

The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities

Erik Klintskog* Zacharias El Banna† Per Brand* Seif Haridi†

*Swedish Institute of Computer Science, Box 1263, SE-164 29 Kista, Sweden

†IMIT-Royal Institute of Technology, Electrum 229 * 164 40 Kista, Sweden

E-mail: {erik,perbrand}@sics.se, {zeb,seif}@imit.kth.se

Abstract

This paper describes a novel language independent model for distribution of language entities, which allows for fine-grained instrumentation of entity consistency protocols on a per-entity basis. The model is implemented as a middleware component, designed to enhance arbitrary high-level programming languages with distribution support on the language entity level. The middleware library is extendable using internal interfaces to add new protocols over three different aspects of distribution.

1. Introduction

This paper presents a design and implementation of a middleware library, the DSS (Distribution SubSystem). The DSS is designed to simplify the implementation of efficient distributed programming systems providing transparent distribution. Our view on transparent distribution is that threads located in different processes manipulate shared data structures with the same semantics as if the threads were located in the same process (modulo failure and latency). Our view differs from the common view on transparent distribution where manipulation of distributed data structures is only required to be syntactically similar to manipulation of local data structures, but may have different semantics, e.g. Java RMI [10] and .Net remoting [13].

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 usage patterns of distributed objects).

1.1. Motivation

This work is motivated by the need of language independent middleware providing full distribution support for arbitrary high-level programming systems (PS). By full support we mean that all language entities should potentially be sharable, with preserved semantics, in a distributed computing environment.

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 data types that are linked to the operating system or hardware can only be shared with a limited distribution behavior; files, for instance, can only be shared as stationary objects with remote access.

Sequential consistency is generally a requirement [11] to preserve the semantics of many language entities. A prototypical example is the semantics of objects in OO languages. This does not preclude the use of a weaker consistency model to improve the performance of distributed applications [20], but from our point-of-view this should be reflected in a different type of language entity (e.g. different type of object). To write efficient, stable and scalable distributed applications a language entity should be able to use different entity consistency protocols. This has been attempted for Java in numerous extensions [6, 2, 17, 9, 16] that offer more than one protocol; but still they only offer a limited set. In our view there should be a complete suite of state-of-the-art entity consistency protocols. The challenges are: 1) providing an efficient language independent solution, 2) avoiding a combinatorial explosion of interface functions in the API, 3) achieving an easily extendable implementation for future consistency protocols.

1.2. Contribution

The major contributions of this paper can be summarized as follow. Firstly, for the developer of a distributed programming system (DPS), we provide a model of distribution-support for language entities. The model is 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 systems (e.g. Java, C# and Mozart[1], but not C).

Secondly, for the application developer, we provide a model of distribution that guarantees functional properties (i.e. preserving consistency) for a given distributed entity. Mapping programming language entities to entity consistency protocols can be done at runtime on an entity basis, based on expected pattern of use and not on the entity type.

Thirdly, we describe a novel component-based design of entity consistency protocols. The model allows for fine-grained control of non-functional properties, as well as simplified design and incorporation of new distributed protocols.

Finally, we describe the design and implementation of the DSS¹. A programming system can be extended with the DSS in order to add distribution support.

1.3. Paper Organization

Sections 2 and 3 describes the language independent entity model for distribution of language entities. The entity consistency framework is described in Sect. 4. The internals of our middleware library is described in Sect. 5. Sect. 6 relates the DSS to other work, and is intentionally located at the end since it requires an understanding of the DSS model. The implications of the design, its strengths and future research directions is discussed in Sect. 7. A short conclusion is given in Sect. 8.

2. A Model for Sharing Language Entities

The model implemented by our middleware library is an approach to achieve transparency for shared language entities. The model uses the notion of *local entity instances*, acting as proxies for a shared entity that are present at every process with a reference to the shared entity. These entity instances are all inherently equal. Threads perform operations on their local instance and all operations are coordinated by an entity consistency protocol, provided by a middleware library. The role of the entity consistency protocol is to decide where and

¹ Available for download at <http://dss.sics.se>

when an operation will be executed. Conceptually, an operation can either be executed at the instance, *local execution*, or at another instance(s), *remote execution*.

