2. How does the system react to increasing the amount of requests per node from 125 to 500, using increments of 125?
3. How does the system behave with respect to the system configuration (25%, 50%, 75% and 100% of nodes falling into the same sub-unit)?

Average cost depending on the number of nodes. The results of this test were not particularly significant given the rapidness of nodes' attempts to update shared memory. Executing this test presents problems as it interferes with the provider's ability to



(a) 125 requests per node

(b) 250 requests per node

(c) 375 requests per node

(d) 500 requests per node

Fig. 9. Updates

handle requests; the significant difference between the values shown in Figure 9 is due to the way the topic abstraction is implemented, and thus adding more machines to the provider only partially mitigates the issue.

Average cost depending on the number of requests. The results demonstrated that when the number of clients is four, there is no substantial difference. If the number of nodes continues growing however, then the provider becomes inundated with requests and consequentially is no longer able to satisfy those requests expeditiously. With regard to observed spurious values: when the number of nodes is high they can be attributed to external factors, but if the number 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 which features all nodes belonging to the same subunit, consistently performs the worst due to its semantic being equivalent to broadcast communications. The configuration which features 50% of all nodes belonging to the same subsystem performs relatively well. The other two tests were carried out concurrently and thus, since the JMS provider knows the number of both publishers and subscribers it can allocate more resources where they are needed.

5 Related Work

In this section we compare our DSM middleware system with other DSM systems featuring a design and approach comparative to ours.

Previous systems typically do not address problems related to the communication channel. In particular, several implementations assume that packets are delivered in the correct order while messages cannot be lost; TreadMarks [2] and IVY [15], for instance, use a combination of UDP and timeouts to maintain coherence among replicas. While this shortcoming is not a significant hindrance in LAN environments (where message latency is much lower compared to the Internet), as a design choice this is particularly critical since we don't make any assumption about the communication channel.

The most popular DSM systems have been built with a predefined operating system/hardware architecture combination (*e.g.*, TreadMarks, IVY, Brazos [26]) or relying on modified compilers (*e.g.*, Munin [7], Shasta [22], Jackal [29]), including approaches based on the Java language with modifications required within the JVM (*e.g.*, Java/DSM [31]).

Maintaining memory coherence in our DSM middleware bears similarities to the solution adopted by TreadMarks, there are several differences however. For example, at the implementation level, our DSM system is object-based while TreadMarks is page-based. Consequentially, TreadMarks may be affected by false sharing and fragmentation problems which require additional protocols [3] in order to be minimized. 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. For example, TreadMarks propagates updates to all nodes while our solution "hits" interested nodes only. JDSM [25] uses a centralized node (the Cluster Manager) to intercept requests, creating a bottleneck as the number of nodes increases.

The use of group communication has precedents. Orca [4] uses group communications to implement a sequentially consistent object-based DSM. Orca, like the system described in this paper, relies on features of the communication channel (reliable, total ordered broadcast) to implement transparent replication as well as to maintain data-object consistency. Brazos uses two main system threads to reduce communication overhead and uses a selective multicast to reduce communication traffic. Finally, the work described in [24] presents an object-based DSM, designed as an extension to the .NET Framework. It provides a causally consistent memory model where causal relationship is achieved through vector logical clocks sent with every message.

Java has been used previously to implement a DSM system. Java/DSM [31] addresses problems that arise when a heterogeneous set of nodes are used, however it modifies the memory management of the JVM and thus it is not portable. As in our system, JDSM does not 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 of a Cluster Manager, Server and Client) requires that requests be sent from the Client to the Cluster Manager and then forwarded to one of the available Servers. However, over-involvement of the Cluster Manager in this scenario produces a potential bottleneck, hindering scalability. Finally, node interactivity can use three different communication protocols, TCP/IP socket, PM and VIA. Again, the lack of MOM in a middleware environment leaves DSM middleware without services that can be incorporate at little cost (*e.g.*, transactions).

Java Past Set (JPS) [19] differs from our system primarily because shared objects are not replicated, hence for reasons explained in section 2.3, scalability problems arise when the number of accessing clients increase. Data location is achieved with *element object descriptors* and can be distributed according to several policies (*i.e.* place the object with 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. A commonality between JPS and our DSM is the data update phase, defined as *user redefinable memory semantics*, allowing the application programmer to define the memory operation semantic.

Finally, the only known system designed to host multi-player games is Plurix [23], providing the illusion of a shared memory by utilising a custom operating system and Java compiler; the resulting architecture works exclusively on Intel platforms while the memory coherence unit is a page of virtual memory.

6 Conclusions and Future Work

In this paper we have presented distributed shared memory middleware. In comparison to existing approaches, this platform provides greater transparency to the application programmer. Firstly, it has been identified that the proposed protocol does not prerequisite any hardware or software architectural assumptions (the features of Java handle the differences). Secondly, the run-time system transparently handles the message passing details. Thirdly, this system is able to use non-reliable channels, where packets can be lost and delays are of an arbitrary length.

To summarize, we have shown the applicability of group communications, replication, caching and interest management techniques to support the construction of event-oriented distributed systems. The next step is to conduct a more extensive plan of tests. In particular, we plan to deploy our middleware solution in a WAN environment such as PlanetLab or the Grid, where packets can be lost or delayed.

