Creating a Distributed Programming System Using the DSS: A Case Study of OzDSS

Erik Klintskog

April 4, 2005

Swedish Institute of Computer Science, Kista, Sweden

SICS Technical Report T2004:16
ISSN 1100-3154
ISRN:SICS-T–2004/16-SE

Contents

1 Introduction **3**
- 1.1 Outline . . . 3

2 The Distribution Subsystem **3**
- 2.1 Further Information About the DSS . . . 4
- 2.2 Model of a Programming Language . . . 4
- 2.3 DSS Distribution Support . . . 4
- 2.4 Abstract Entities . . . 5
- 2.5 Localization and Globalization . . . 5
- 2.6 Distribution Strategies . . . 6
- 2.7 The Abstract Operation Interface . . . 6

3 Mozart **7**
- 3.1 The Logical Variable . . . 7
- 3.2 The Mozart Runtime-System . . . 8
  - 3.2.1 The Virtual Machine . . . 8
  - 3.2.2 The Heap . . . 9
  - 3.2.3 The Thread Model . . . 9
  - 3.2.4 Suspending and Resuming a Thread . . . 10

4 OzDSS **10**
- 4.1 Distributing Language Entities . . . 11
  - 4.1.1 The Cell . . . 12
  - 4.1.2 Capturing Operations . . . 12
  - 4.1.3 The Mediator . . . 13
  - 4.1.4 Executing an Abstract Operation . . . 14
  - 4.1.5 Suspending and Resuming Threads . . . 15
  - 4.1.6 Callbacks . . . 15
- 4.2 Handling Mozart Threads . . . 16
  - 4.2.1 Representing a Thread . . . 16
  - 4.2.2 Resuming a Suspended Operation . . . 17
- 4.3 Customizing Distribution Behavior . . . 18
- 4.4 Reporting Failures . . . 18
- 4.5 Transporting Data . . . 19
- 4.6 Garbage Collection . . . 20
  - 4.6.1 Calculating the Root Set . . . 20
  - 4.6.2 Dropping Unreferred Distributed Entities . . . 20
- 4.7 I/O Handling . . . 21

5 Green Threads vs. Native Threads **21**

6 Discussion **22**
- 6.1 Where to Place the Guard . . . 23
- 6.2 Performance Comparision . . . 23
- 6.3 OzDSS vs Mozart . . . 25

1 Introduction

Developing distributed applications is greatly simpilfied by the use of a programming language that incorporates distribution support into the programming model. Given that distribution support is succesfully integrated into the programming model, a distributed application can be developed much like a centralized application (i.e. an application that is executed on one machine only). One powerfull approach to programming level distribution support is network transparency. We use the network transparency in the meaning that the physical distribution of an application does not affect the functionality. Thus, disregarding where a resource such as an object is located in a distributed system, the object can be accessed. Network transparency provides freedom in deployment of data over a distributed system.

Devloping implementations of programming languages, called programming systems, with comprehensive distribution support is a complicated and time consuming task. This is indicated by the scarsness of programming system that provides some degree of network transparency and in the mean time offers a stable implementation. However, we argue that this is not nessesary the case. Distribution support for a programming language can be divided into a set of tasks that are found in most existing distributed programming system, we denote a programming system that provides a programming model that incoroporates distribution support a distributed programming system. The tasks that are found in most distributed programming systems is a messaging facility that allows processes to communicate, protocols that allows data structures to be accessed from more than one proces, and serializing or marshaling routines that makes it possible to pass programming level deta betwene processes. Programming level data can for example be data structures of operations on data structures. In this technical report we describe how the Distribution SubSystem (DSS), a middlware that provides generic distribution support for programming systems is coupled to two different programming systems, Mozart[2] and C++, in order to realize two distributed programming systems.

Mozart, the target programming system for our experiment with using DSS to realize a distributed programming system, allready provides transparent distribution on the level of data structures[1]. However, the existing implementation of distribution support is tightly integrated with the code of the virtual machine, i.e. a monolitic system. This have the effect that knowledge in both the virtual machine and in the distribution support is required to maintain, and extend the Mozart system. The purpose of this document is to show how the DSS coupled to Mozart results in a non-monlitic distributed programming system, that is efficient and provides the same functionality that the monolitic Mozart system does. Efficincy is in this case measured towards the original Mozart system.

1.1 Outline

2 The Distribution Subsystem

The Distribution Subsystem (DSS) is a middleware that supports distribution of programming systems on the level of single data structures. A key component in the DSS is its expressive interface towards a programming system that is based on a concept of abstract entities that captures different basic semantically behaviors, like mutable or immutable access. Programming system data structures, i.e. language entities are pro-

vided distribution support by being coupled to an abstract entity. This section serves as a brief intrduction to DSS. Further information regarding details in the DSS nessesary to understand the design desiscions taken when coupling Mozart to DSS are introducted throughout the document.

2.1 Further Information About the DSS

The purpose of this document is to describe the Mozart system is coupled to the DSS. Detailed descriptions of the DSS is on purpose left out of this document. For deeper understanding of DSS we direct the reader to the following documents:

- *The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities[7]*
  The paper describes the API and the assoicated semantic model provided by the DSS. The internal structure of the DSS is also briefly described.

- *A Peer-to-Peer Approach to Enhance Middleware Connectivity[8]*
  The paper describes the structure of the messaging layer of the DSS. The component based design enables simple customization of connection establishment strategies. This is illustrated by the use of a simple P2P algorithm (Gnutella-like) to find suitable route between sites in the face of firewalls, NATs, etc.

- *Internal Design of the DSS[5]*

  This technical report gives an overview of the implementation of the DSS on the level of single classes. Focus is put on major concepts such as the classes that makes the interface towards a programming system, the messaging layer of DSS, and the consistency protocols of DSS.

Moreover, a webpage is maintained, `http://dss.sics.se`, where the DSS sources and the documents describing the DSS can be retrived.

2.2 Model of a Programming Language

DSS is primary designed to provide distribution support for programming systems. By coupling a programming system to DSS, a distributed programming system is created. The primitives provided by DSS are explicitly designed to enable distribution of data structures, denoted langauge entities. A distributed language entity can, in difference from a local language entity, be accessed and invoced from more than one process. Examples of language entities in Oz[13] are cells, records, objects and logical variables. Interaction and manipulation of a language entity is done by explicit operations. Examples of operations on language entities are the assing operation of the cell, field access of a record, method invocation of an object, and determining the value of a logical variable. Moreover, DSS explicitly models threads. An operation on a language entity is supposed to be performed by a thread. Threads can be suspended on an operation and can be resumed after being suspended.

2.3 DSS Distribution Support

The unit of distribution in the model supported by the DSS is a language entity. A language entity is said to be distributed if it can be accessed from more than one process. A distributed language entity is network transparent if operations can be performed on

the entity similarly from any process. In effect, disregarding how a language entity is distributed, the language entity is accessed in teh same way. The distribution support provided by DSS is network transparent.

A distributed language entity requires a consistency protocol to be network transparent. For example concider RPC, there exist a home process and a set of proxyies that allows transparent invocation of the procedure at the home. RPC is an example of a protocol. Other examples of consistency protocols are mobile state and multiple-reader/single-writer type of protocols.

