The Design and Evaluation of a Middleware Library for Distribution of Language Entities

Erik Klintskog[1], Zacharias El Banna[2], Per Brand[1], and Seif Haridi[2]

[1] Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, Sweden, http://www.sics.se/

[2] IMIT-Royal Institute of Technology, Electrum 229 * 164 40 Kista, Sweden, http://www.it.kth.se/

Abstract. The paper presents a modular design of a distribution middleware that supports the wide variety of entities that exist in high level languages. Such entities are classified into mutables, immutables and transients. The design is factorized in order to allow multiple consistency protocols for the same entity type, and multiple coordination strategies for implementing the protocols that differ in their failure behavior. The design is implemented and evaluated. It shows a very competitive performance.

1 Introduction

We present the design and implementation of a middleware library, the DSS (Distribution SubSystem). The DSS is designed to simplify the implementation of distributed programming systems. Using the DSS we can add distributed programming facilities to programming systems/languages which are normally not distributed. We claim that the effort needed to create a distributed programming system(DPS) using the DSS to handle distribution involves much less programming effort than to explicitly program the necessary distribution support in the system itself.

Our system aims at completeness; by this we mean that different paradigms of distributed computing can easily be implemented using the DSS. By completeness we include both functional (e.g. extending objects to distributed objects with preserved semantics) as well as non-functional aspects (e.g. providing the most efficient distribution support for all patterns of use of distributed objects).

1.1 Background

Middleware support for programming language distribution – be it partial or total – can conceptually be divided into two categories: programming language dependent middleware and programming language independent middleware. Independent middleware commonly targets language interoperability and offers only limited distribution support, e.g. CORBA [1]. Language dependent middleware can potentially offer complete distribution support. However the design

and implementation is time-consuming. In practice, the amount of actual distribution support in a distributed programming system (DPS) reflects the trade-off between desired completeness and the amount of work required to realize it.

The trade-off is, we believe, reflected in the fact that most DPSs are incomplete even as regards functional aspects, in that distribution does not preserve desirable properties of the language semantics [2, 3, 4, 5, 6]. Few systems are functionally complete [7, 8, 9], but none, to our knowledge, is complete as regards efficiency aspects.

1.2 Motivation

This work is motivated by the need of a language independent middleware that provides full distribution support for arbitrary high-level programming system (PS). By full support we mean that all language entities should potentially be sharable in a distributed computing environment, with preserved semantics. Distribution on the level of language entities means that threads residing in different processes can share entities as if they were residing in the same process[3].

Examples of language entities are first class data structures such as objects, primitive data types or channel abstractions. Note that we exclude unsafe data types such as C pointers. On the other hand, code can be shared (e.g. procedure values in Oz and classes in Java). Furthermore some data types will be shared with limited distribution behavior; files, for instance, could be shared as stationary objects which only allow remote access.

Sequential consistency is generally a requirement[10] to preserve the semantics of many language entities. A prototypical example is the semantics of objects in OO languages. However, this does not preclude the use of a weaker consistency model to improve the performance of a distributed application [11], but from our point-of-view this should be reflected in a different type of language entity (e.g. different type of object).

1.3 Contributions

The major contributions of this paper can be summarized as follows. Firstly, for the developer of a DPS, we provide a model of distribution support for language entities, based on the type of distribution support a given entity requires. The model is general enough to support all to us known language entities, found in almost all high-level programming languages/systems (e.g. Java, C# and Oz[12]).

Secondly, for the application developer, we provide a model of distribution that guarantees functional properties (i.e. preserving consistency) for a given distributed entity. This model also allows for fine-grained control of non-functional aspects. Assignment of entity consistency protocols can be done in runtime, based on expected pattern of use per entity instance, and not on the entity type.

Thirdly, we describe a novel component-based design of entity consistency protocols. The model simplifies implementation of new protocols, increases code

[3] This means location transparency modulo failure and latency

reuse, and enables fine-grained customization of entity consistency protocols from the DPS level.

Finally, we present the implementation and evaluation of our language independent middleware library, called the Distribution SubSystem[4] (DSS). The DSS efficiently implements the above-described contributions, as shown by our evaluation.

1.4 Paper Organization

The rest of the paper is organized as follows. In Sect. 2 the language independent entity model is described. Sect. 3 describes our novel structure of entity consistency protocols. The structure and actual implementation of our middleware is briefly discussed in Sect. 4. More attention is given to the performance of our implementation as shown in Sect. 5. The design is compared to similar systems in Sect. 6 and a conclusion is given in Sect. 7. Note that this paper focuses on the design of key concepts and evaluation of our middleware library, and not on the philosophy behind the design and practical issues such as how a PS can be coupled to the library. Those issues are explained in detail in [13].