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.

2.1. The Abstract Entity

Each local entity 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 a connected abstract entity instance. All interaction with an abstract entity instance is done using *abstract operations*², 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 thread how to 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 Analogous to the distinction between abstract and concrete language entities there are potentially many concrete operations per abstract operation.

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[18], 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 effectively 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.

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[5] is a popular optimization in distributed object systems. To efficiently support distribution of this class of entities, we provide *Relaxed Mutable* and *Relaxed Transient* abstract entity types that guarantee at most PRAM (or FIFO) [14] consistency.

3. Interaction 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 an *entity 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 if previously skeleton.

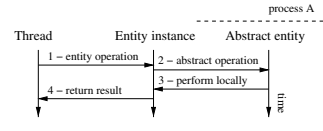


Figure 1. Sequence diagram of interaction with a complete entity instance.

- **executeOperation** The callback is given a description of a concrete entity operation to execute on the complete 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 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 prime example is a shared object using the mutable abstract entity. Depending on the chosen type of protocol, different events can be observed for the same abstract operation. We present three examples, in the form of sequence diagrams, depicting interaction with a shared object. In all three examples a thread invokes a method on the object. In the first example the protocol allows for local execution, clearly the object has the status complete. In the second and third example the operation cannot be performed immediately, the object may or may not have skeleton status. The second example shows the sequence diagram when a state moving protocol is used, the third example shows the sequence diagram when an operation moving protocol is used.

3.1. Example: Complete Entity Instance

Fig. 1 depicts the sequence of events that occurs when a thread at process A performs a method invocation (1) on a shared object. The method invocation is translated into an abstract operation and passed on (2) to the abstract entity. The entity instance's status is complete and the operation may be performed locally, so the abstract operation returns (3) **local**. The operation is executed and the result is returned to the calling thread (4).

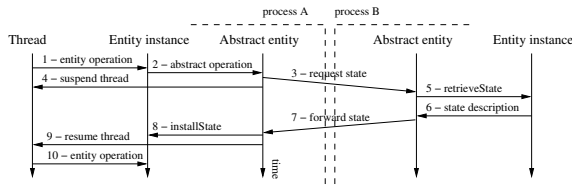


Figure 2. Sequence diagram of a state transporting protocol.

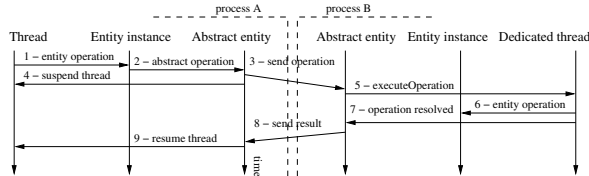


Figure 3. Sequence diagram of the operation transporting protocol.

3.2. Example: State 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 request for a state description is sent³ (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(6). The description is passed back (7) to the abstract entity at process A, where it is installed (8) using the `installState` callback. When the state is properly installed, the suspended thread is resumed (9) using `resumeThread`, and told to redo the operation (10).

3.3. Example: Operation Transporting Protocol

Fig. 3 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

³Via the coordination network, to be described later.

operation description and uses the callback `executeOperation` to perform the operation locally. The operation is executed by a dedicated thread⁴ (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).

3.4. Globalizing, Localizing and Failing an Entity

The act of associating a local entity instance with an abstract entity instance is called *globalization*. When an entity instance has been globalized, it can be referenced from other processes than the process of origin. The reverse act, called *localization* happens when an entity instance is released from its abstract entity. After localization the entity can be freely accessed from the PS level, without interaction with the DSS.

Globalization occurs when a reference to a previously local entity is passed over the network. The one exception to this are immutable data types associated with the immediate replication protocol. For efficiency reasons entities are only globalized when they truly become shared. Globalization is thus associated with the act of exporting an entity reference. Abstract entities are also created in association with importation. When a reference to shared entity is imported, either received in an entity state description or received as an argument to an entity operation, an entity instance and an abstract entity instance is created. The DSS automatically creates the abstract entity, and calls the PS to create the corresponding local entity instance. The DSS supports the at-most-one-copy-per-process property, so that new local entity instances are only created if the entity is unknown to the importing process.

