KTH Microelectronics
and Information Technology

Thread-based Mobility for a Distributed Dataflow Language

Dragan Havelka

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfillment of the requirements for
the degree of Licentiate of Philosophy

The Royal Institute of Technology
Department of Microelectronics and Information Technology
Stockholm

April 2005

TRITA-IMIT-LECS AVH 05:01
ISSN 1651-4076
ISRN KTH / IMIT / LECS / AVH-05 / 01–SE

Printed by Universitetsservice US-AB 2005

Abstract

Strong mobility enables migration of entire computations combining code, data, and execution state (such as stack and program counter) between sites of computation. This is in contrast to weak mobility were migration is confined to just code and data. Strong mobility is essential for many applications where reconstruction of execution states is either difficult or even impossible. Typical application areas are load balancing, reduction of network latency and traffic, and resource-related migration, just to name a few.

This thesis presents a model, programming abstractions, an implementation, and an evaluation of thread-based strong mobility. The model extends a distributed programming model based on automatic synchronization via dataflow variables. The programming abstractions capture various migration scenarios. These scenarios differ in how migration source and destination relate to the site initiating migration. The implementation is based on replication of concurrent lightweight threads between sites controlled by migration managers. The model is implemented in the Mozart programming system. The first version is complete and a work concerning resource rebinding is still in progress.

To Aleksandra

Acknowledgments

I would like to thank my thesis advisors professor Seif Haridi and associate professor Christian Schulte. I am grateful to Seif for giving me an opportunity to do my research, and to Christian for teaching me how to structure ideas and how to put them on the paper. I want to thank Konstantin Popov for his unselfish support, especially for guidance through the land of Oz. I am using this opportunity to express my gratitude to the original team from Universität des Saarlandes, Saarbrücken for inventing and creating the Oz programming language which is foundation of my research.

During the years of study I have spent a lot of time with guys that helped me with their friendship to overcome difficulties. We spent our lunch time talking about life and I shared with them a lot of joy. I would like to mention their names without any specific order: Frej Drejhammar, Sameh El-Ansary, Erik Klintskog, Joe Armstrong, Per Brand, Fredrik Holmgren, Anna Neiderud, Nils Franzén, Mahmoud Rafea, Ali Ghodsi.

Finally, I would like to thank my wife Aleksandra for her unconditional support and love. Without here I would not be here writing this. I also want to thank my mother Ružica for her love, my brother Nikola for being my role model and my old friends Dragan Despot and Zdravko Jovičić for being just that.

Contents

Chapter 1

Introduction

In this thesis we present a model and an implementation of strong mobility based on distributed dataflow computing. We identify a set of essential primitives and abstractions (`Go`, `Pull`, and `Push`) for explicit thread migration. Common programming patterns for mobile applications are presented.

1.1 Code Mobility

Despite the widespread interest in the research community in code mobility there is no widely accepted definition of the code mobility. The most often quoted informal definition is given by Carzaniga, Picco, and Vigna in [8].

> *A capability to dynamically change the bindings between code fragments and location where they are executed.*

We give the following definition: Mobility is the ability to move a *program* between *execution environments* as it is executing. An execution environment is typically a virtual machine running inside an operating system process.

In this thesis we focus on a particular form of mobility namely *thread based mobility* where threads are lightweight and executed inside a virtual machine.

1.2 Mobility Overview

In this section we first give short history of code mobility. Then we proceed by focusing on strong mobility. We show how strong mobility can be modeled and

implemented at different levels, from operating systems to programming languages. We continue by comparing the strong and weak mobility and argument why we choose the explicit migration before the implicit migration.

1.2.1 First Examples of Mobility

Mobility is a concept that goes way back to the 1970's. For example, code mobility has been used for *remote batch job submission* [4]. Postscript also used the mobility of code for printer control.

In the area of distributed operating systems the concept of mobility has been investigated from a different perspective. Here, the focus of research was *process migration* with the goal of achieving transparent load balancing. Thus migration was implicit, hidden from the application programmer and provided only as a part of system internals. Several systems have implemented process migration with more or less success. For example: Xos, V-Systems, DEMOS/MP, LOCUS, Accent, Sprite, Charlotte. In 1988, the Emerald system introduced explicit migration of both passive and active objects (an active object contains a thread of execution as a part of the state). Thus, Emerald provides programming abstraction for migration and in that way introduces a new programming model.

1.2.2 Second Generation of Mobility

With the Internet came the second generation of the code mobility systems. The motivation was not only for load balancing and mobility is not part of an operating system but of a programming system. The systems are virtual machine based and mobility is used for: network traffic reduction, dynamic reconfiguration, mobile agents, disconnected operations, etc. Implementations are mainly based on three different techniques:

- The simplest way to provide code mobility is based on a technique known as *code on demand* or *remote evaluation*. An example of the first approach is used in Java applets.
- The next step was to provide so-called *weak mobility*. This definition is introduced by Vigna et.al. [10] and defines ability of a system to move code and data. Many systems provide weak mobility, for example: Java, Mozart, Obliq, Erlang, and .NET.
- Finally, so-called *strong mobility* [10] is ability of the system to move execution state together with code and data.