2 An Abstract Model of Language Entities

The set of language entities found in most high-level programming languages is large. These entities are from a programming point of view semantically different, even though they might have the same name. However, from the distribution point of view, those differences can, to a large degree, be abstracted out and we are left with surprisingly few abstract entity types. The proposed model provides distribution on the level of abstract entities.

2.1 The Abstract Entity

Our shared entity model uses the notion of a *local entity instance*, acting as the local representative for a shared entity. A local entity instance is present at every process holding a reference to the shared entity. All instances are inherently equal, none is more privileged, i.e. there is no a priori centralized control. Each instance is connected to an *abstract entity instance*, coordinating operations performed by threads on the local instances.

When a local entity instance becomes shared, it may not be accessed directly anymore; operations must be directed to its abstract entity instance. All interaction with an abstract entity instance is done using *abstract operations*[5], expressing manipulations of the shared entity. An entity operation is translated into an abstract operation, expressing a corresponding semantic type of manipulation. The result of an abstract operation tells the calling thread how to

[4] Available for download at http://dss.sics.se

[5] Analogous to the distinction between abstract and concrete language entities there are potentially many concrete operations per abstract operation.

proceed: perform the operation on the local instance, continue with the next instruction or wait for a later decision.

At any point in time a local entity instance is either *complete*, i.e. it has a representation that allows for local execution of operations, or *skeleton*, i.e. it merely acts as a proxy. The status is explicitly controlled by the abstract entity instance.

Entity types that are to be distributed must be matched with a suitable abstract entity type. The matching is based on the centralized semantics of the entity type. Different *abstract entity types* capture different functional needs and guarantee consistency according to a consistency model (e.g. sequential consistency). An abstract entity instance actually provides a single interface to a set of entity consistency protocols with the same functional properties. In order to support distribution, at least one entity consistency protocol is needed per abstract entity type. However, to efficiently capture non-functional requirements, multiple entity consistency protocols are required. A non-functional requirement might be maximum number of hops, bandwidth utilization, or resilience to failures.

2.2 Different Types of Abstract Entities

We have currently identified three meaningful abstract entity types, all guaranteeing sequential consistency.

Mutable. This type has two abstract operations. *Update* indicates that the state is to be altered while *access* means to read. The *mutable* is preferably used by language entities that allows for destructive updates, e.g. objects. Suitable protocols for this type are: remote-execution, mobile state[14], and read/write invalidation.

Immutable. The immutable state of an entity is at some point replicated to a processes referring it. It can then be accessed through *access*. This means that all entity instances eventually become complete and no synchronization is then needed. Protocols for the *immutable* are eager-, lazy- and immediate replication.

Transient. This type has two abstract operations: *access* and *bind*. *Bind* terminates the coordination of the entity, thus removing all abstract entity instances. *Access* suspends the caller until a bind operation has been performed. The *transient* is preferably used for languages entities such as logical variables in Oz[15], and futures in Multi lisp[16].

Not all language entities guarantee sequential consistency in the centralized case. Oz ports and Erlang channels are examples of such entities. Also, asynchronous remote method invocation[17] is a popular optimization in distributed object systems. To efficiently support distribution of this class of entities, we provide two abstract entity types that guarantee at most PRAM (or FIFO) [18] consistency, called *Relaxed Mutable* and *Relaxed Transient*.

2.3 Interacting with an Abstract Entity

A language entity interacts with its abstract entity using abstract operations. In order to resolve operations an abstract entity needs to interact with its entity instance, using four entity-instance callbacks:

retrieveState The callback returns a state description of the entity instance, i.e. a description that can change any entity instance's status from skeleton to complete. Clearly this is only legitimate if the entity instance from which the description is retrieved is complete.

installState Install a state description to the entity instance, making it complete.

executeOperation The callback is given a description of a concrete entity operation to execute on the local entity instance.

resumeThread A thread previously suspended on an abstract operation is resumed. The thread is told to either redo the operation or continue with the next instruction.

The described interaction framework[6] is complete in the sense that either a state- or an operation-transporting protocol can be used transparently. A state transporting protocol allows for local access for the entity instances by moving a state description to the executing process. In contrast, an operation transporting protocol moves an operation description to a process(es) hosting a complete instance to execute it there.

A shared object uses the mutable abstract entity. Depending on the chosen type of protocol, different events can be observed for the same abstract operation. Bellow are two examples where an object is distributed using either a state or an operation transporting protocol. The sequence diagrams show the respective events for the same method invocation on the shared entity. Note that in both cases the initiating entity instance has skeleton status.