There are two conditions triggering localization, both detected in the DSS. Firstly, when the memory management facility detects that only one process refer to a shared entity the need for distribution support ceases, the abstract entity instance may be reclaimed (i.e. there is only one instance left). Secondly, an abstract entity instance can explicitly terminate distribution support, e.g. when an immutable or transient become complete.

Due to changes in the environment, i.e. process failures, the DSS can detect that a shared entity has ceased to function properly. This is reflected up to the PS level where appropriate actions can be taken, e.g. thread exceptions.

⁴A thread created solely for the purpose of remote execution.

3.5. Example of an Abstract Entity

An example of a distributable array is presented below, in pseudo code, to depict how a language entity can be distributed using the abstract entity model. The language entity is an object containing an array of integers. The *mutable* abstract entity type is suitable for this entity type and thus the array object implements the *mutable mediator* interface, to be used by the abstract entity for callbacks.

Note that this example assumes a high level programming system, a pseudo system that takes care of thread-scheduling. Any references to descriptions of entity operations or calling threads are passed in DSS specific formats. In our pseudo code example we call the operation mediator a *DSS_data* type and it manages arguments and/or entity descriptions.

```
class IntegerArrayObject implements MutableMediator

bool isDistributed;
mutable_abstract_entity absent
int state[40]

AOrret executeOperation(DSS_data op){
  int type = op.getData()
  case 'type' of 'write' then
    int elem = op.getData()
    state[elem] = op.getData()
  elseif 'read' then
    int elem = op.getData()
    op.readData(state[elem])
  end
  return OK
}

void installState(DSS_data val){
  state = val.getData()
}

void retrieveState(DSS_data val){
  val.readData(state)
}

void globalize(){
  isDistributed = true
  absent = DSS.createMutableAbstractEntity(.default.)
  absent.connect(this)
}

void localize(){
  isDistributed = false
  absent.dispose()
}

% The method returns true if the operation could be
% performed immediately. Otherwise tells the running
% thread to suspend while the abstract operation
% is performed

bool accessElement(int& ret, int elem){
  if isDistributed == false then
    ret = state[elem]
    return true
  else
    case absent.doAbsOp(DSS_data('read',elem))
    of local then
      ret = state[elem]
      return true
    elseif suspended then
      return false
  }
}
```

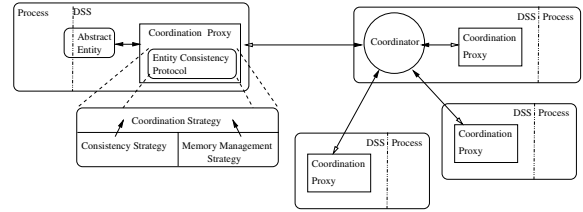


Figure 4. The coordination network. The abstract entity is coupled to a coordination proxy connecting it to the coordination network.

```
bool void assignElement(int val, int elem){
  if isDistributed == false then
    state[elem] = val
    return true
  else
    case absent.doAbsOp(DSS_data('write', elem,val))
    of local then
      state[elem] = val
      return true
    elseif suspended then
      return false
  }
}
```

This code complies with the API requirements in order to use the various mutable entity consistency protocols as shown in Figures 1, 2 and 3. Note that the creation of a dedicated per-operation thread was left out in the **executeOperation** method. In our case, the array can assume exclusive access to the state, during execution of callback and abstract operations.

4. Coordination of Abstract Entities

Every abstract entity is connected to a *coordination proxy*, depicted at the left of Fig. 4. An entity consistency protocol executes over the dynamic sub-network, called the *coordination network*, formed by participating coordination proxies. A coordination network has a hub, called the *coordination hub*, realized by one or more *coordinator(s)*. A property of the network is that proxies can always contact the coordination hub, while the inverse is not necessarily true.