In this thesis we focus on a model of strong mobility in a distributed language with dataflow synchronization and an implementation in Mozart/Oz [25].

1.2.3 Strong Mobility

Computation in concurrent systems is organized into multiple threads. On systems supporting weak mobility, a thread is executed on a site and it stays at the same site from creation to termination.

Strong mobility allows thread migration at any execution step between sites by extending a system supporting weak mobility. The time of migration is independent of the execution state of a thread: a thread can be migrated regardless of whether it is runnable or suspended. The fact that migration can occur at any execution step is important. One of the main advantages is that the *migration call* can appear at any place in the code. The same is not true for systems with weak migration where a programmer has to program capturing and recreating of threads and computation state referred to by these threads. The full execution state is not always maintained. That means that migration can be done only at certain points in the program.

There are many benefits of strong mobility, for example: dynamic load balancing between sites in distributed systems achieved by distributed scheduling; network latency avoidance and network traffic reduction by moving communicating components closer to each other; dynamic reconfiguration of distributed applications and failure avoidance; migration to sites with resources required for computation; and as enabling infrastructure for mobile agent platforms.

Strong mobility is known from operating systems theory as the process migration. In our model, the units of migration are threads. Migrating a thread can require the migration of several objects, more precisely the objects referred by the thread. Thread migration is based on *thread replication* by creating an exact copy (a clone) of the original thread at the destination site and destroying the original thread at the source site.

When a thread migrates all data values that are accessed by the thread stack are migrated as well. The exceptions are local resources that are referenced by the migrating thread. We define a *resource* as a data structure whose use is restricted to one site (for example file handles). Here, we assume that resources are ubiquitous and dynamically rebound when threads are migrated. A thread is executed at a *location* which we refer to here as a *site*. Migration is always performed between two sites: a *source site* and a *destination site*. The thread migration implies migration of the computation state consisted of a stack of statements. A statement is a closure defined by a program counter that points to next instruction and environment needed for execution of the instruction.

Thread migration can be initiated from the source site, destination site, or a third site (that is, a site that is neither source nor destination site). On the source site the thread migration can be initiated by the thread itself or by other thread.

1.2.4 Explicit Versus Implicit Migration

One way to classify applications that take advantage of strong mobility is to focus on the use of mobility. Applications come in two major groups: mobile applications (*site-aware* applications) that are specifically written to use migration, and applications that are not *site-aware* but use mobility only to increase performance. These two groups are used in the following discussion to compare explicit and implicit migration.

1.2.5 Implicit Migration

Implicit migration is transparent (invisible) to the application programmer. This means that the programmer does not program thread migration. Thread migration is caused by some external event instead. For example, migration of an active object (an object together with a single thread executing methods) can trigger the migration of the associated thread.

Implicit migration is preferable for load balancing where performance issues should be separated from applications. For example, large-scale simulations with thousands or even millions of threads that are not focused on mobility, but mobility is used for performance reasons.

In the case of mobile applications (mobile agents), thread migration is part of the application and the implicit model is severely limited.

1.2.6 Explicit Migration

Explicit migration is not transparent. The application programmer explicitly "invokes" migration (for example, by using some migration abstraction). Explicit migration guarantees full awareness about where computation is performed.

Applications that are interested in migration due to performance reasons only, still can be completely separated from thread migration issues if appropriate abstractions are provided.

In the case of mobile applications, explicit migration provides abstractions for migration. We restrict our attention to explicit migration as the consequences of migrating a thread are isolated to a single thread and are therefore easy to understand.

In summary, explicit migration with appropriate abstractions can be used for both load balancing and mobile applications. Finally, expressiveness of the explicit migration allows for implementation of the implicit migration with the explicit migration.

1.2.7 Unit of Mobility

Explicit migration is used to migrate an entity which represents sequential flow of computation. What this entity really encapsulates vary between implementations. For example, the unit of mobility can be an active object like in the Emerald system [21], or it can be an operating system process like in Agent-Tcl [13] or it can be a thread as in Sumatra [1].

1.3 Motivation

While the usual argument for strong mobility is load balancing, it is also vital for a broad range of applications: mobile agent platforms, network traffic reduction, fault tolerance, etc. Strong mobility allows the migration of code, data, and execution state (stack and program counter), as opposed to weak migration which requires only migration of code and data. The advantage strong mobility offers over weak mobility is that migration can take place at any time. The application programmer does not have to stop computation, collect the computation's state for migration (which might be a very difficult task), and restart the computation after performing weak migration.

The migration model described here is well adjusted to systems with the following characteristics. Computation is organized into multiple concurrent lightweight threads. Distribution is supported which includes both distribution of data and code (such as procedures, classes, and objects).