The model of distributed language entities supported by the DSS is based on equal instances. An instance that represents a distributed lanaguega entity is present at every process that referes a distributed language entity. Thus, if one or more threads at a process referes a given distributed language entity, there should be a local representative of the language entity.

2.4 Abstract Entities

A local representation is controled and coordinated by the DSS middleware, operations cannot be performed on the instance without consulting the DSS. This is because an operation might have to be passed over the network in order to be resolved as in an RPC protocol. Moreover, if a replication type of protocol is used, the local repersentative has to be turned into a valid state before an operation can be performed. Control of a local representation is done by *abstract entities*. An abstract entity representes a semantical behavior, such as mutable access. Interaction with an abstract entity is done using abstract operations. Thus, operations performed on a local entity instance msut be translated into abstract operations on the associated abstarct entity. The result from the abstract operation tells how the operation on the language entity should be performed.

An abstract operation can tell the invoking thread, i.e. the thread that tries to perform an operation on a local entity instance, to suspend and later redo the operation in the case of a replication protocol. If a remote execution protocol is used, the thread is told to suspend, and later is resumed with the result of the operation. To control programming system threads the DSS exposes an interface that enables control in the form of suspension and resumption of threads.

The DSS provides three types of different abstract entity types that capture different basic semantical behaviors. First, the *mutable* abstract entity allows for both update and access of a language entity's state. Second, the *immutable* abstract entity allows for only non-destructive access of a language entity's state. Third, the *transient* abstract entity that allows for mutable updates of a languege entity until a certain point, defined from progarmming system level, where only immutable access is allowed.

The purpose of the abstract entity model is to make coupling of a programming system to DSS simple. A language entity is made distributable by interfacing the implementation of the language entity towards an abstract entity type that implements the semantical behavior nessesary to distribute the language entity. For example, an Oz cell should be coupled to the immutable abstract entity, the record that does not allow for any updateds should be coupled to the immutable abstract entity, and the logical variable to the transient abstrcat entity.

2.5 Localization and Globalization

Two central concepts in the distribution model of DSS is *globalization* and *localization*. Globalization is when a local language entity is being refered from more than one

process, i.e. the language entity becomes a distributed language entity. Localization is when a distributed lanaguage entity becomes a local entity instance, i.e. only refered from one process again. Typically, a language entity starts as a local entity, only being refered from the process where it is created. First when a reference to the language entity is passed from the creation process to another process is the entity made a distributed language entity.

This model of distribution on demand releaves the programmer of a distributed application from explicitly knowing which data that will be distributed and which data will be solely accessed from its creation process. Moreover, the local representation is intrisnicly smaller memory vice. Thus, it is benefical to not make data distributed unless nessesary.

Globalization takes in practice place when a local language entity is coupled to an asbtract entity and put under the control of the abstract entity. This explains the growth in memory footprint of a distributed language entity. For example, concider if every language entity created would be associated with an abstract entity, disregardning whether the entity will ever be distributed or not, the memory consumption would inventeble be conciderable larger than if the more conservative scheme that the DSS emphicice is used. However, the abstract entity model does not prevent using the later, eagerly distribting model. At creation, every langguage entity can be associated with an abstract entity, with the increased memory footrpint and the nessesary overhead o translatingf operations on the langauge entity into abstract opertaions.

2.6 Distribution Strategies

The abstract entity defines an interface and a set of opreations in the form of abstract operations. However, the abstract entity does not define how an abstract operation in reality is resolved. It has been indicated that both remote execution and replication type of protocols can be used. The protocol used, that defines how operations performed on different local entity isnatnces are resolved correctly is denoted a *distribution strategy*. Thus, a distributed langauge entity is represneted by the triplet, the local entity instance, the abstract entity, and a distribution strategy. The distribution strategy is assigend at creation of an abstract entity(i.e. globalization of a languiage entity), and will from that point be hidden behind the abstract entity interafce.

Internally, DSS hosts a framework for distribution strateges that allows for customization in three different non-functional domains. Customziation is possible in how a language entity is replicated, how the access structure is organized, and how the distributed garange collection is resolved. This is further described in the documents describing the DSS, see Section 2.1.

2.7 The Abstract Operation Interface

On the level of implementaion, a langauge entity instance that acts as an representative for a distributed language entity must implement a facility that translates operations into abstract operations. We denote this facility the *guard*. The purpose of the guard is to interact with the abstract entity and control the calling thread. An operation performed on a representative must be translated into an abstract operation and performed on the abstract entity. The result from the abstract operation (i.e. the guard) tells the calling thread how to proceed:

proceed The operation can be executed locally by the calling thread.

suspend The status of the entity instance does not allow for local execution. The distribution strategy resolves how the operation should be completed, while the calling thread should be suspended. Later, the thread will be resumed in one out of two modes:

doLocally The entity instance is in a state that allows for local exectution. Perform the operation without invoking the guard.

remoteDone The operation has been executed elsewhere. The suspended thread is passed the result of the operation. The thread should continue with the next instruction after the operation on the language entity.

The pseudo code below shows how the abstract entity is consulted before the *operation* is invoked. Note the `suspend` case, the thread is suspended with the `suspended()` call. The call returns when the thread should be resumed. At this point the `getResumeValue()` function returns how the thread should continue, do the operation or skip the operation.

```
void guardedOperation(Args...){
  switch(abstractEntity->abstractOperation()){
    case proceed:{
      break;
    }
    case suspend:{
      suspend();
      if(getResumeValue() == operationDone){
        return;
      }
      break;
    }
  }
  <Operation>
}
```

3 Mozart

The programming language Oz is a concurrent multi-paradigm language, simultaneously supporting the declarative, object oriented and constraint programming paradigms. The Oz language has features that make it a good candidate for transparent distribution support. First, the language is concurrent, and its implementation, Mozart, allows for thousands of simultaneously running threads. Second, the language makes a clear separation between mutable and immutable data structures. Immutable data structures have values that cannot be changed after creation, e.g. atoms and records from declarative programming. Finally, the language is dynamically typed and in addition to data structures allso code (i.e. classes and procedures), closures(higher order functions), and threads are first class.

3.1 The Logical Variable

A central concept in Oz is the logic variable data type. Logic variables are used for various purposes such as explicitly synchronizing threads, communication channels and placeholders for to-be defined information. Moreover, logic variables are used internally in the Mozart system to suspend and resume threads.

The value of a logic variable is at creation undetermined. The value of a logical variable can be determined once, and causes the logical variable to dissapear. Determining the value of a lgoical variable replcaes the logical variable with the value it is defined to. A thread that tries to read the value of a logical variable is suspended until the value is defined.

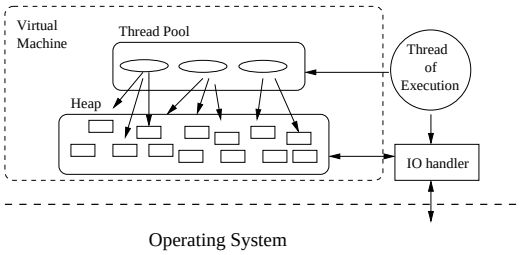

Figure 1: A conceptual view of the Mozart runtime-system internals. One thread of execution time-shares between the thread pool and the I/O handler(the garbage collector is not shown in this picture). Threads execute byte code instructions that manipulate objects on the heap.