To minimize the size of a coordination network, at-most-one coordination proxy per process is allowed. Obviously a process can have an arbitrary number of coordination proxies for different coordination networks for different shared entities. For any given shared entity, there exists one local entity instance, one abstract entity instance, and one coordination proxy per process and one coordination hub in the system as a whole.

4.1. 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. 4. This framework divides entity consistency into three separate sub-strategies, each implemented as a module with a well specified interface, running as parallel protocols over the coordination network.

An optimal entity consistency protocol for a particular entity and usage pattern is just a matter of sub-strategy composition. This potentially increases code reuse (in the form of reused sub-strategies) and simplifies development of entity consistency protocols. Note that while an abstract entity can use any coordination and memory management strategy it is limited in its choice of consistency strategy, the chosen consistency strategy must reflect the semantics of the abstract entity type.

4.1.1. Coordination Strategy

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

4.1.2. Consistency Strategy

The task of this sub-strategy is to uphold entity semantics, controlling entity instances connected to the coordination network and threads performing operations on the entity instances. In practice, the protocol resolves abstract operations for the abstract entity.

Interaction with local entity instances together with communication and addressing services, provided by the coordination strategy, simplifies implementation of a wide range of protocols. Examples of protocols for the mutable abstract entity are: remote-execution, mobile state and read/write invalidation. For the immutable we provide different kinds of replication: immediate-, eager- and lazy replication. For the transient abstract entity we provide protocols based on the work of [7].

4.1.3. 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 coordination hub and freeing resources. Of course by the time

when localization is achieved there may be no local references left to the local entity instance, which is handled by the memory management outside the DSS. The DSS provides implementations of fractional weighted reference counting, reference listing, time lease and persistent entities.

4.2. Classifying Network Problems

The processes which hosts the proxies and coordinators of a coordination network are subject to failures, making the processes unavailable⁵. From the perspective of a coordination proxy, two different problems can be experienced. First, the coordination hub can be unavailable. Second, a process necessary for the correctness of the consistency strategy can be unavailable, e.g. the process currently holding the state in a mobile state protocol. Both problems will make the entity instance non-coordinated, i.e. further operations are not possible. This information is propagated to PS level as a failure status on a language entity.

5. The Middleware Library

The DSS implements an expressive and efficient messaging framework used by all modules that need inter-process communication. It provides an implementation of entity consistency protocols, including interaction with entity instances (over the abstract entity interface) and inter-coordination network communication (using the messaging framework).

5.1. Messaging Framework

The messaging service provides a network abstraction for protocols executed by coordinator proxies and coordinators. The abstraction hides issues regarding reliable delivery, serialization and failure detection. Channels between processes are created allowing for unidirectional intra coordination-network communication. Message passing is FIFO, asynchronous, priority based and reliable.

5.1.1. The DSite, a Process Abstraction

All processes known to the messaging service, including the local process, are represented as *DSite* objects. DSite objects are created from descriptions imported in messages. A DSite object stores a globally unique identity of the DSS instance it represents. The identity is assigned when boot strapping a DSS. Intra

⁵We use a simple failure model that describes the availability of a process, i.e. available, communication-problem and crash-failure

DSS communication is based on the unique identities of different DSS instances, i.e. name based addressing.

A DSite serves three purposes: first, it is used as a channel abstraction; messages passed to the DSite will eventually arrive at the process it represents. Second, the DSite reflects the known availability status of the process, represented as a three-state failure model: **available**, **communication-problem**, and **crash-failure**. This information is used, by coordination proxies and coordinators, to detect and report failures of coordination networks. Third, DSites are first class and can be compared for equality, stored and passed in messages. This enables proxies and coordinators to reason on the level of locations and processes, as needed by the consistency protocol.

5.1.2. An Abstraction for a Message

The messaging service provides an abstraction for messages, called a *MessageContainer*. MessageContainers are used for all asynchronous inter- and intra-DSS communication. The purpose is to hide aspects of asynchronous message sending such as delays, resending, serialization, and deserialization.