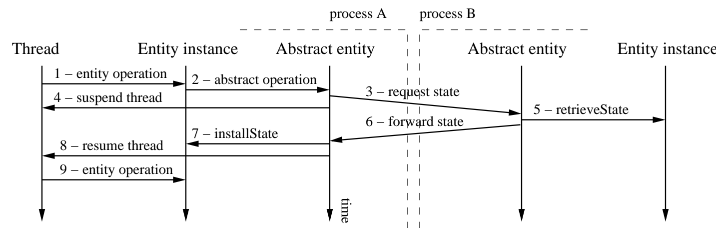

Fig. 1. Sequence diagram of the state transporting protocol, e.g. mobile state protocol.

[6] Due to space limitations the interfaces are described on a conceptual level. Concrete API descriptions and code examples can be found in [13].

Example: State Transporting Protocol Fig. 1 depicts the sequence of events that occurs when a thread at process A performs a method invocation (1) on a shared object, whose state is located at process B. The method invocation is translated into an abstract operation and passed on (2) to the abstract entity. A request for a state description is sent[7] (3) to the abstract entity located at process B. Simultaneously the thread is told to suspend itself (4). The abstract entity at process B receives the state request (5) and uses the callback **retrieveState** to get a state description. The description is passed back (6) to the abstract entity at process A, where it is installed (7) using the **installState** callback. Finally the suspended thread is resumed (8), using **resumeThread**, and told to redo the operation (9).

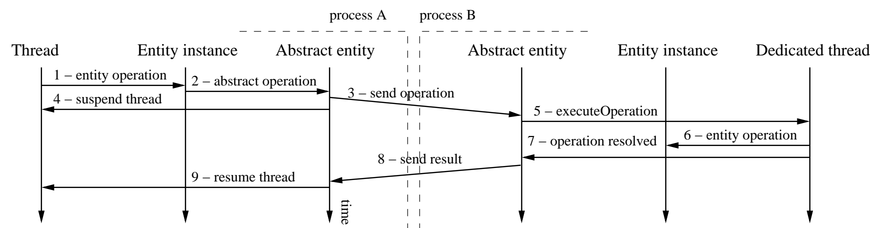

Fig. 2. Sequence diagram of the operation transporting protocol, e.g. remote execution protocol.

Example: Operation Transporting Protocol Fig. 2 depicts the sequence of events that occurs when a thread at process A performs a method invocation (1) on a shared object, whose state is located at process B. The method invocation is translated into an abstract operation and passed on (2) to the abstract entity. A description of the operation is sent (3) to process B. Simultaneously the thread is told to suspend itself (4). The abstract entity at process B receives (5) the operation description and uses the callback **executeOperation** to perform the operation locally. The operation is executed by a dedicated thread[8] (6) and the result is returned to the abstract entity (7). The result is passed back to the abstract entity at process B (8), that passes the result to the suspended thread and resumes it (9).

2.4 Classifying Language Entities into Abstract Entities

Assigning abstract entities to language entities can preserve the structuring imposed by the language, but is not required to. Multiple language entities can

[7] Via the coordination network, to be described later.
[8] A thread created solely for the purpose of remote execution.

be distributed together as one abstract entity, and one language entity can be decomposed into multiple abstract entities. Note that regardless of the chosen structuring, an abstract entity must capture the semantics for the structure in the centralized case, e.g. an entity that allows for destructive updates should not use the *immutable* type.

For example, an array can be treated as one (composed) *mutable* entity or it can be decomposed into an *immutable* structure referring *mutable* cells. When distributed, the array structure will be replicated and the array structure in turn will refer mutable cell instances. Thus when updating an array element only one cell will be affected.

3 Coordinating Entity Instances

A *coordination proxy* is present at each process hosting a local entity instance. The coordination proxy is connected to its abstract entity (as depicted at the left of Fig. 3).

An entity consistency protocol executes over the dynamic sub-network, called the *coordination network*[9], formed by participating coordination proxies. A coordination network has a hub, called the *coordinator*. A property of the network is that proxies can always contact the coordinator, while the vice verca is not necessarily true. For the remote execution protocol, a proxy would send the operations to the coordinator, that in turn passes them to the proxy where the state is located.