What has been describe above is called bining a logic variable to a value. Logic variable allso supports unification of values. Unificaition, however, is out of the scope of this report.

3.2 The Mozart Runtime-System

Oz code is compiled to byte-code, executed by a virtual machine. The virtual machine is part of a runtime system, implemented in C++. The runtime system supports lightweight threads, called green threads. The virtual machine is based on the notion of *builtin-function*. A *builtin-function* can be seen as an abstraction of a set of byte-code instructions and is implemented as a C++ function. Oz primitive data structures, called language entities, are internally explicitly represented as C++ objects. Interaction (and manipulation) of language entities is done by *builtin-functions*.

Internally, the Mozart runtime-system is divided in four conceptual modules (depicted in Figure 1), the thread pool, the heap, I/O handler, and a garbage collector. Oz level data structures exist on the heap. The thread pool hosts the lightweight Oz threads that executes byte code and manipulates the data structures on the heap. A thread is either running, preempted, or suspended. At any point in time there can be at most one running thread. The I/O handling module reads and writes to sockets. The garbage collector is invoked periodically to remove unused data structures from the heap. One single native thread time shares between the thread pool, the garbage collector and the I/O handler. Thus, there is no running oz thread when the I/O is active, nor when the garbage collector is active.

In order to avoid monopolization of the single operating system thread, the time consumed by the three different activities is bounded. I/O handling is non-blocking and is periodically checked for messages to send or receive. The garbage collector, invoked when the heap is full, will finish its execution in bounded time. In order to preempt Oz threads, the Mozart runtime-system is periodically interrupted. At an interrupt, IO is handled, potential GC is performed, and the oz threads are scheduled.

3.2.1 The Virtual Machine

Mozart is a multithreaded system that manages memory by a copying garbage collector. Language level data structures are explicitly represented as C++ objects. There

is a direct mapping between a language level operation on an language entity and an operation on the C++ object that internally in the virtual machine reresents the entity.

Mozart is implemented as a register based virtual machine that executes byte-code instructions. In addition to the byte-code instructions the virtual machine hosts a set of sub-routines that byte code instructuions invoke, called *builtin-functions*. An oz program is compiled into a sequence of byte code instructions that calls builtin-functions. The builtin framework contains macro constructs that automatically performs type checks. Checks that an argument is determined (i.e. not unbound) is included in the type check. Thus, when a builtin is invoked, the in argument are determined and of the right type.

Byte code-instructions and builtin-functions differ in granularity. Byte-code instructions typically handles the registers of the virtual machine, calls buitin-functions and simple manipulations of basic data types such as integer arithmetics. The builtin-functions on the other hand execute operations on language entities, interacts with I/O and controls the virtual machine. Examples of typical builtins are array manipulation, printing to the screen, and defining parameters of the garbage collector.

3.2.2 The Heap

The language Oz is dynamically typed, i.e. no(or little) type information exist at the level of Oz-code. Instead of doing type checks at compile time, type checks are done at runtime. Given a reference to a language entity, the type of the entity can be derived. A reference to a language entity is represented in the virtual machine as a Tagged reference, called an `TaggedRef`. A `TaggedRef` contains information regarding the type of the language entity it refers.

3.2.3 The Thread Model

The green threads of Mozart are lightweight and the runtime system can schedule thousands of threads without any noticeble overhead. Threads are preemptive and are scheduled according to a round-robin scheme. A thread can suspend, or be preempted only on the granularity of byte-code instructions. Consequently, a thread cannot be suspended while executing a builtin-function.

A call to a builtin-function is represented by a special byte-code instructions that has as argument the target builtin-function and the arguments to the builtin-function. The builtin-function interface towards the virtual machine controls the calling thread. Depending on the status of the arguments to the builtin-function, the operation will either *succed*, *fail*, or *suspend*:

succeed The thread executes the next instruction.

fail An exception is raised, and the exception handling mechanism is invoked.

suspend The thread is suspended because some of the arguments to the *builtin* was not determined i.e. unbound. Note that the thread is not suspended in the midst of the operation, instead the thread is suspended conceptually before executing the *builtin*. The runtime system will guarante that the thread will redo the *builtin* when at least one of the undetermined arguments is determined.

replace An instruction has been pushed on the stack of the thread, much like a functional call. The thread will first execute the pushed instruction and then the next instruction.

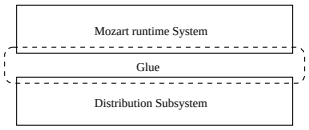

Figure 2: A schematic picture of the OzDSS system. The distribution support of Mozart is replaced by the DSS. In order to couple the two systems together, a glue layer is introduced.

3.2.4 Suspending and Resuming a Thread

Threads in Mozart suspends on the granularity of byte code instructions if the byte code instruction cannot be executed for some reason. The common cause is that one or more of the arguments are undetermined (i.e. a logical variable). In our case, suspension should conceptually be on a virtual instruction in between the instruction that calls a *builtin-function* and the next instruction.

Mozart threads are controlled by logic variables, a thread is said to be suspended *on* a logic variable. Explicit suspension of a thread is achieved by forcing the thread to call an special instruction that suspends on a special-purpose logical variable, called a control-variable. The thread is resumed by binding the control-variable (when the value is defined, the Mozart runtime system automatically resume the suspended thread).

4 OzDSS

We have used the Mozart system to create OzDSS in which the integrated distribution support of Mozart has been replaced by an instance of the DSS middleware. OzDSS provides the same programming model to the programmer as Mozart, with conservaitive extensions in the form of interafces that allows the programmer to make use of the instrumentability provided by DSS. In short, a programm written for Mozart can be executed on the OzDSS system.

In order to couple the DSS to the Mozart engine, the functionality of Mozart must be adopted to the interface of the DSS. Data structures and threads of Mozart must be coupled to their counterparts in DSS, the abstract entities and the thread representation accordingly. A mediation layer, called the *glue* is introduced between the Mozart system and the DSS, see Figure 2.

Figure 3 depicts the Mozart system extended with the DSS. Note how the glue hosts intermediate objects, on the form of boxes that connects entities on the heap with abstract entities in DSS. In order to couple Mozart with DSS, the glue must provide mediation of four tasks:

1. **Distribution enables language entities.** Language entities in Mozart is, is based on type, coupled withe eligible abstract entities in DSS. Furthermore, operations on languge entities must be intercepted. and if the language entity is distributed, translate the operation into an abstract operation.

2. **Control Mozart threads.** The structure necessary for controlling language level threads from the DSS must be provided. It must be possible to suspend a threads that performs an operation on distributed language entities and resume a sus-

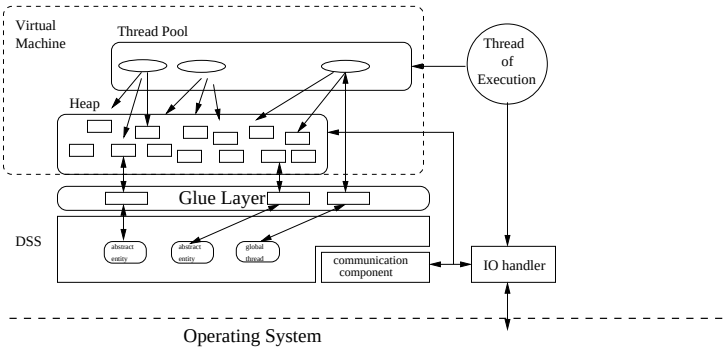

Figure 3: The DSS coupled to the Mozart runtime system. The figure depicts the internal runtime scheduler of the Mozart system, denoted *Thread of Execution*. The runtime scheduler interleaves thread execution and IO handling. The threads of the virtual machine interact with the DSS over the abstract entity interface. Furthermore, the DSS makes use of the IO handler to resolve process communication.

pended thread so that the thread redo the operation on the distributed languege entity.