1.4 Contributions

This thesis makes the general contribution of a model, programming abstractions, and an implementation architecture for thread-based strong mobility for a distributed dataflow language. More specifically, the contribution are as follows:

Thread-based Mobility. Thesis contributes a model for strong mobility based on threads and dataflow languages. The model is an extension of a well-established model for distributed programming in a concurrent dataflow language. The model requires that mobility is under explicit control of the programmer.

Programming Abstractions and Application Scenarios. The thesis identifies three programming abstractions (`Go`, `Push`, and `Pull`) which capture common programming idioms in the construction of mobile applications. We also show how the abstractions can be used in prototypical application scenarios.

Implementation Architecture. All programming abstractions are implemented on top of a primitive for thread mobility. This thesis identifies thread replication together with migration managers as basic building blocks for an architecture to implement thread-based mobility.

1.5 Approach

In this thesis we have taken the following approach: Strong mobility is made explicit. Thus, we introduce a set of useful programming patterns together with corresponding migration abstractions.

Foundation of the migration model is based on *mobile threads*, *migration managers*, and *sites*.

Mobile Threads. The unit of mobility is a mobile thread which is an abstract data type encapsulating: a thread, a site, and a record of resources used by the thread.

Migration Managers. Migration managers control migration of mobile threads. Migration is performed by sending messages between migration managers at the sender and receiver site.

Sites. A site is communication channel used for thread migration. It accepts only the migration messages and there is only one site per process.

Mozart/Oz. Strong mobility is integrated into the Oz programming language. The essential features of Mozart / Oz that make the implementation of strong mobility possible are explicit concurrency, lightweight threads, automatic synchronization through dataflow variables, transparent distribution of data structures (stateless, stateful, and single-assignment).

1.6 Source Material

Part of this thesis material has been published in the following internationally peer-reviewed article:

- Dragan Havelka, Christian Schulte, Per Brand, Seif Haridi. Thread-based Mobility in Oz. In *Proceedings of Multiparadigm Programming in Mozart/Oz: Second International Conference*, volume 3389 of *Lecture Notes in Computer*

Science, pages 137-149, Charleroi, Belgium, October 7-8, 2004. Springer-Verlag [18].

1.7 Outline

The next chapter gives short introduction of the Mozart programming system which is used as the implementation platform. Chapter 3 identifies migration abstractions by presenting several programming patterns. Chapter 4 presents the migration model together with the migration abstractions and the data types used by the model. It presents a migration primitive as foundation for the migration abstractions and how thread states are reflected during migration. An implementation of the model in the Mozart programming system is sketched in Chapter 5 followed by an evaluation in Chapter 6. Related work is discussed in Chapter 7. The thesis concludes with Chapter 8.

Chapter 2

Short Introduction to Oz

2.1 General System Prerequisites

In this thesis we assume that programs execute concurrently by executing typically many lightweight threads. Threads synchronize automatically by using dataflow variables (also known as logic variables). Dataflow variables serve as place-holders for not yet known values. Threads are assumed to be first-class language entities in that they can be passed as arguments to procedures, stored in data structures, and so on.

A thread is a stack of statements. It executes by trying to execute its topmost statement on the stack. A thread automatically suspends if its topmost statement suspends due to insufficient information available on its dataflow variables. Thread resumption again is automatic: providing the value for a variable automatically and fairly resumes all threads suspending on this variable. For more details on our model of computation see [29].

We also assume that execution can be distributed across several sites of computation: both data structures as well as code (in form of procedures, objects and their attached classes) are distributed. For more details on our model of distribution see [32, 16, 17].

2.2 Properties for Strong Mobility

Strong mobility requires several properties from the system of which the most important ones are:

- **Concurrency.** Without concurrency strong mobility can be achieved only

at operating systems level as process mobility and that is not *fine-grained strong mobility* which is our goal. The requirement is even stronger. Thus, it is preferable that a system provides *lightweight concurrency* with the small overhead.

- **Communication primitives.** A distributed system must provide some communication primitives to transfer data, code, and execution state between machines.

- **Dynamic loading.** A system must provide a way to dynamically load libraries or modules to solve dependencies created by migration.

- **Distribution.** Strong mobility includes migration of data structures and code and the system must provide the methods for serialization / unserialization of data structures and code.

- **Globalization.** The globalization is very important property of distributed systems to achieve global uniqueness of entities and is also important for an efficient implementation of strong mobility.

The Mozart system provides all properties listed above and even more, it provides them in such way that implementation comes as a natural extension of the system with minor issues that were more or less straight-forward to overcome. In the following sections we describe the Mozart system in more detail.