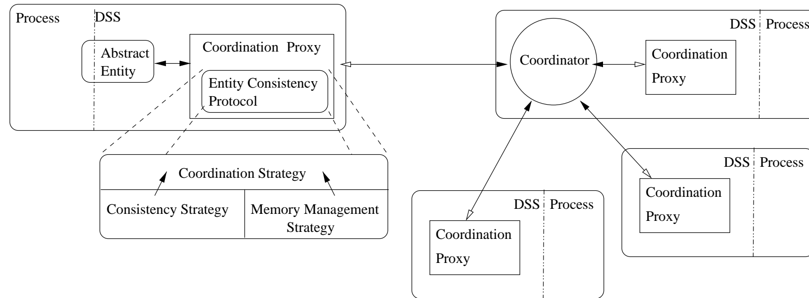

Fig. 3. The coordination network and the per-process coordination proxy. The abstract entity is coupled to a coordinating proxy connecting it to the coordination network. Bellow to the left is the expanded framework of the entity consistency protocol. Note that in this example the coordinator is located at one of the processes hosting a coordination proxy

[9] To minimize this subnet, at-most-one coordination proxy is allowed per process.

3.1 Three Dimensions of Entity Consistency

The entity consistency protocol is realized as a framework, as shown in the expansion of the Coordination Proxy in Fig. 3. This divides the strategy over entity consistency into three separate sub-strategies. Firstly, the *memory management strategy* detects when a shared entity is no longer needed, i.e. the number of proxies reaches one. Secondly, addressing within the coordination network is realized by a *coordination strategy*, also providing a messaging service for the other two modules. Finally the *consistency strategy* upholds the entity semantics, controlling entity instances connected to the coordination network and threads performing operations. The three strategies are implemented as three protocols running in parallel over the coordination network.

Each strategy is implemented as a module with a well-specified interface, one interface per sub-strategy type. An optimal entity consistency protocol for a particular entity and usage pattern is just a matter of composition. This potentially increases code reuse (in the form of reused sub-strategies) and simplifies development of entity consistency protocols.

Coordination Strategy The coordination strategy defines how the messaging infrastructure and services are realized. These messaging services are then used by the consistency- and the memory management strategy. This includes defining the location and behavior of the coordinator and providing routines for inter coordination-network communication. Examples of coordination strategies are: a stationary coordinator, a mobile coordinator and replicated coordinators.

Consistency Strategy The protocol resolves abstract operations for the abstract entity. A consistency strategy is divided into two parts: one *end-point unit*, present at each coordination proxy, and one *arbitrator*, located at the coordinator(s).

Interaction with local entity instances together with communication and addressing services, provided by the coordination strategy, simplifies implementation of a wide range of protocols:

Remote-execution Every end-point sends all operations to the arbitrator that sends them on to one selected end-point unit, hosting the complete local entity instance. A synchronous version of the protocol is available to the mutable abstract entity, where the write operation is acknowledged (possibly with a result). An asynchronous version exists for the relaxed mutable abstract entity.

Mobile state The entity's state is moved between local entity instances. Regardless of the type of operation, an end-point requests the state from the arbitrator and waits until it arrives. Several consecutive writes and reads can then be performed locally.

Read/write-invalidation A protocol that allows for exclusive update or concurrent access to a local state. Two versions of the protocol exist. The *eager*

protocol records the readers and automatically updates them when the state has changed. The *lazy* protocol requires all readers to actively request read permission after an invalidation.

Pilgrim A mobile state protocol inspired by the work in [19]. This protocol is optimized for the case when a small set of proxies reads and writes frequently.

Replication This class of protocols is used by immutables. For lazy replication, the state is retrieved when an entity instance first tries to access the state. For eager replication it is requested immediately after the creation of the coordination proxy. Immediate replication transports the state with every reference to an entity, but duplicates are avoided due to the at-most-once property of coordination proxies.

Once only A class of protocols used to realize transient behavior, inspired in their design by [15].

Memory Management Strategy Properly packaged distributed garbage collection algorithms [18] detect when the number of coordination proxies reaches one. When this occurs the last entity instance can be localized, hence dismantling the coordinator and freeing resources. Of course by the time when localization is achieved there may be no references left, which usually will be handed by the memory management outside the DSS.

Similar to the consistency strategy, the memory management strategy is divided into end-point units and one *detector* (located at the coordinator). The existence of an end-point at a coordination proxy guarantees that the proxy is accounted for by the detector. Using this framework the DSS implements: fractional weighted reference counting[20] , reference listing[21], time lease and persistent entities.

4 Middleware Design and Implementation

To simplify the coupling of a PS to the DSS, the DSS is internally divided in two subcomponents: the Advanced Asynchronous Protocol Machine(AAPM), and the Communication Service Component (CSC). As depicted in Fig. 4 the AAPM implements the abstract entities, protocols of the coordination network and a high level messaging service. The messaging service is based on a notion of remote processes, *DSites*, providing reliable, in order, asynchronous messaging. The CSC is an interface for communication routines and networking tasks such as connection establishment, data transportation and failure detection for the messaging service. The purpose of the CSC is to abstract away the OS from the AAPM, as depicted in Fig. 4. The CSC is easily replaced to enable custom implementations based on application knowledge, e.g. specialized addressing schemes or failure detectors.