3. **Integrate the marshaler with the DSS.** Mozart implements a marshaler[12] for the integrated distribution support layer, this marshaler should be adapted to the new distribution layer.

4. **Integrate the DSS in the garbage collection loop.** Garabage collection of instances of distributed languege entities is controled by DSS. DSS will have references to data structures on the heap that has to be updated. Finally, the DSS requires information of when distributed data structures are no longer needed in order to cater for global garbage collection.

4.1 Distributing Language Entities

The purpose of OzDSS is to show the feasability of DSS as a middleware for programming systems. It is thus natural that OzDSS adapts the same model of distribution as Mozart, transparent distribution of the language entities of Oz. The same data structures are to be used for distributed entities as for local entities, i.e. no special entities are to be used. This is for two reasons, first, minimal change should be imposed on the Mozart engine, second, a distribute entity should behave as a local entity. A distributed entity is represented by a, potential, *complete* instance at each process that refers the distributed entity. Complete means that the representation is in a coherent state and that operations can be performed on the instance. Any of the instances that represents a distributed language entity can act as server for the other instances. The behavior is defined by the associated distribution strategy in the DSS. For the rest of the document we will use the notation of a *local entity*, *distributed entity* and *entity instance*. A local entity is a language entity that can only be accessed from one process. A distributed entity is an entity referred from multiple processes. A distributed entity is represented by an entity instance at each process that holds a reference to the distributed entity.

OzDSS is designed with the assumption that the distribution support should impose minimal penalty on the performance of the local execution. Thus it is important to not slow down manipulation of local data structures. Ideally the memory footprint of a data structure should not grow just because the data structure potentially can be distributed. However, when distributed, the total size, including the abstract entity, will be larger than a local instance

This section describes how the language entities of Oz are coupled to abstract entities. In order to simplify the description, the cell language entity is chosen as an example.

4.1.1 The Cell

The cell is a language entity that implements an explicit mutable pointer to arbitrary Oz language entities. Three operations are possible to perform on the Cell; access, read the current value of the cell; assign, update the value of the cell; exchange, update the value and return the old value, atomically. The basic operations are reflected in the interfaces provided by the class that implements the cell:

```
class Cell:public Tertiary{
private:
  TaggedRef val;
public:
  Cell(Board *b,TaggedRef v) : Tertiary(b, Co_Cell,Te_Local), val(v){}
  TaggedRef getValue() { return val; }

  void setValue(TaggedRef v) { val=v; }

  TaggedRef exchangeValue(TaggedRef v){
    TaggedRef ret = val;
    val = v;
    return ret;}

};
```

The cell inherits from the Tertiary class, a class that auguments a language entity with meta information. The meta information is used to express non-functional aspects regarding an language entity. We are here only addressing the use of the Tertiary for differentiation between a local and a globalized language entity. However, the Tertiary is allso used by the constraint engine of Mozart. The Tertiary holds a `Tagged2` object, that provides a tag (a two bit value) and a pointer value.

```
enum TertType {
  Te_Local       = 0,
  Te_Distributed = 1
};

class Tertiary: public ConstTerm{
private:
  Tagged2 tagged; // TertType + Board || TertType + OTI
public:

  TertType getTertType()       { return (TertType) tagged.getTag(); }
  void setTertType(TertType t) { tagged.set(tagged.getData(),(int) t); }

  void setIndex(int i) { tagged.setVal(i); }
  int getIndex() { return tagged.getData(); }

  bool isDist() { return (getTertType() == Te_Distributed); }
};
```

The memory layout of the cell object is depicted in Figure 4. The cell requires two memory words for its representation, one word for the meta-information from the tertiary and one word to refer other data structures.

4.1.2 Capturing Operations

An operation on a language entity is implemented as a builtin-function. The code for the builtin-functions is separated from the byte-code instruction interpreter, it is thus convenient to intercept an operation ona distributed langauge entity on the level of the

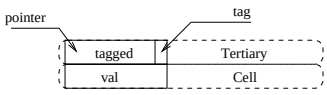

Figure 4: The memory layout of the object that implements the cell. The object is two words large. The base class of the Cell, the Tertiary adds one word containing meta information. The cell in requires one word for storing the reference to the data structure the cell refers.

corresponding builtin-function. A distributed language entity differs (on the Mozart level= only in the information found in the Tertiary extension, thus the same builtin-function will be called for a local and for a distributed entity. Conceptually, and shown in the code below, if the entity is distributed, the guard has to be invoked. The function pointer `cellDoAccess` acts as the guard for the access operation on a cell.

```
OZ_Return accessCell(OZ_Term cell,OZ_Term &out)
{
  Tertiary *tert=(Tertiary *) tagged2Const(cell);
  if(tert->isDist()){
    if((*cellDoAccess)(tert, out))
        return BI_REPLACEBICALL;
  }
  out = ((Cell*)tert)->getValue();

  return PROCEED;
}
```

The guard can allow for local access of the cell. Local access is reflected in the code when the `cellDoAccess` function returns false. In the true case, the return value `BI_REPLACEBICALL` indicates that a new operation has been pushed upon the call-stack of the running thread. Note that the return value, `out` is passed by reference to `accessCell`. Not shown here, `out` is called by reference to the `cellDoAccess` function as well, this will be further discussed in section 4.1.4 .

If the operation is performed locally, either because the cell is local or because the abstract entity allows for a local access the builtin returns `PROCEED`. The return code `PROCEED` tells the virtual machine that the builtin has completed successfully.

The example of cell access is representative for how one representation of a language entity is used to implement operations on both a local and distributed entity. Whether a language entity is distribute or solely local is described in the meta data held by the Tertiary extension.

4.1.3 The Mediator

An intermediate object connects the abstract entity and the entity instance. The intermediate object, simply called the *mediator* translated operations on the language entity into abstract operations. Morever, the mediator implements the code required by the manipulation done by the abstract entity on the entity instance, e.g. perform remote operation, require an entity state description or installing an entity state descritpion. Figure 5 depicts how the mediator acts as a bridge between the language level entity instance and the abstract entity.

Another possibiilty to connect language entities and abstract entities would have been to let the language entity communicate directly with the abstract entity. In comparision, the choosen model makes the implementation of the languge entity free from distribution specifics, resulting in leaner and simplier to maintain code. Moreover, the design shows good separation of conerns, knowledge of how an langauge entity

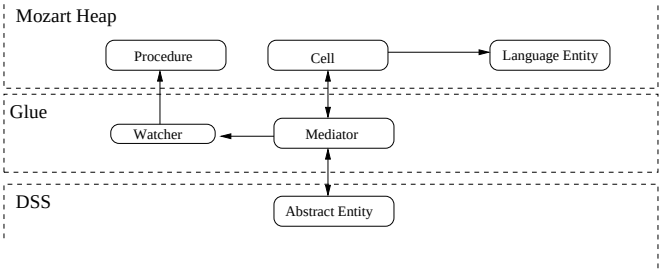

Figure 5: A distributed cell with one installed watcher. The cell is connected to a mediator in the glue; the mediator is in turn connected to an abstract entity. A failure handler local to the process is installed on the distributed cell, explicitly represented by a watcher object. The procedure associated with the watcher exists on the heap and is referred to by the watcher.

is distributed does not exists at virtual machine level, only the fact that an language entity is distribute. A mediator is created first when a language entity is globalized, i.e. assocaited with an abstract entity, and removed when localized.