2.3 Oz and Mozart

Oz is a multi-paradigm concurrent dynamically-typed language with dataflow synchronization. Concurrency in Oz is explicit and threads are first-class entities. Mozart is a network transparent distributed programming system implementing Oz [25]. Thus, a distributed application can be developed completely in a centralized setting [17]. Oz provides a variety of built-in data types: stateless, stateful and single-assignment. Oz entities can be distributed between Mozart processes. Distribution of stateless data entities is achieved by copying (replication). Consistency of distributed stateful entities is implemented by distribution protocols [32, 16]. When a data entity is sent between two Mozart processes, its memory representation is converted to a network representation at the source site and the memory representation is created at the destination site upon the reception. Translation to network representation is called *marshaling* and translation back is called *unmarshaling*. The term *serialization* is also used.

σ	::= **skip**	*empty statement*
	| `X = Y` | `X = V`	*tell statement*
	| $\sigma_1\ \sigma_2$	*sequential composition*
	| **proc** {`X LV`} σ **end**	*procedure creation*
	| {`X LV`}	*procedure application*
	| **local** `X` **in** σ **end**	*declaration*
	| **if** `X` **then** σ_1 **else** σ_2 **end**	*conditional statement*
	| **case** `X` **of** V_1 **then** σ_1 `[]` V_2 **then** σ_2 **end**	*pattern matching*
	| **thread** σ **end**	*thread creation*
	| **for** `I` **in** `X..Y` **do** σ **end**	*for loop*
`V`	::= `S`	*simple value*
	| l($X_1 \ldots X_n$)	*tuple construction*
	| l(f_1:$X_1 \ldots f_n$:X_n)	*record construction*
`S`	::= `l` | *integer*	*literal or integer*
`l`	::= *atom* | `true` | `false`	*atom and names*
`X, Y, Z`	::= *variable*	*variable*
`LV`	::= ϵ | `X LV`	*list of variables*

Figure 2.1: Basic Statements of Oz

2.4 Oz Syntax

In this thesis we use code fragments written in Oz to present the programming patterns using strong mobility and to present a part of implementation. To help a reader we show a subset of the Oz statements in Figure 2.1.

2.5 Concurrency

The Mozart runtime system is implemented in C++ and runs as a single-threaded operating system process. The Mozart engine controls the execution of concurrent threads. The scheduler is responsible for the fair and preemptive scheduling of threads. In the Mozart system a thread is spawned by:

thread σ **end**

where σ is any valid Oz statement. The forked thread runs concurrently with the current thread which has executed this statement. The current thread resumes immediately with the next statement. The Mozart system provides a rich set of operations on the first class threads. These are summarized in Table 2.1.

Table 2.1: Operations on threads in the Mozart system

{Thread.is Thr Bool}	Tests whether Thr is a thread.
{Thread.this Thr}	Returns the current thread.
{Thread.state Thr State}	Returns current state of Thr.
{Thread.resume Thr}	Resumes Thr.
{Thread.suspend Thr}	Suspends Thr.
{Thread.isSuspended Thr Bool}	Tests whether Thr is currently suspended.
{Thread.injectException Thr X}	Raises X as exception on Thr.
{Thread.getPriority Thr Prio}	Returns the current priority of Thread.
{Thread.setPriority Thr Prio}	Sets Thr's priority to Prio.
{Thread.preempt Thr}	Preempts Thr.

2.6 Distribution

Mozart provides network transparent distribution. An application can run on a network of computers as on a single computer. Data entities in Mozart can be shared between sites. Stateless entities are shared by replication. Stateful entities are shared by using distributed protocols. Thus, an operation on a distributed stateful entity implies activation of the entity specific distribution protocol and exchange of messages between the involved sites. These messages can contain Oz values. Before values are sent they are transformed to network a representation, *marshaled*. Marshaling is triggered by the distribution subsystem and is not directly exposed to the application layer. Thus, the system does not provide the programming abstractions for marshaling and unmarshaling. It provides abstractions for pickling of *stateless data structures*. Pickling is closely related to marshaling and is used for saving of data structures to persistent storage but it cannot be used for the stateful data structures.

The part of the system responsible for distribution is called distribution subsystem.

2.6.1 Distributed Subsystem - *DSS*

The distribution subsystem, *DSS*, uses the notion of an abstract entity to coordinate operations performed on distributed entities. Thus, shared entities are not accessed directly. Instead operations are redirected to the corresponding abstract entity. Abstract entities provide a generic interface to a set of consistency protocols of which at least one is needed per entity. An abstract entity interacts with a language entity by using abstract operations. These operations are used for example: to retrieve state of the distributed stateful entity, to delegate execution of the language entity operation, and to resume thread that previously suspended on the abstract operation. Detailed description of *DSS* can be found in [23, 22].

Chapter 3

Programming Patterns

This chapter introduces some common programming patterns for mobile applications and identifies migration abstractions and primitives.

3.1 Go: Self Migration

The abstraction Go is useful for proactive mobile agents (that is, agents which initiate their migration in anticipation of future problems, requirements, or changes).

Consider as an example: a mobile agent MA moves between sites and collects as well as offers information:

1. MA collects information about computing resources such as: processor power, amount of available memory, available software components and libraries, and available external hardware resources.