There remain problems to address: client connection to the JMS provider for example, which is responsible for the poor performance observed in certain circumstances, or issues of fault tolerance during memory synchronization. Differing approaches to the latter problem have been proposed, however some techniques (*e.g*, [6]) are not scalable while others (*e.g*, [27]) allow only one crash occurrence. A possible solution requires more realistic assumptions about the communication channel to solve the consensus problem (or alternatively to use a randomized algorithm). If it is possible to solve consensus, it becomes feasible to adopt an approach similar to that presented in [18] to periodically save the memory content in stable storage, while the use of failure detectors would inform the run-time when the memory has to be restored (our system uses a write-update algorithm).

References

[1] Abdelaziz, M., et al.: Project Shoal, a dynamic Java clustering framework, `https://shoal.dev.java.net`

[2] Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer 29(2), 18–28 (1996)

[3] Amza, C., Cox, A.L., Dwarkadas, S., Jin, L.-J., Rajamani, K., Zwaenepoel, W.: Adaptive Protocols for Software Distributed Shared Memory. Proceedings of the IEEE, Special Issue on Distributed Shared Memory Systems 87(3), 467–475 (1999)

[4] Bal, H.E.: Orca: A Language for Distributed Programming. ACM SIGPLAN Notices 25(5), 17–24 (1990)

[5] Ban, B., et al.: JGroups, a toolkit for reliable multicast communication, `http://www.jgroups.org`

[6] Cabillic, G., Muller, G., Puaut, I.: The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In: Proceedings of the 14th IEEE International Symposium on Reliable Distributed Systems (SRDS 1995), September 1995, pp. 96–105 (1995)

[7] Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Implementation and Performance of Munin. In: Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP 1991), pp. 152–164. ACM Press, New York (1991)

[8] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The Many Faces of Publish/Subscribe. ACM Computing Surveys (CSUR) 35(2), 114–131 (2003)

[9] Fekete, A., Kaashoek, M.F., Lynch, N.: Implementing Sequentially Consistent Shared Objects using Broadcast and Point-To-Point Communication. Journal of the ACM (JACM) 45(1), 35–69 (1998)

[10] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM (JACM) 32(2), 374–382 (1985)

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1995)

[12] Gharachorloo, K., Lenosk, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In: Proceedings 17th Annual International Symposium on Computer Architecture, pp. 15–26. IEEE Computer Society, Los Alamitos (1990)

[13] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558–565 (1978)

[14] Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers C-28(9), 690–691 (1979)

[15] Li, K., Hudak, P.: Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems 7(4), 321–359 (1989)

[16] Lipton, R.J., Sandberg, J.S.: PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Dept. of Computer Science, Princeton University (September 1988)

[17] Menasce, D.A.: MOM vs. RPC: Communication Models for Distributed Applications. IEEE Internet Computing 9(2), 90–93 (2005)

[18] Morin, C., Kermarrec, A.-M., Banatre, M., Gefflaut, A.: An Efficient and Scalable Approach for Implementing Fault-Tolerant DSM Architectures. IEEE Transactions on Computers 49(5), 414–430 (2000), `citeseer.ist.psu.edu/morin97efficient.html`

[19] Pedersen, K.S., Vinter, B.: Java PastSet: A Structured Distributed Shared Memory System. IEEE Proceedings – Software 150(2), 147–153 (2003)

[20] Pu, C., Leff, A.: Replica Control in Distributed Systems: An Asynchronous Approach. In: Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD 1991), pp. 377–386. ACM Press, New York (1991)

[21] ScalAgent Distributed Technologies. JORAM: Java Open Reliable Asynchronous Messaging (2005), http://joram.objectweb.org

[22] Scales, D.J., Gharachorloo, K., 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), pp. 174–185. ACM Press, New York (1996)

[23] Schöttner, M., Wende, M., Göckelmann, R., Bindhammer, T., Schmid, U., 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 2003), Tokyo, Japan, May 2003, pp. 502–509. IEEE Computer Society, Los Alamitos (2003), http://www.plurix.de/

[24] 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 2003), Tokyo, Japan, pp. 457–462. IEEE Computer Society, Los Alamitos (2003)

[25] Sohda, Y., Nakada, H., Matsuoka, S.: Implementation of a Portable Software DSM in Java. In: Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI 2001), pp. 163–172. ACM Press, New York (2001)

[26] Speight, E., Bennett, J.K.: Brazos: A Third Generation DSM System. In: Proceedings of the First Usenix Windows NT Symposium, August 1997, pp. 95–106 (1997), http://www-brazos.rice.edu/brazos

[27] Sultan, F., Iftode, L., Nguyen, T.: Scalable Fault-Tolerant Distributed Shared Memory. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (Supercomputing 2000), pp. 20–32. IEEE Computer Society, Los Alamitos (2000)

[28] Sun. Java Message Service. Sun Microystems, Version 1.1 (April 2002)

[29] Veldema, R., Hofman, R.F.H., Bhoedjang, R.A.F., Bal, H.E.: Runtime Optimizations for a Java DSM Implementation. In: JGI 2001: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, pp. 153–162. ACM Press, New York (2001)

[30] Wuu, G.T., 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 1984), pp. 233–242. ACM Press, New York (1984)

[31] Yu, W., Cox, A.L.: Java/DSM: A Platform for Heterogeneous Computing. Concurrency - Practice and Experience 9(11), 1213–1224 (1997)