The OzDSS system implements failure handling on the level of single language entities. If a distributed language entity is unusable due to a network problem, the entity is said to have failed, dedicated code, called a watcher, is executed. A watcher, in the form of an Oz procedure, is assigned an entity together with a trigger condition.

4.1.4 Executing an Abstract Operation

In order to minimize the dependencies between the centralized engine and the DSS, the actual interaction with the DSS is done in the glue. Thus, code in the builtin called by the virtual machine detects that the language entity is distributed and calls a guard function in the glue. Details regarding how to retrieve the mediator from a language entity are local to the glue.

A language operation performed on a distributed entity is transformed into a dedicated function in the glue, i.e. each language operation has its distributed counterpart in the glue. Thus, the choice of abstract operation is statically defined. For example, the cell access operation is transformed into *cellDoAccessImpl* that always performs a *read* abstract operation on the mutable abstract entity associated with the cell instance, see below.

```
bool cellDoAccessImpl(Tertiary *p, TaggedRef &ans)
{
  CellMediator *me = static_cast<CellMediator*>(index2Me(p->getIndex()));
  AbstractEntity *ae = me->getAbstractEntity();
  MutableAbstractEntity *mae = static_cast<MutableAbstractEntity*>(ae);

  DssThreadId *thrId = getThreadId();

  PstOutContainerInterface** pstout;
  OpRetVal cont = mae->abstractOperation_Read(thrId,pstout);
  if (pstout != NULL){
    *(pstout) = new PstOutContainer(oz_nil());
  }

  switch(cont)
    {
    case DSS_PROCEED:
      {
        return true;
      }
    case DSS_SUSPEND:
      {
        OZ_Term var = oz_newVariable();
        ans = var;
```

```
         thrId->setThreadMediator(new SuspendedCellAccess(me, ans));
         return false;
      }
   }
}
```

The code above shows how the mediator is retrieved, via a cast, from the tertiary using the `getIndex` method. In turn, the abstract entity is retrieved from the mediator. Follows is the invocation of the abstract operation. Note how the `pstout` double pointer argument to the abstract operation is handled first as a return value and potentially later filled with a value. Only if `pstout` points to a non `NULL` value is a pstcontainer allocated. In this very case the pstcontainer transports a dummy value (see Section 4.1.6 where the callbacks are described).

If the abstract entity returns `DSS_SUSPEND` the calling thread is suspended. Later, the thread is resumed and told to either redo the operation, or resumed and passed the result of the operation (i.e. the operation has been executed remotely). Disregarding how a thread suspended on an abstract operation is resumed, the builtin is, from the perspective of the Mozart virtual-machine, already executed.

4.1.5 Suspending and Resuming Threads

If the abstract operation returns `DSS_SUSPEND`, the calling thread is suspended. A thread suspended on an abstract operation is resumed and either instructed to redo the language operation, or handed the result of the operation and instructed to perform next instruction. A thread resumed and told to redo the operation should not redo the *guard* but only the *operation*(see Section 2.7).

As allready described, it is impossible to suspend while executing a builtin-function. Thus, the task of redoing the operation is delegated to the thread mediator (the thread mediator is described in detail in Section 4.2). Thus, if the thread is resumed and told to redo the operation, the *builtin-function* is not called, but a function that implements the *operation* is called.

As seen in the example above, disregarding how the thread is woken up, a placeholder `var` is created. The answer thus points at a language entity, in the form of a logical variable.

4.1.6 Callbacks

Apart from controlling operations on data structures, an abstract entity must be able to interact with the data structure that represents the local language entity instance:

- Execute a language level operation on a language entity instance.
- Retrieve a description of the correct state of a language entity instance. The description must be detailed enough to turn another instance of the distributed language entity into the same state as instance the description is retreived from.

In order to couple a data structure to an abstract entity, the abstract entity must be able to retrieve and install a complete representation of language entity. Moreover, each abstract operation exposed by the abstract entity requires corresponding callbacks of the mediator. The cell mediator is used as an example to explain how the callbacks are implemented in OzDSS.

An operation performed on a language entity that represents a distributed entity is either executed locally or remotely. If the operation is executed remotely, it is actually the abstract operation that is transported. The operation on the language entity is

passed as an argument to the abstract operation to the process where it will be executed. The abstract operation is performed on the target instance (the cell mediator), with the operation as the argument. The code below depicts the callback code for the *read* abstract operation implemented by the cell mediator and called by the abstract entity. In the case of the cell there is only the *access* language operation that is realized by the read abstract operation. The language operation takes no arguments, this is reflected in the callback code, the in argument `pstin` is not used. The operation is atomic, similar to the language level operation, thus no thread is spawned to execute the operation.

```
AOcallback
CellMediator::callback_Read(DssThreadId *id,
                            DssOperationId* operation_id,
                            PstInContainerInterface* pstin,
                            PstOutContainerInterface*& possible_answer){
  CellLocal *cell = static_cast<CellLocal*>(getConst());
  TaggedRef out   = cell->getValue();
  possible_answer = new PstOutContainer(out);
  return AOCB_FINISH;
}
```

Retrieving and installing the state of a cell is shown below. When transferring the state from one process to another, a state description is retrieved from the first process, and put in a `pstconatiner`. At the destination process, the install method is called with a `pstcontainer` holding a description of the current state. The cell instance at the receiving process sets its pointer to the received `TaggedRef`.

```
PstOutContainerInterface*
CellMediator::retrieveEntityRepresentation(){
  CellLocal *cell = static_cast<CellLocal*>(getConst());
  TaggedRef out   = cell->getValue();
  return new PstOutContainer(out);
}

void
CellMediator::installEntityRepresentation(PstInContainerInterface* pstIn){
  PstInContainer *pst = static_cast<PstInContainer*>(pstIn);
  TaggedRef state =  pst->a_term;
  CellLocal *cell = static_cast<CellLocal*>(getConst());
  cell->setValue(state);
}
```

The three examples shown above depict the strengths in dynamic typing. Knowledge of the type of arguments passed in pstcontainers is not required; instead the values are passed as opaque data to and from the language level data structure.

4.2 Handling Mozart Threads

The abstract operations interface provided by the abstract entities is based on the notion of a calling thread. In order to handle different implementation of threads the DSS provides a generic framework for handling programming system threads. An programming system level thread that interacts with an abstract entity must have an DSS representative in the form of a `DssThreadId`. A `DssThreadId` communicates with its programming system level thread over an instance of a callback interface.