2. MA offers information collected on already visited sites to the local agents (that is, the agents that are located on the visiting site).

3. MA gets a list of neighbor sites and chooses one of them for migration.

4. MA performs migration.

5. MA repeats the outlined execution.

A code example for MA is as follows:

```
proc {Collector Info ThisSite}
   ListOfNeigh NextSite SiteInfo UpdatedInfo
in
   SiteInfo = {CollectInfo}
   {OfferInfo Info}
   UpdatedInfo = {UdateInfo Info ThisSite SiteInfo}
   ListOfNeigh = {GetNeigh}
   NextSite = {ChooseNext ListOfNeigh}
   /* Here comes the migration      */
   {Go NextSite}
   /* Executes on site 'NextSite' */
   /* after migration has finished  */
   {Collector UpdatedInfo NextSite}
end
```

MA is started by spawning a thread which calls `Collector` appropriately:

```
thread {Collector StartInfo CurrentSite} end
```

Please note that the `Go` abstraction has one argument representing the destination site.

3.2 `Pull`: Execution Locator

The abstraction `Pull` is used to move execution from a *source site* to a *destination site*, and is invoked from the destination site. It is useful for several reasons: traffic reduction, network latency avoidance, and other resource-related issues.

An example of use in the case of traffic reduction can be implemented in the following way. A procedure `TrafficController` takes a list of remote threads and checks for each thread if it is worth moving. The decision is made based on specified criteria (for example, measuring the amount of network-traffic produced by the thread). This can be programmed ad follows:

```
proc {TrafficController RemoteThreads}
   for T in RemoteThreads do
      if {WorthMoving T} then
         {Pull T}
      end
   end
end
```

Note that the `Pull` abstraction has one argument representing the thread to be pulled to the current site. Note that this is in contrast to `Go` which takes the

site as its argument.

3.3 `Push`: Execution Mediator

The abstraction `Push` is used to *mediate execution* between sites. An example of use is dynamic load balancing. For example: A *distributed scheduler (DS)* has access to a list of thread queues with one queue per involved site. The goal is to optimize performance by moving threads from heavily loaded sites to less loaded sites. The corresponding code example is presented below:

```
proc {LoadBalance SiteList}
   LoadList
   HighestLoadSite LowestLoadSite
   Thr
in
   LoadList = {GetLoads SiteList}
   HighestLoadSite = {Max LoadList}
   LowestLoadSite  = {Min LoadList}
   Thr = {ChooseThread HighestLoadSite}
   {Push Thr LowestLoadSite}
end
```

Note that the `Push` abstraction takes two arguments, the thread to be migrated and the destination site. It can be invoked from any site including the source and destination sites.

3.4 Summary

In this chapter we have identified three migration abstractions by presenting several programming patterns. The abstractions and their usage are summarized in Table 3.1.

The abstractions `Go`, `Pull`, and `Push` cover all possible cases for initiating explicit migration. They are based on a primitive which

- on the source site:
  - captures the thread's execution state
  - serializes the thread (builds a network representation of the thread)
  - sends the serialized thread to the destination site
- on the destination site:

Table 3.1: Migration Abstractions

	Invoked at:	**Called from:**	**Programming patterns:**
{Go Site}	source site	thread itself, proactive	self migration: *mobile agents*
{Pull MT}	destination site	another thread (reactive, forced)	execution attractor: *traffic reduction, network latency avoidance, resource related migration*
{Push MT S}	any site	another thread (reactive, forced)	execution mediator: *load balancing, distributed scheduling*

– rebuilds the thread on the destination site.

The use of thread and site references is summarized in Table 3.2. All abstractions have in common that they require representations of both threads and sites in the programming language. It is important to notice that both `Push` and `Pull` can be used to achieve implicit migration. Actually that is what the presented examples show only from the perspective of the threads which are migrated.

Table 3.2: Use of thread and site references

	Site reference	Thread reference
Go	yes	no
Pull	no	yes
Push	yes	yes

Chapter 4

Thread Migration Model

In this chapter we present the thread migration model. First, we give a general overview of the model. Then we introduce the used data types, the migration abstractions and the migration managers.

4.1 Overview

The model rests on three basic pillars: migration abstractions, migration managers and thread migration.

Migration abstractions: The main goal of the abstractions is to provide meaningful tools for strong mobility to the application programmer. The abstractions are identified in such way that they cover all interesting programming patterns with mobility in focus. The main guidelines for the abstraction selection process are the following:

- How the migratory thread relates to the thread which initiates migration.
- How thread migration is related to the source and destination sites.

Migration manager: Migration managers coordinate migration of threads. There is one migration manager per site and thread migration is performed by sending messages between migration managers at the sender and receiver site.

Thread migration: Thread migration is based on thread replication. Thus, a copy of the thread is created at the receiver site and the original thread is destroyed.