The MessageContainer is represented as a stack of items. This enables different layers in the DSS to add information to outgoing messages and remove information from incoming messages. The items on the stack are either data structures internal to the middleware, as integers and DSites, or external data structures (i.e. from the PS) as entity operations and entity state descriptions.

A MessageContainer is created whenever a coordination proxy or coordinator needs to pass a message to another coordination proxy or coordinator. Upon reception of the message a MessageContainer is also created. Due to resource limitations and scheduling, MessageContainers can exist for quite some time. For simplicity of design and implementation no extra provision is made for the special case when messages are sent between Proxies and Coordinators in the same process, short circuiting the messaging service.

5.1.3. Inter-Process Communication

The DSS is internally divided into one core module and two replaceable service-modules. The core module, the Advanced Asynchronous Protocol Machine (AAPM) implements the abstract entities, the protocols of the coordination network and a high level messaging service. It uses the two replaceable service-modules (as depicted in Fig. 5), the Communication Service Component (CSC) and the I/O Factory (IOF) to realize the name based addressing of the DSites.

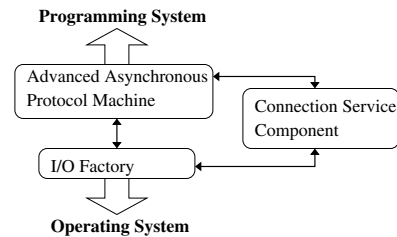


Figure 5. The three modules that internally makes the DSS.

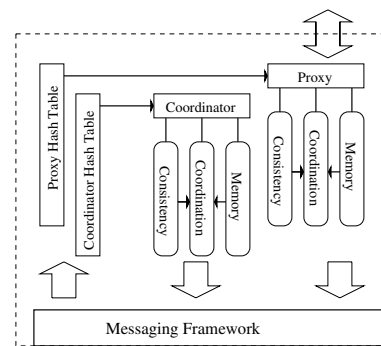


Figure 6. Internals of the Asynchronous Protocol Machine.

The IOF implements abstractions over different kinds of inter-process communication services, e.g. TCP/IP, shared memory, HTTP. It is responsible for establishing connections to remote processes, using communication service specific address descriptions (e.g. IP address and port for TCP/IP). For portability reasons the IOF handles all interaction with the OS required by the DSS.

The CSC is responsible for establishing connections to a process given its unique name. It does not directly establish the connections, but uses the services provided by the IOF. Furthermore, it is also responsible for detecting communicating problems with remote processes and classify them into a DSite availability status. The CSC is easily replaced to enable custom implementations based on application knowledge, e.g. specialized addressing schemes or failure detectors.

5.2. The Entity Consistency Protocols

The coordination proxies and coordinators are represented as objects. The sub-strategies of the coordination strategy are represented as instances of interface classes. Every coordination proxy and coordinator is associated with one instance of a consistency class, a coordination

class, and a memory management class (see Fig 6).

Every coordination network has a unique identity. The identity of a coordination network is used to locate the correct coordination proxy or the coordinator in a process. The AAPM has two hash tables, mapping coordination network identities to existing proxies and to existing coordinators (see Fig. 6).

Messages sent over the coordination network are either addressed to a proxy or to a coordinator. This information is used at the receiving AAPM to know if a coordination proxy or a coordinator is the destination of the message. The complete coordination network address is thus a triplet of *destination process* (DSite), *coordination network id*, and a *proxy-* or *coordinator* tag.

5.2.1. Sub-Strategy Classes

The instances of the three sub-strategy classes interact with the coordination proxy or coordinator they are associated with and the messaging service. Interaction from proxy/coordinator-object to strategy object is conducted over interface methods defined by the respective sub-strategy base class.

The consistency and memory management objects have access to the coordination object. The coordination strategy provides an interface for inter coordination network communication. The interface returns message containers prepared for a given destination, where a destination can be either the coordinator of the network or a coordination proxy in the network located at a particular process.

6. Related Work

We relate our work to a wide range of middleware systems, that claim to provide some sort of transparency. The systems are examples of both system dependent and system independent solutions. One of the strengths of the DSS is its system independence. However, it is still interesting to compare the DSS with system dependent solutions on the level of what functionality the systems provide.