4.2.1 Representing a Thread

The Mozart thread representation is extended with field holding an opaque value that can be read and written from the glue. The field is used to store a reference to a `DssThreadId` class that represents the thread in the DSS. A DSS thread representation is first allocated when needed, i.e. when a programming system thread performs an operation on a distributed entity. The code below shows how the `DssThreadId` is retrieved for the currently running thread.

```
DssThreadId* getThreadId(){
  TaggedRef thr = oz_thread(oz_currentThread());
```

```
  DssThreadId *id = reinterpret_cast<DssThreadId*>(oz_thread_getDistVal(thr, 1));
  if(id == NULL) {
    id = dss->m_createDssThreadId();
    oz_thread_setDistVal(thr, 1, reinterpret_cast<void*>(id));
  }
  return id;
}
```

The `ThreadMediator` is only invoked by the DSS when resuming the programming system level thread. Consequently, the `DssThreadId` only needs a reference to a mediator when the thread is suspended. This is depicted in the code for the cellAccess, see Section 4.1.4.The mediator requires implementation of two interfaces, one that instructs the thread to redo the language operation (only the *operation*), and one that instructs the thread to continue with the next instruction. The interface for the Mediator glue base class is shown here:

```
class SuspendedThread: public ThreadMediator
{
public:
  OZ_Return suspend();
  OZ_Return resume();

  virtual WakeRetVal resumeDoLocal() = 0;
  virtual WakeRetVal resumeRemoteDone(PstInContainerInterface* pstin) = 0;
};
```

4.2.2 Resuming a Suspended Operation

The `SuspendedThread` class requires implementation of two methods that resumes the language-level suspended thread. The first method `resumeDoLocal` tells the suspended thread that the language level operation now can be safely performed on the target programming system data structure. However, the *guard* should not be executed again. The `resumeRemoteDone` method tells the suspended thread that the operation has been executed, and the result of the operation is found in the `pstin` argument.

Since the builtins in Mozart are atomic, it is impossible to execute the *body* of and operation without invoking the *guard* by forcing the suspended thread to redo the bytecode instruction. Instead, the `SuspendedThread` instance will execute the body. Below is the code for resuming and instructing a thread suspended on a cell access to do an operation locally. Note that the object holds a reference to the variable `a_var` used as a placeholder for the answer (see Section 4.1.4). Moreover, the method returns `WRV_DONE` signaling the DSS that the operation is completed, i.e. no programming system thread is spawned that is manipluating the state of the language entity instance.

```
WakeRetVal SuspendedCellAccess::resumeDoLocal(DssOperationId*){
  CellLocal *cell     = static_cast<CellLocal*>(a_med->getConst());
  OZ_Term contents = a_cell->getValue();
  oz_unify(a_var,contents);
  resume();
  return WRV_DONE;
}
```

Passing a result to a suspended thread in OzDSS is straightforward. The SuspendedThread object holds a reference to the logical variable `a_var` used as a place holder for the answer. The result of the operation is received in the `pstin` argument.

```
WakeRetVal SuspendedCellAccess::resumeRemoteDone(PstInContainerInterface* pstin){
  PstInContainer *pst = static_cast<PstInContainer*>(pstin);
  oz_unify(a_var, pst->a_term);
  resume();
  return WRV_DONE;
}
```

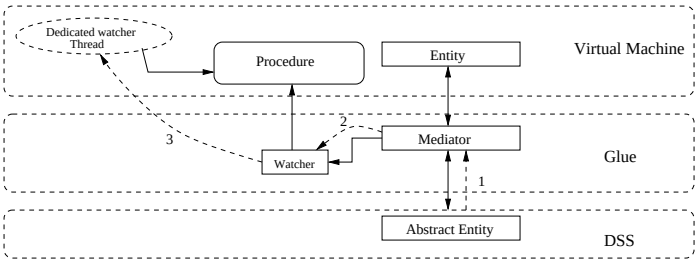

Figure 6: A watcher is "fired". The abstract entity reports a fault state to the mediator. The mediator in turn finds a watcher that whose trigger state matches the fault state reported by the abstract entity. A dedicated Oz thread is created to execute the action code of the watcher.

4.3 Customizing Distribution Behavior

The DSS allows for customization of distribution behavior for each abstract entity. The behavior for a distributed entity is defined at globalization, when the a local data structure is made a distributed entity.

One design principle in OzDSS is to keep the number of distributed language entities as small as possible. Only if an entity is referred from more than one process should the entity globalized. Globalization thus takes place when a reference to a local entity is passed over the network the first time. Since the user has little or no control over exactly when a language entity is globalized custom distribution behavior for a given entity must be assigned an entity before the entity is actually globalized. The custom distribution behavior, , is stored in a hash table that maps entity memory addresses to custom values.

When a language entity is globalized, a lookup is done in the hash table that holds customization information for local language entities. If an entry is found, the value of the entry is used to define the distribution behavior of language entity. Otherwise, if no entry is found, a default value based on entity type is used.

4.4 Reporting Failures

The OzDSS system provides a failure reporting mechanism on the level of single distributed entities. A distributed data structure has a current failure status that describes the access status of the entity. The failure status has three values, similar to the failure information provided by the DSS: *permanent failed*, *temporary failed*, and *no problem*.

The failure model of OzDSS is inspired by the failure model of Mozart. The two models both provide *watchers*, i.e. an action that is executed asynchronously when the failure status of an entity confines to a certain value.

A watcher is installed on a particular entity, in the form of a Oz procedure with a trigger condition. Internally in the glue, the watcher is represented by a `Watcher` object held by the Mediator of the target entity. Figure 6 depicts a distributed entity with one installed watcher. The abstract entity reports a change in fault state (1). The Mediator checks if the fault trigger of the watcher matches the new fault state reported by the abstract entity (2). The watcher is triggered; a new oz level thread is created (3) that execute the procedure held by the watcher. The watcher is said to have "fired", and is removed from the mediator.

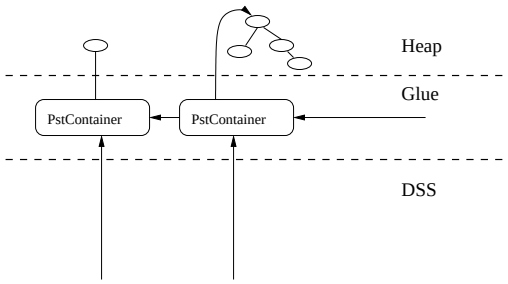

Figure 7:

4.5 Transporting Data

The DSS implements a mode of message transport that allows for late marshaling and suspended marshaling[12]. The Mozart system is well suited for such a model, as will be described in this section, and can thus make use of the benefits the model enables. Late marshaling is when a message is transformed into its serialized format when actually put on the wire. Thus, a message is passed in structured format to the DSS; later, when the message is sent, the glue is asked to serialize the message. Suspendable marshaling is when serializing of a message can be interrupted because of lack of buffer space. This is beneficial if messages are large. Instead of serializing a large message into an even larger buffer (assuming that the serialized format of a message is larger than the structural format), the message is serialized in chunks. To achieve this, the marshaling mechanism must be able to stop in the middle of traversing a data structure and later continue marshaling.

A challange in the late marshaling scheme is that none of the context information available when a message is sent is available when the message is actually marshaled. Context information can for example be type information. Concider a remote object. When a method of the remote object is invoced, information about the type of different arguments of the method is known. If the arguments are stored in a message structure and later marshaled, the context information available at the point of invocation are no longer present. For the marshaler to know how to serialize the message, the type information must be explicitly represented.