4.2 Basic Data Types

The model is based on two data types: *Site* and *Mobile Thread*.

4.2.1 Site

A site is communication channel used for thread migration. It accepts only the migration messages. There is only one site per process and a reference to a local site is received by calling the abstraction `GetSite`.

4.2.2 Mobile Thread

Mobile thread is an abstract data type consisted of three data types: a thread `T`, a record `RS`, and a site `Site`.

- `T` is a stack of statements which can execute only if its top statement can be executed. The execution of the statement removes the the statement from the stack and additionally can:
  - Change the store.
  - Push new statement on the stack.
  - Create a new thread.
- `RS` holds references to local resources used by the thread `T`.
- `Site` is a reference to a site where the thread `T` executes.

4.3 The Model

Thread migration, independent of the abstractions, is based on replication (that is, creation of an exact copy of the thread at the destination site). All abstractions discussed earlier use thread replication as follows: after the replica has been created, the *original* thread is destroyed.

Each site runs a *migration manager* which controls migration of mobile threads. *MobileThread* migration is done by sending and receiving messages between migration managers. The migration managers use *sites* as communication channels.

The information needed to perform migration of a mobile thread MT is located at the source site of MT and the replication process is started there.

In the following MM_s refers to the migration manager of the source site, whereas MM_d refers to the migration manager at the destination site. T refers to the thread that belongs to MT.

To migrate a thread we proceed as follows:

At the source site the thread T is *suspended*, its execution state is collected, serialized, and sent to the destination site together with the list of resources used by T.

The migration manager MM_s waits until an acknowledgment of thread reception issued by the migration manager MM_d is received. The acknowledgment confirms the existence of two copies of T, the *original* thread at the source site and the *replicated* thread at the destination site. Then, the *original* is *terminated* and the *replica* is resumed. An exception is raised on the thread T if:

- The reply from MM_d does not arrive in the given time frame (that is, before a timeout).
- The received reply informs MM_S that the migration failed due to *unmarshaling* issues (for example unresolved resource references).

At the destination site, when the serialized thread is received by MM_d, it rebuilds the thread T_r from the network representation (that is is *unmarshals* the thread). MM_d updates MT's attribute `site` with a reference to the new local site (that is the site where MM_d resides). After rebuilding, an acknowledgment message is sent to MM_s and the replica T_r is *resumed*. MM_d rebinds all resources used by T_r. The acknowledgment message is `success` or `failure` depended on the result of rebinding of resources.

MM_s and MM_d synchronize once during thread migration. Figure 4.1 shows the interaction and the synchronization between migration managers.

A code example of thread replication is shown below:

```
fun {ThreadReplicate MobThr Site}
   Reply MarshThr
   Timeout = 5000
   Return
in
   {MobThr getThread(?Thr)}
   {Thread.suspend Thr}
   /* Send serialized mobile thread and */
   /* synchronization variable */
```

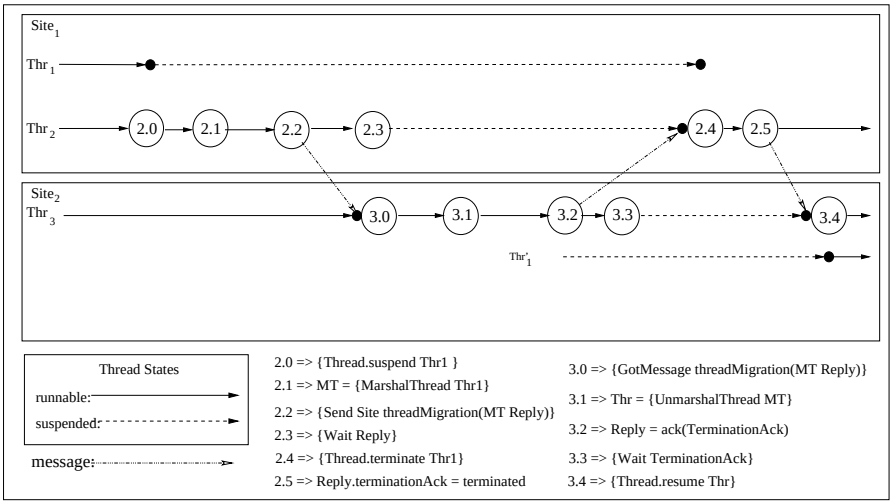

Figure 4.1: Execution of $\{ThreadReplicate\ Thr_1\ Site_2\}$

```
   {SendThread Site threadMigration(MobThr ?Reply)}
   /* Wait with timeout on acknowledgment */
   {WaitOr Reply Timeout}
   if {IsDet Reply} then
      if Reply == failure then
         /* Raise exception and terminate thread */
         {Thread.injectException Thr migration_failed}
         Return = failed
      else
         /* Terminate thread */
         {Thread.terminate Thr}
         Return = success
      end
   else
      /* Raise exception and terminate thread */
      {Thread.injectException Thr migration_failed}
      Return = failed
   end
   /* Inform the initiator thread if the migration */
   /* succeeded or not. */
   Return
end
```