Our implementation of the DSS is a linkable C++ library and our CSC implementation is a fairly straight forward C++ implementation based upon TCP/IP. The DSS uses interface classes, *mediators*, for all complex data structures shared with the PS: entity references, entity states, operations and threads. Mediators

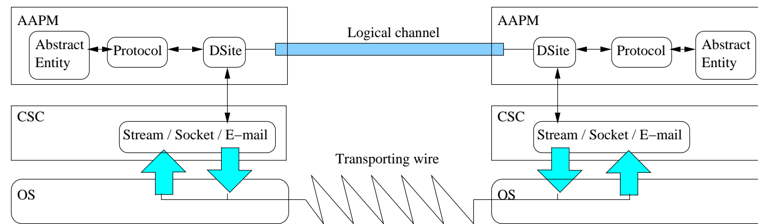

Fig. 4. The relation between a generic AAPM, a transporting CSC and the underlying communication system, e.g. the OS. The CSC is free to use whatever means desired to transport data.

are represented as C++ classes to simplify coupling. Marshaling of mediators is a cooperative activity involving both the DSS and the PS. Our model allows for late marshaling, i.e. messages are serialized when actually put onto the wire and not when inserted into the messaging service. The DSS knows how to serialize its internal data. When messages contain mediators, the programming system is asked to create a serialized representation.

5 Evaluation

In order to evaluate the performance of the DSS we have conducted three major tests. Firstly, a test that measures pure messaging speed. Secondly, a test evaluating the performance of the different consistency strategies, executed in a controlled environment. Finally, an impact evaluation of using different consistency strategies, in a real world application.

To evaluate the DSS we used four different applications/systems:

Socket-application. A small C++ socket application which measures the raw I/O cost for messaging on our cluster.

C++DSS. A thin C++ library on top of the DSS that allows for sharing of mutable and transient data structures. This is included to measure the raw cost of abstract operations.

Mozart. Development version 1.3.0 of the distributed programming system that implements the multi-paradigm language Oz. Included to measure the difference between the tightly integrated distribution layer in Mozart compared to a pure Oz virtual machine coupled to the DSS.

Oz-DSS. Development version 1.3.0 of the Mozart system with its internal distribution support replaced by the DSS. The Oz-DSS system is far more expressive when it comes to distribution than the original Mozart system and it clearly separates the local-execution engine from the distribution subsystem.

All applications were compiled with gcc 2.95.2 using standard optimizations. The tests were conducted on a cluster of AMD ATHLON XP 1900 workstations, equipped with 512 MB of memory, interconnected by a 100Mbit LAN. The workstations run standard Red Hat Linux version 7.3 without X windows.

5.1 Messaging Performance

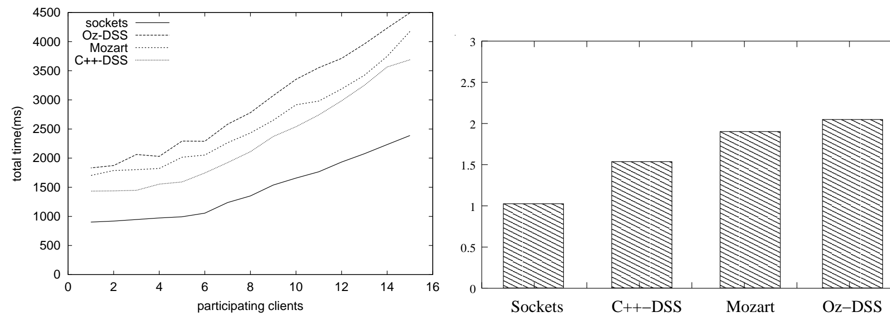

Fig. 5. The time to perform 10000 sequential remote requests for the four systems. The left graph shows the result when the number of nodes ranges from 1 to 15. The right graph show the result for one node, normalized against the socket application.

The test measured the time to perform 10000 sequential remote requests with one server and varying number (1-15) of simultaneously running client processes. The test was conducted with all four applications. The right diagram of Fig. 5 shows the result for the different applications running with one single client. The times are normalized to the socket application.

The C++-DSS application has only a 50% overhead compared to the raw socket program. This is a surprisingly small difference considering the differences in functionality. The socket program is extremely optimized for the test while the C++DSS is a generic distribution platform.

The tightly integrated distribution of Mozart gives 12% better performance over the Oz-DSS system. However, in the light of increased functionality and superior extendibility, this small difference is certainly acceptable.