Mozart makes use of tagged pointers, given a tagged pointer to a data structure the type of the data structure can be deduced. Furthermore, Mozart provides a generic traverser of oz data structures, used for both marshaling and pickling that takes as argument a tagged pointer and produces a marshaled representation. In addition, the generic marshaler can suspend in the middle of traversing a data structure.

The DSS provides an interface for programming system level data, called PSTC. In the case of OzDSS, two instantiations of the PSTC are implemented. One for outgoing messages, `PstOutContainer` and one for incoming messages `PstInContainer`. Both incoming and outgoing PSTC's holds pointers to language entities on the Oz heap, in the form of tagged pointers. Thus, the containers have no explicit knowledge of what they transport. Figure 7 depicts two `PstOutContainer` objects pointing at their data structures.

The PSTC's are managed by the DSS; it is the DSS that decides when a PSTC can be deleted. The glue is responsible for maintaining the PSTC from the point of creation until the point of destruction. Destruction is decided by the DSS.

4.6 Garbage Collection

Insert a figure of all the roots and alike

The Glue has a dual role during garbage collection of the centralized system. It is a root for the centralized garbage collector and in the same time a subject to garbage collection. In addition the DSS must be garbage collected in order to free internal resources.

Garbage collection of the glue is divided into two steps. First, the root set from the glue is calculated and all pointers are followed. Second, the glue sweeps all entity instances connected to abstract entities. Any language entity instance not member of the root set of the glue nor found by the complete root set of the programming system is removed. The complete root set is defined as the union of the roots from the glue and the roots from the programming system.

4.6.1 Calculating the Root Set

The contents of the PstContainers and the – on distributed operations – suspended threads are roots for the local garbage collector. Furthermore, some distributed entities can be roots for the local garbage collector, defined by the language entity instance's associated abstract entity.

The PstContainers and the suspended threads are organized in linked lists. Finding the roots from the former two sets is done by traversing the lists. Eventual root status of a distributed entity is defined by the abstract entity.

4.6.2 Dropping Unreferred Distributed Entities

Any distributed language entity not found when following the root set of the centralized engine and the root set of the glue is subject to removal. The local garbage collector calls the `engGC` method of the mediators of those distributed language entities that are found to be live. The purpose is to make all the watchers installed on Mediator roots for the garbage collector.

When the local garbage collection is over, and all potential roots have been followed, the Glue is called to remove unmarked distributed language entities. The list of Mediators is traversed in order to find the entities that can be deleted and localized. Below is the method that calculates whether a Mediator be remoed.

```
bool
Mediator::removeMediator(){
    DSS_GC status = getCoordAssInterface()->getDssDGCStatus();
    switch(status){
    case DSS_GC_LOCALIZE:
      if (hasLocalGCStatus()){
          localize();
      }
      return true;
   case DSS_GC_NONE:
      if (hasLocalGCStatus()){
        return false;
      }else{
        return true;
      }
    case DSS_GC_WEAK:
      if (!(hasLocalGCStatus())){ // Try to remove weak
        getCoordAssInterface()->clearWeakRoot();
      }
      return false;
    case DSS_GC_PRIMARY:
      return false;
    }
}
```

Note that a language entity is only localized if the abstract entity has `DSS_GC_LOCALIZE` as root status and the entity is found by the local garbage collector. If the entity is not

referred from the heap, the Mediator is simply removed. Furthermore, an abstract entity can report a weak status that implies that that the entity instance is, temporary, used as a repository, e.g. the entity holds the state when using a mobile state protocol. The weak status prevents removal of a language entity instance, if the language entity is not found by the local garbage collector the mediator tries to actively remove the weak status.

4.7 I/O Handling

Mozart implements a generic IO-handler that provide an interface based on file descriptors, similar to the sockets interface. However, the model is event driven and controlled by the runtime system of Mozart. There is one thread of execution that either executes language threads or the IO. The consequence is that there is no interleaving between the IO and the threads. The OzDSS implements a specialized communication-component that is adapted to the generic IO-handler of Mozart.

5 Green Threads vs. Native Threads

Basically, there are two approaches to implementing concurrency in programming systems. One approach is to use the thread support provided by the operating system, sometimes called native threads. Another approach is to develop a dedicated runtime system that supports for concurrent threads, called green threads. Native threads are easy to use; a thread is similar to a process. The drawbacks are little or no control over thread scheduling and a heavy-weight framework. Green threads potentially give total control over scheduling, and can be made very light weight. The major drawback of the green approach is that the threads are restricted in what they can execute. For example, a Mozart thread can only be preempted or suspended while executing byte-code instructions, not while executing ordinary C++ code.

In parallel to the development of the OzDSS system, a distributed C++ library has been developed. The library, called the Distributed Entity Library (DEL), supports a set of distributable objects with semantics that resembles the basic types of Mozart, i.e. ports, variable, atoms, and cells. Concurrency in DEL is solved using POSIX thread, i.e. native threads. Each user level thread is represented by a POSIX thread. In addition, the communication component is managed by a dedicated POSIX thread.

The DSS is not thread safe and must be protected from concurrent access by multiple threads. The solution is to ensure single thread access of the DSS by a lock. Figure 8 depicts how the DSS is guarded by a lock, shown as the dotted black line that encapsulated the DSS box.

However, of more interest is how the programming system level threads are controlled by the DSS, below is the code for a distributed operation on a distributed cell in C++DSS:

```
MapBaseType*
CellMediator::write(MapBaseType* msg){
  PstOutContainerInterface** pstout = NULL;
  g_mcu->getDSS(); // single access to the DSS starts
  PThreadMediator* th = g_mcu->m_getThreadMediator();
  OpRetVal cont = a_ae->abstractOperation_Write(th->m_getThread(), pstout);
  if (pstout != NULL){
    *(pstout) = new SuspendablePstOut(msg);
  }
  g_mcu->retDSS(); // single access to the DSS ends
  switch(cont){
  case DSS_SUSPEND:
  th->m_suspend(); // here is the thread suspended
  if(th->m_getState() == TREMOTE)
      return th->m_getResult();
```

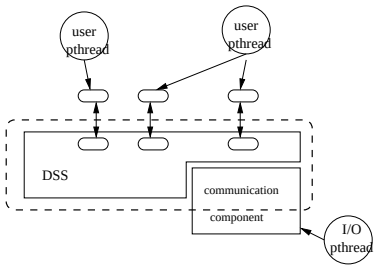

Figure 8: A schematic picture of an application that make use of the C++DSS library. Two threads, denoted *user pthread* manipulate distributed data structures. A third thread, denoted *I/O pthread*. The challange of this system compared to the OzDSS system is that the threads can be preempted while invocing the DSS. Thus, since the DSS is not thread safe, a lock ensures that only one thread at a time can access the DSS, depicted by the dashed line that encompasses the DS.

```
    // otherwise do a local write
  case DSS_PROCEED:
    return a_cell->write(msg);
    break;
  }
}
```

Single access to the DSS is guaranteed between the call `g_mcu->getDSS` and `g_mcu->retDSS`. The `pthread` is associated with a mediator that holds a reference to the global thread identity of the thread. Note how simple the suspension is handled. The call to `m_suspend` suspends the pthread. When the call returns the suspension is over, and the thread mediator is queried for how to continue. Either the result of the operation is held by the mediator or the pthread does the operation locally. This should be compared to the complicated structures required to handle the green threads of Mozart (see Section 4.2).