CORBA [12] is an example of programming language independent middleware focusing on inseparability. CORBA requires data to be structured into objects and interaction between objects is solely achieved through method shipping, i.e. operation transporting. 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 [15] 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.

The concept of distribution support based on a clear distinction between mutables and immutables was introduced with the Emerald[8] 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. Discussion

Complete transparent distribution over the Internet is, as pointed out by [19], not achievable. Process/link failures and latency can not be hidden from the programmer, a distributed programming system will behave differently than a centralized programming system. More importantly, the behavior also changes with the way an application is distributed. It is clearly advantageous to keep these changes of behavior to a minimum.

The DSS is designed to support transparent distribution of language entities. An interesting and natural group of language entities can be found by taking the super-set of most high-level centralized programming languages/systems. Sequentially consistent objects in OO languages and single-assignment variables are found here. Other interesting language entities are rarely found in centralized programming languages/systems, e.g. asynchronous method invocation. Our aim with the DSS is to provide support for all potentially useful language entities. We began with established language entities found in high-level programming languages and believe that we support them all. We may be less complete in supporting mutables with weaker consistency, but believe that such entities will fit well into the DSS framework.

The DSS meets a number of important design and implementation criteria; 1) it can be coupled to a centralized programming systems to provide full distribu-

tion support, 2) the DSS is both customizable on many different levels and extendable and 3) it is an efficient implementation.

7.1. Coupling a Programming System to the DSS

The DSS is designed to be coupled to existing sufficiently high-level programming languages/systems. In addition, the DSS is also designed for future programming languages/systems that have been extended with additional language entities, e.g. mutables with weaker consistency. A detailed description can of how to couple a PS to the DSS can be found in our technical report[3]).

The abstract entity model implemented in our DSS makes distribution of language entities simple. Potentially all entities are sharable. The result of the coupling is a distributed programming system. Programmers familiar with the centralized programming system will find the DPS almost as easy to use as the PS. The major difference will be the selection of consistency strategy, a choice depending on a rough estimation of usage pattern which requires no knowledge of distributed algorithms or networking.

Our reference DSS implementation consists out of an efficient Advanced Asynchronous Protocol Machine (AAPM) realized as a linkable C++ library, a simple CSC implementation, and an IOF that provides abstractions over TCP/IP.

7.2. Customization and Extendibility

The salient property of our DSS is in its possibilities for customization. There are three important aspects of customization and extendibility. The first and central to our model of entity distribution is the fine-grained instrumentation of single distributed entities, i.e. the stipulation of which particular protocol should be used to keep entity instances consistent.

The DSS carefully distinguishes between the usage of connections from how connections are made and monitored. This is reflected in the interface between the AAPM and CSC. Connection establishment is kept separate from the protocol machine for easy customization. Examples of issues that may need customization are dealing with firewalls, dynamic IP-addresses, and appropriate security measures. Monitoring includes failure detection. Failure detection is highly dependent on the environment and an open-ended research question.

The DSS framework is easily extended with new protocols. This is reflected in our 3-dimensional consistency strategy space, where new coordination, consistency, and memory-management strategies may easily be incorporated. This is easy because 1) the protocols

will make use of the already existing messaging framework and 2) because the sub-strategies can be orthogonally extended in each dimension. Of particular importance is to be able to add a consistency strategy, using it in conjunction with already existing coordination and memory-management strategies. As consistency strategies are directly coupled to entity semantics there is probably considerably more variation here than for the coordination and memory-management strategies.

7.3. Performance

We have conducted tests in order to compare the pure messaging performance of our reference AAPM implementation to raw socket communication [4]. The overhead imposed by the language independent interface, the extendable entity consistency framework and the messaging framework to support the abstract entity model is only 2 times that of raw socket communication.

In order to measure the overhead of the implementation of our approach, we replaced the tightly coupled distribution support of the Mozart system with the DSS, resulting in the OzDSS system[4]. The Mozart system gives 12% better performance over the Oz-DSS system. However, in the light of increased functionality and superior extendibility, this small difference is acceptable.