The left diagram of Fig. 5 shows the total time to conduct the test when the number of clients increases. It is interesting to note that all DPSs increases the time proportionally to the socket application. This indicates, at least within the interval of 1 to 15 nodes, that the I/O capacity of the underlying operating system is the dominant factor when communicating with multiple nodes.

5.2 Protocol Evaluation

In this section, five entity consistency protocols for the mutable abstract entity are compared using the Oz-DSS. Factors like the number of participating processes, number of threads per process and the ratio between reads and writes were altered in order to show how the protocols performed under different usage patterns.

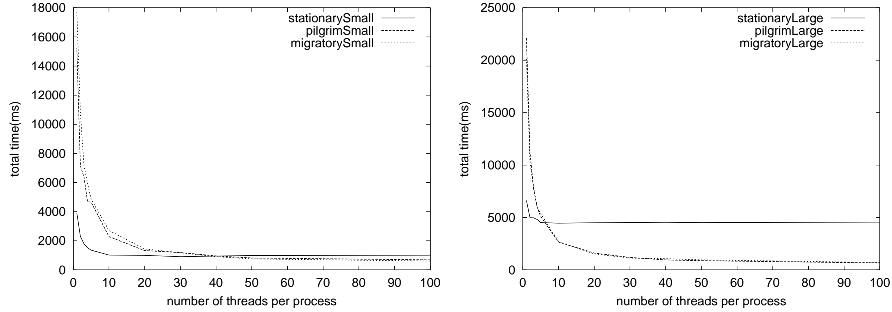

Fig. 6. The total time for 12 Oz-DSS processes to conduct 10000 accesses to a mutable state, using three different protocols, under different degree of concurrency. In the left graph the mutable state is a single integer value and in the right a list of 1000 integer values.

The Impact of Concurrency Each process performed 10000 accesses to a shared state under varying degree of concurrency; i.e. one thread doing 10000 iterations, two threads doing 5000 iterations, and up to 100 threads doing 100 iterations each. The test was conducted with 12 clients, using the mobile state, pilgrim and stationary protocols. The left diagram of Fig. 6 shows the plot of the total time for all clients to conduct all iterations against the number of threads per process. Note that increased concurrency has two effects. Firstly, latency will be masked which is notable in all tests. Secondly, concurrency will also batch pending operations when threads wait for the mobile state. This is observable; the state moving protocols advance from being outperformed to outperforming the stationary protocol. When the size of the state increases, from one single integer value to a list of 1000 integer values, the cost for I/O increases, depicted in the right diagram of Fig. 6. Since the state moving protocols communicate less when the degree of concurrency is increased, they perform better than the stationary protocol.

When considering the work load of the server, measured as the sum of user and system time used by the server process, a slightly different picture emerges, (see the left diagram of Fig. 7). To start with, the pilgrim protocol does not use the server at all. This is due to the long sequences of accesses of the state. Furthermore, the stationary protocol requires almost the same amount of resources as the mobile state protocol when the number of threads is small and is quickly outperformed when concurrency is increased. This shows the strength of the state moving protocols in batching work and low utilization of the server process. It also shows their weakness: higher access time when the number of competing processes for the state grows large.

Utilizing the Read/Write Ratio The two cache invalidation protocols, with lazy and eager updates, allows the state of a shared entity to be read in parallel, but updated at only one process at a time. This caters for substantial reduction

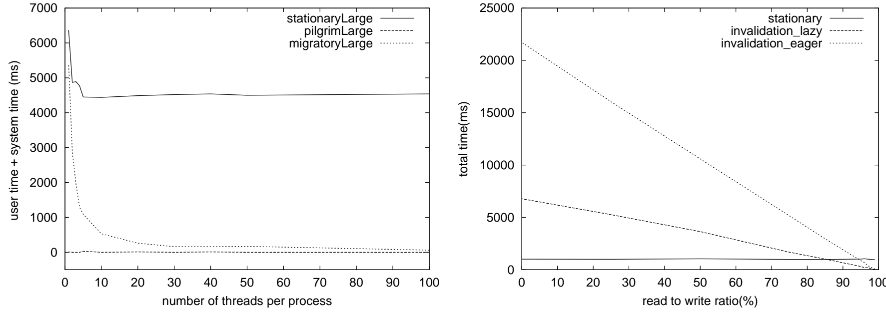

Fig. 7. Left: the total system and user time for the server process when 12 Oz-DSS processes does 10000 state accesses, using three different protocols, under different degrees of concurrency. Right: The total time for 15 Oz-DSS processes to perform sequential 10000 accesses, using three different protocols, under different read to write ratios.

in message traffic when the ratio of reads vs. writes is high. To validate this we conducted a test to see at which point our invalidation protocols are better than the simple stationary protocol. The test was conducted with 15 client processes, each performing 10000 sequential accesses to a shared state. The proportion of operations that were reads ranged from 0 to 99%, the resulting graph is shown in the right diagram of Fig. 7. The test shows that invalidation, with lazy updates, is overall superior to eager updating. The invalidation protocols impose a notable overhead when the access pattern has a low read to write ratio, but both protocols also improve in performance notably when the ratio gets high (above 85% for lazy and 95% for eager).