An observation is that supporting distribution for a native-thread system seems simpler than for a green-thread system. However, the lack of control over the scheduling of a native-thread system requires careful design of data access. The DSS is not thread safe. If more than one thread would execute methods on the DSS or any of the DSS related objects (abstract entities etc.) the result is undefined.

6 Discussion

The development of OzDSS started summer 2002 and continued for approximately one year. Two persons where involved in the development and both spent minor part of their time on development of the system. A running system was produced with surprisingly little effort, especially in the light of the effort required to implement the distribution support for Mozart.

OzDSS is a fully functional prototype. It has been used to execute distributed applications spanning more than 20 nodes and has proven to be reasonably stable. Naturally OzDSS cannot be compared to Mozart when it comes to stability, Mozart is a product while OzDSS is a prototype.

Replacing the integrated distribution support of Mozart with generic distribution support provided by the DSS was educating. Many conclusion where drawn from the

process. In this section some of the design choices are discussed. The resulting system, OzDSS, is compared with repect to efficiency with other distributed systems. Special focus is on comparing OzDSS with Mozart.

6.1 Where to Place the Guard

Given our limited knowledge of the implementation of the Mozart virtual machine, we chose to locate the guard within the atomic operation on an Oz entity. Given the green-thread model of Mozart, this choice of guard location required duplication of code. Each different suspended operation required a special handler in the case that the thread is resumed and told to do the operation locally (see Section **??**).

Another possible solution is insert a *guard-instruction* at byte-code level. Before any byte-code instruction that operates on an entity a special instruction is inserted. The special instruction calls the abstract operation and reacts to the result, thus treatment of suspended threads would be straightforward. Currently, a thread can be suspended when performing an operation on a language entity because some of the arguments are not yet determined (i.e. unbound). A special suspension status could be introduced, suspended on abstract operation.

The two latter designs would result in a more elegant system, than the chosen design. However, both designs require major changes to the Mozart virtual machine. In addition, the *guard-instruction* design would require not only changes to the virtual machine, but also to the Oz-compiler. A better match between the thread model of the DSS and the thread model of Mozart can be achieved; however, the development cost is most likely high. Moreover, there is noting that indicates that the resulting system would be more efficient than the design implemented in the OzDSS system.

6.2 Performance Comparision

To give a fairly good estimate of how efficient the OzDSS system is we have used simple test programs, performing remote object invocations, to benchmark the system in comparison to other middleware. These estimates are presented here as a proof-of-concept that despite some inevitable overhead of using the DSS, being language independent and general in design, it is efficiency wise still a viable approach.

The test program used was a small client-server implementation using the distribution primitives offered by each evaluated system. The server side of the test program creates an data structure of defined size that is passed by copy between processes, denotet the *load*. In addition, a language entity that allows for retreival of the first data structure is created. A client program can perform a language operatio on the distributed languagee entity in order to retrieve the *load*. In Erlang, Mozart and OzDSS the distributed language entity was realized by the port type. In Java and C#, remote objects where used.

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

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

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

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

- SUN Java2. The Standard edition SDK, version 1.4.1_01 for Linux, in a pre-compiled rpm package downloadable from suns' java homepage [10]. This implementation will be referred to as Sun in this chapter.
- IBM:s Java2. The JDK for Linux 32-bit xSeries (Intel Compat.) version 1.4.0, available for download in a pre-compiled rpm package, from the IBM homepage [4]. This java implementation will be referred to as IBM for the rest of this chapter.
- Mozart developers version 1.3.0 [3]. Downloaded from CVS repository 2002-06-20. Compiled from source code using standard compilation options.
- Mozart developers version 1.3.0 as the Oz base with the DSS version 1.0 fully coupled. The complete source is not yet publicly available.
- Erlang version OTP R9B-0 [11]. Compiled from source code using standard compiler options.
- .Net Remoting Release version of the .Net framework [9].

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

The choice of two versions of Java was to give an estimate of the differences in distribution behavior between different implementations. The CORBA facilities used,

Language dependant middlewares (RMI) versus General DSS.

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

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

such as the ORB and the idl-to-java generator, were those included in each Java distribution, with the implementation of the test program following the POA model.

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

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

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

6.3 OzDSS vs Mozart

The OzDSS system is fully compatible with the Mozart on the level of Oz code. In addition, the OzDSS system provides more functionality when it comes to distribution than the Mozart system. Mozart provides one distribution pattern for each distributed-able data structure (not all data structures are distributable). OzDSS on the other hand allows for custom distribution pattern on a single language entity basis.

As indicated by the evaluation in Section 6.2 the coupling to the DSS inflicts a small overhead, compared to the closely integrated solution. This is a natural consequence of the indirection caused by the glue and the DSS. However, this is a small price to pay when taking into account the added functionality provided by the OzDSS compared to Mozart. As shown[6], efficient distribution is a result of choosing the right protocol.

Given that the size of the DSS library is excluded, the OzDSS system is approximately 20k lines of code smaller than the Mozart system. The distribution support for Mozart is 31523 lines, and the glue layer for Oz is 12694 lines. Furthermore, the glue layer is mediating between the virtual machine and the DSS and is functionality wise closer to the virtual machine than the DSS. Thus, we argue that it is easier to maintain the OzDSS system. First, the OzDSS system is smaller code wise. Second, the complicated distribution support code is not part of the programming system code

base.

References

[1] Mozart Consortium. http://www.mozart-oz.org.

[2] Mozart Consortium. The mozart system. http://www.mozart-oz.org/.

[3] Mozart Consortium. Mozart developers version 1.3.0, December 2002. Available for download at http://www.mozart-oz.org/download/.

[4] IBM. Java developers kit version 1.4.0, December 2002. Available for download at http://www-106.ibm.com/developerworks/java/jdk/.

[5] E. Klintskog. Internal design of the dss. Technical Report T2004:15, Swedish Institute of Computer Science, 2004.

[6] E. Klintskog, Z. El Banna, P. Brand, and S. Haridi. The design and evaluation of a middleware library for distribution of language entities. In 8^{th} *Asian Computing Conference*, Dec. 2003.

[7] E. Klintskog, Z. El Banna, P. Brand, and S. Haridi. The dss, a middleware library for efficient and transparent distribution of language entities. In *Thirty-seventh Annual HAWAI'I INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES*, Jan. 2004.

[8] E. Klintskog, V. Mesaros, Z. El Banna, P. Brand, and S. Haridi. A peer-to-peer approach to enhance middleware connectivity. In *OPODIS 2003:* 7^{th} *International Conference on Principles of Distributed Systems*, Dec. 2003. To appear.

[9] Microsoft. Microsoft .net sdk with service pack 2, December 2002. .NET Remoting Release version available through Microsoft Developers Network.

[10] Sun Microsystems. Java2 standard edition version 1.4.1, December 2002. Available for download at http://java.sun.com/j2se/1.4.1/.

[11] Erlang OTP. Erlang otp version r9b-0 source code, December 2002. Available for download at http://www.erlang.org/download.html.

[12] Konstantin Popov, Vladimir Vlassov, Per Brand, and Seif Haridi. An efficient marshaling framework for distributed systems. In *PaCT*, 2003.

[13] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.