A code example of a migration manager is presented below:

```
proc {WaitForThreads S}
   for Msg in {GetMessage S} do
      case Msg
      of threadMigration(MobThr Reply) then
         Thr
         LocalSite
      in
         LocalSite = {GetSite}
         {MobThr getThread(?Thr)}
         /* Update the current site of MobThr */
         {MobThr putSite(LocalSite)}
         /* Try to rebind resources */
         if {RebindResources MobThr} then
            /* In the case of success resume the thread */
            Reply = success
            {Thread.resume Thr}
         else
            /* In the case of failure terminate the thread */
            Reply = failure
            {Thread.terminate Thr}
         end

      [] migrate(MobThr Site Result) then
         MobThrSite LocalSite
      in
         LocalSite = {GetSite}
         {MobThr getSite(?MobThrSite)}
         /* Check if MobThr is local thread. If thread */
         /* has already moved then forward the request */
         if LocalSite \= MobThrSite then
            {Send MobThrSite Msg}
         else
         /* If migration is successful Result is bound */
         /* to 'success' otherwise to 'failed' */
            Result = {ThreadReplicate MobThr  Site}
         end
      end
   end
end
```

4.3.1 Migration and Thread States

A thread can be in one of the following states: *runnable*, *running*, *suspended*, or *terminated*. Thread migration can be requested for a thread which is in any of the above mentioned states. The thread state after migration must remain the same. The base case is the migration of the suspended thread. The detailed behavior description for each case follows:

Suspended Thread: This case is slightly more involved and exploits certain invariants on dataflow synchronization. In a language with dataflow variables a thread suspends if its topmost statement cannot execute due to yet unbound dataflow variables.

The migration manager adds a migration specific suspension to the thread. This locks the thread as it prevents unplanned resumption of the thread. The thread is *replicated* and during the process all variables on the thread stack are discovered and distributed according to their distribution protocols.

The *original* thread is *terminated* and the *replicated* thread is *resumed*. When the thread is scheduled to run at the *destination* site it suspends on the same statement that caused suspension at the *source* site. Thus, the thread rediscovers its suspensions on its own. This allows to maintain suspension on dataflow variables locally (that is, suspension information is not distributed across the net). This property is a direct consequence of using dataflow variables for distributed computing [16].

Running Thread: A running thread cannot be migrated directly. There is only one running thread at each site and a thread cannot migrate itself. Thus, a thread which wants to migrate itself delegates its migration to another thread. The thread to be migrated is stopped first, and another thread (the thread executing the migration manager) performs *migration*.

Runnable Thread: A runnable thread waits in a runnable queue to be scheduled for execution. The migration manager suspends the thread and performs the migration. The *original* thread at the *source* site is *terminated*, and the *replicated* thread which was created in the *suspended* state is *resumed* (that is, added to the runnable queue at the *destination* site).

Terminated Thread. A terminated thread has no stack and it can not be runnable again. Thus, the migration is not meaningful and the thread that requested migration is properly informed.

4.3.2 Failure Model

A failure can occur under any step of the migration protocol and some of these are specially interesting. For example the thread can be marshaled at the sender site and sent to the receiver site, but due to some unknown reasons no reply arrives back from the receiver site. In that case we have several possibilities:

- Thread has been received and unmarshaled by the migration manager at the receiver site and it has sent a reply but the reply has been lost on the way due to network failure.
- Thread has never been received by the migration manager at the receiver site.

We cannot be sure which of these two has occurred. Our approach is to always terminate the original thread. This way we guarantee the global existence of at most one copy of the thread.

4.3.3 Request Forwarding

Another issue that must be handled is the case when several migration requests are incoming from different sites for the same thread at the same time. We are using asynchronous communication for implementation of the protocol and several migration requests for the same thread can be already in "the pipe" waiting. Thus, all other requests except the first one are invalid. To solve this we use the following approach: the migration manager does additional check on the thread when the migration request is received and if the current site of the thread is not the local one then the request is *forwarded* to the accurate migration manager which is situated at the same site as the thread.

4.4 Programming the Abstractions

With the help of migration managers and thread replication as discussed above, the abstractions introduced in Chapter 3 are implemented in the following way:

4.4.1 `Push` Abstraction

The `Push` abstraction can be used from any site including the source site and destination site. The *distributed scheduler* presented in the previous chapter uses `Push` to migrate a thread that is not at the same site as the scheduler. Thus, we cannot assume that the `Push` abstraction is used at same site where thread is currently located.

```
proc {Push MobThr Site}
   ThrHomeSite Result
in
   {MobThr getSite(?ThrHomeSite)}
   {Send ThrHomeSite migrate(MobThr Site ?Result)}
   {Wait Result}
   if Result == failed then
      raise threadMigrationException end
   end
end
```