5.3 Distributing a Real Application

As a proof of concept, we took an already existing application developed for the Mozart platform, added the possibility to annotate the distribution behavior of single entities and tested it on our Oz-DSS system. The application, a distributed version of the snake game with self-learning actors, is interesting from a distribution point of view. Each actor, one per node(process), reads a section of a shared matrix and decides how to do a move, i.e. update an element in the matrix. The matrix is distributed on the level of single matrix elements.

The tests were conducted with the matrix elements distributed using different protocols(for mutables). The number of nodes was altered in order to show the scalability for different consistency strategies(i.e. protocol choice).

The ratio between reads to writes is high in the application. As shown in the left diagram of Fig. 8, the two invalidation protocols do very well, while both the migratory and stationary protocols simply do not scale.

By varying the size of the matrix, the interaction between the processes is implicitly varied. For a smaller matrix, the chance that two processes will read

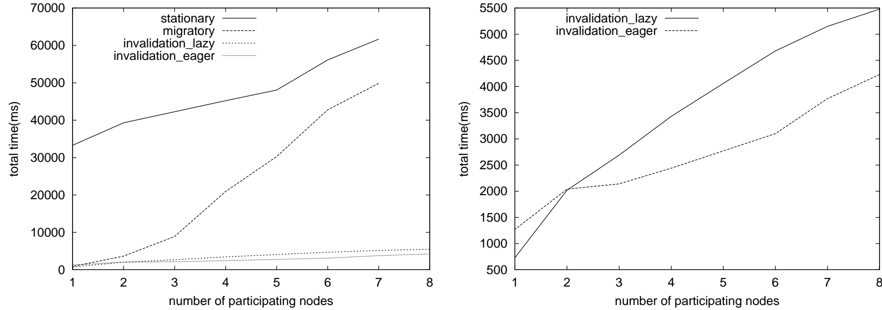

Fig. 8. Total time to run the distributed snake game for 1000 iterations, with varying number of clients. The left graph shows the result for all four protocols. The right graph shows the result for the two invalidation protocols.

the same element increases, and for a larger matrix it decreases. The consequences are depicted in the right diagram of Fig. 8; the eager invalidation protocol performs better than the lazy protocol on a small matrix and vice versa. It is beneficial to distribute information on element update immediately and not wait until it is asked for, if the chance that another processes will read the element is high.

6 Related Work

We relate our work to a wide range of middleware systems, both of the language dependent and the language independent type.

CORBA [1] is an example of programming language independent middleware focusing on interoperability. CORBA requires data to be structured into objects and interaction between objects is solely achieved through method shipping. The DSS differs from CORBA in that no structuring is enforced, instead the natural structuring of the programming language can be mapped to appropriate distribution support. The DSS supports objects(mutables) in addition to many other abstract entity types with a multiplicity of entity consistency protocols, method shipping being only one choice for mutables.

InterWeave [22] is limited to distributing data on the level of abstract memory pages. Once again this is only one particular mapping to mutables and is achievable with the DSS as well. Unlike CORBA but similar to the DSS InterWeave has an open architecture for consistency protocols, called the coherence module with eligible protocols. While we have a dynamic architecture for coordination, i.e. the coordination strategy, they have chosen a static model with dedicated servers, much like the stationary coordination strategy in the DSS. Furthermore InterWeave has no support for automatic memory management.

.Net [5] offers a single entity consistency protocol for one single type of entity, objects. It is however possible to change the protocol, using the (not well-documented) interception mechanism at considerable performance cost.

JavaParty [23] and cJVM [24] are, though dedicated to just Java, two interesting systems with respect to their functionality. Using preprocessing and new library routines, JavaParty offers true transparency for Java with a proper thread distribution model and provides mobile state protocols. Similar to our model JavaParty allows for definition of entity consistency protocols in runtime for single objects. cJVM provides similar features as JavaParty but with the approach of extending the runtime system. The architecture is open for protocol addition, every object can choose from a set of consistency protocols. However, the patterns are monolithic and cannot be constructed from sub components as in our model(by composing coordination, consistency, and memory management strategies). Both systems are geared toward Java objects, and tightly integrated with the Java system.