The OzDSS shows competitive performance when compared to existing DPSs[3]. In a comparison between OzDSS[3], Sun Java, IBM Java, and Microsoft .Net, the OzDSS system was fastest by a factor of three. This result should be seen in the light of different I/O systems and serializing mechanisms in the three systems. However, it indicates that the DSS system is fast or at least as fast as state-of-the-art DPS systems.

The above results indicate that an efficient realization of our model is possible. The overhead of using a language level stream construct instead of a raw socket is only 2 times. The arguments for using sockets are effectively suppressed except for the most performance critical applications. Furthermore, the small gain in developing dedicated distribution support is simply not worth the effort when a DSS based solution only imposes a 12% overhead.

7.4. Future Work

Failure detection in our DSS implementation is weak. It is currently based on TCP errors, and only detects crash-failures in LAN settings for processes that does not change their addresses. Using our extendable CSC structure, experimenting with different strategies for failure detecting is simple. We plan to investigate this issue further.

Security is an issue that has not been targeted in the DSS. We plan to introduce encrypted channels and a trust model, based on capabilities.

8. Conclusion

We have presented a novel architecture for a language-independent middleware library. This library, the DSS, can be coupled to high-level programming languages to create powerful distributed programming systems. These distributed programming systems can then offer the programmer an extremely simple and powerful distributed programming model. In addition, the DSS is designed to be both customizable and extendable.

A novel design of entity consistency protocols is presented. Functionality is separated into three different parts or sub-strategies. For each entity one protocol from each sub-strategy type is chosen which makes for a wide variety without combinatorial explosion. In addition, a new sub-strategy protocol can be added and combined with already existing sub-strategies of other types.

9. Acknowledgments

We wish to express thanks to Joe Armstrong and Frej Drejhammar for invaluable help and advice.

This work is supported by the European IST-FET PEPITO project and the Swedish Agency for Innovation Systems and the Swedish Research Council.

References

- [1] M. Consortium. <http://www.mozart-oz.org>, Dec. 2002.
- [2] M. Dahm. Doorastha: a step towards distribution transparency, 2000.
- [3] P. B. 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.
- [4] P. B. Erik Klintskog, Zacharias El Banna and S. Haridi. The design and evaluation of a middleware library for distribution of language entities. In *Proceedings of the 8th Asian Computing Conference (ASIAN'03)*, volume ? of *Lecture Notes in Computer Science*, pages ?-?, Mumbai, India, Dec. 2003. Springer, to appear.
- [5] K. E. K. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In W. Cheng and A. S. M. Sajeve, editors, *Proceedings of 6th Annual Australasian Conference on Parallel and Real-Time Systems (PART'99)*, Melbourne, Australia. Springer Verlag, 1999.
- [6] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for Internet cooperative applications. In *Middleware'98*, The Lake District, England, 1998.
- [7] 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.
- [8] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [9] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of Java remote method invocation (RMI). In *Proc of the 4th Conf. on Object Oriented Techniques and Systems*, pages 19–36, 1998.
- [10] S. Microsystems. The remote method invocation specification, dec 2002. Available from <http://java.sun.com>.
- [11] D. Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.
- [12] OMG., Dec. 2002. The CORBA specifications, <http://www.omg.org>.
- [13] I. Rammer. *Advanced .NET Remoting*. APress, april 2002.
- [14] 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.
- [15] 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.
- [16] E. Tilevich and Y. 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.
- [17] E. Tilevich and Y. Smaragdakis. Nirmi: Natural and efficient middleware. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.
- [18] P. Van Roy, P. Brand, S. Haridi, and R. Collet. A lightweight reliable object migration protocol. In *Internet Programming Languages, ICCL'98 Workshop, Chicago, IL, USA, May 13, 1998, Proceedings*, volume 1686 of *LNCS*. Springer, 1999.
- [19] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In *Mobile Object Systems - Towards the Programmable Internet, Second International Workshop, MOS'96*, volume 1222 of *LNCS*. Springer, 1997.
- [20] H. Yu and A. 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.