The concept of distribution support based on a clear distinction between mutables and immutables was introduced with the Emerald[25] system. However, Emerald did not follow up on the potential strengths of this concept, allowing for a wide range of entity consistency protocols. None of the mentioned systems explore the domain of abstract entity types as we do, nor do they attempt to support all high level programming languages. Furthermore, we have found no trace in the literature exploring what we refer to as mobility for coordinators in open dynamic distributed systems.

7 Conclusion

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

The messaging capacity of the DSS has been evaluated and compared with other systems. The evaluation shows that the implementation is both efficient and with low overhead, especially if all the functionality provided by the middleware is taken into consideration. Furthermore, we have shown that an appropriate choice of protocol is the most dominant factor when tuning a distributed application for performance.

A novel design of entity consistency protocols is presented. By separating functionality into three different parts(strategies), development of new protocols is greatly simplified. The powerful messaging framework simplifies protocol development even further. This is indicated by the few lines of C++ code required to realize the complex consistency strategies mobile-state (281 lines) and eager invalidation (219 lines).

The comparison between Mozart and Oz-DSS indicates that the benefit of tightly integrated distribution support is so small that it is not worth the effort.

We think that the abstract entity model, together with the large protocol base, should make a library based on our model, e.g. the DSS, a first choice for any programming system/language that needs distribution support.

8 Acknowledgments

We wish to express thanks to Joe Armstrong, Prof. Kazunori Ueda, Sameh El-Ansary, and Ali Ghodsi for invaluable help and advice.

This work is funded by the European IST-FET PEPITO project and the Swedish Vinnova PPC and VR FDC projects

References

[1] OMG., December 2002. The CORBA specifications, http://www.omg.org.
[2] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204. Springer, 2002.
[3] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 376–395. Springer, 1999.
[4] Sun Microsystems. The remote method invocation specification, dec 2002. Available from http://java.sun.com.
[5] Ingo Rammer. *Advanced .NET Remoting*. APress, april 2002.
[6] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Efficient Replicated Method Invocation in Java. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 88–96, San Francisco, CA, June 2000. ACM Press.
[7] S. Haridi, P-V. Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.
[8] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, San Francisco, CA, June 1999. ACM Press.
[9] C. Wikstrom. Distributed programming in erlang. In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation, PASCO'94*, volume 5 of *Lecture notes series in computing*, pages 284–293. World Scientific Publishing Co., 1994.
[10] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.
[11] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 305–318. USENIX Association, 2000.
[12] Mozart Consortium. http://www.mozart-oz.org, December 2002.
[13] Per Brand Erik Klintskog, Zacharias El Banna. A generic middleware for intra-language transparent distribution. Technical Report T2003:01, Swedish Institute of Computer Science, SICS, January 2003.
[14] Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. In *Internet Programming Languages, ICCL'98 Workshop, Chicago, IL, USA, May 13, 1998, Proceedings*, volume 1686 of *Lecture Notes in Computer Science*. Springer, 1999.
[15] S. Haridi, P-V. Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, 1999.

[16] R-H. Halstead. Multilisp: a language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 501–538, 1985.

[17] K. E. Kerry Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In W. Cheng and A. S. M. Sajeev, editors, *Proceedings of 6th Annual Australasian Conference on Parallel and Real-Time Systems (PART'99), Melbourne, Australia*. Springer Verlag, 1999.

[18] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. In *Proceedings Int Conf on Parallel and Distributed Computing (PDCS'96)*, pages 125–130, Dijon, France, September 1996. ISCA.

[19] H. Guyennet, J-C. Lapayre, and M. Trehel. Distributed shared memory layer for cooperative work application. In *22nd IEEE Conference on Local Computer Networks (LCN '97), 2-5 November 1997, Minneapolis, Minnesota, USA, Proceedings*, pages 72–78. IEEE Computer Society, 1997.

[20] P. Brand E. Klintskog, A. Neiderud and S. Haridi. Fractional weighted reference counting. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester*, volume 2150 of *Lecture Notes in Computer Science*, pages 486–490. Springer, 2001.

[21] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, December 1993.

[22] C. Tang, D. Chen, S. Dwarkadas, and M. Scott. Efficient distributed shared state for heterogeneous machine architectures. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 412–422. IEEE Computer Society, 2003.

[23] Bernhard Haumacher and Michael Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *CPC2001, 9th Workshop on Compilers for Parallel Computers*, pages 83–94. IEEE Computer Society, June 2001.

[24] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11. IEEE Computer Society, 1999.

[25] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.