4.4.2 Go Abstraction

A special case of Push is when the thread itself initiates the migration process. This operation is called Go. The implementation of Go on top of Push is presented below. Note that Go is a method of a class and not procedure or a function. This is due to the fact that Mobile Thread is an object and Go is used for proactive migration:

```
meth go(RemoteSite}
   Sync Thr
in
   {self getThread(?Thr)}
   thread
      {Thread.suspend Thr}
      Sync = done
      {Push self RemoteSite}
   end
   {Wait Sync}
end
```

Here, the thread to be migrated spawns a new thread that performs migration. The thread running the Go abstraction blocks on the Sync dataflow variable used to synchronize on migration. That is, Go will return when the migration process is finished. Note that the thread is *suspended* and the Sync variable is bound before Push is called. That means that the Go does synchronize on completion of migration.

4.4.3 `Pull` Abstraction

The `Pull` abstraction is implemented on top of the `Push` abstraction. The implementation is presented below:

```
proc {Pull MobThr}
   MySite
in
   MySite = {GetSite}
   {Push Thr MySite}
end
```

Here, `MySite` is the *destination* site where `MobThr` migrates, that is `MobThr` is *pulled* to `MySite`.

4.4.4 Thread References and Thread Migration Protocol

When a thread is created, it is known only to computations at the local site. Later on, a thread reference can be passed to other sites and used to perform network-wide operations on threads. These operations are network transparent and their semantics remain the same as if they where local operations. The network transparency is provided by the underlying system.

It is possible that a thread during its lifetime migrates several times between sites. Computation on the sites that have access to the thread reference must have accurate information about the thread's current site. One way to keep this information up-to-date is to provide a migration protocol.

In our model [31] we assume that the system provides a distribution protocol for objects. Thus, we implement our abstraction `MobileThread` as an object. The migration protocol performed between migration managers keeps the reference to the current site updated.

The protocol that we present here is manager-proxy based.

When a thread reference is sent for the first time to a remote site (for example, when a thread is distributed) a *thread manager* on the *home site* is created. *Home site* is the site where thread execution happens. In addition, when the thread reference arrives at the remote site, a *thread proxy* is created (see Figure 4.2). When `Pull` is executed on the remote site, the thread proxy on that site is triggered and an appropriate protocol message is sent to the manager. When the manager receives the message it initiates thread migration:

- The thread is suspended, and its state is captured and sent to the new home site.
- A new *thread manager* is created on the destination site.

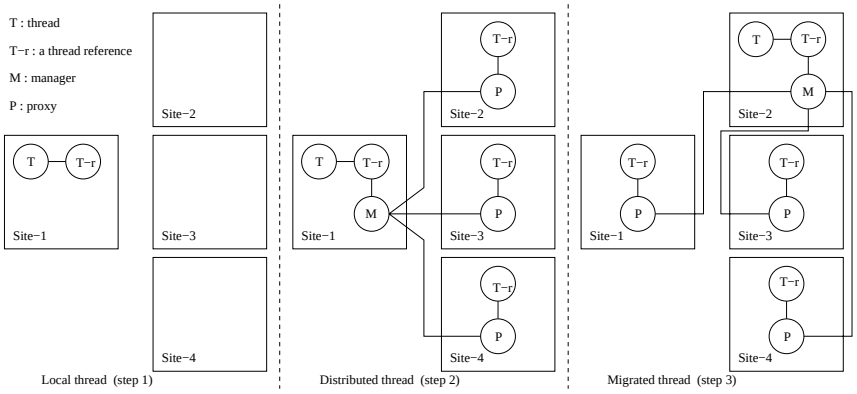

Figure 4.2: Thread References and Migration

- All sites that have thread reference are informed about the new home site and the new manager.

4.5 Summary

In this chapter we have presented the thread migration model based on migration managers, sites, and thread replication. We have also introduced two data types: *Site* and *Mobile Thread*, where a site is a communication channel used for the thread migration and a mobile thread is an *unit of mobility*. It is an abstract data type consisted of a thread, a record of used resources, and a site reference. We have also discussed how thread state affects thread migration and how simultaneous migration requests are handled. In the failure model we guarantee existence of at most one copy of the thread. Finally, we have shown how the migration abstractions: `Push`, `Pull`, and `Go` can be programmed in Oz based on the model.

Chapter 5

Implementation

In this chapter we describe implementation of strong mobility in Mozart. The most relevant properties of the Mozart system concerning the implementation are described and necessary modifications are identified.

We describe the concurrent model in Mozart and present important implementation details about threads. We describe how serialization (marshaling / unmarshaling) is implemented in Mozart and how the system has been extended to support serialization of threads. We proceed with implementation of extensions needed for the replication primitive and the implementation of the migration managers.

5.1 Threads

The Mozart virtual machine (MVM) is a register based machine inspired by the WAM [3]. Instructions use three sets of registers for referring to values:

- **X registers** which VM provides for threads in the same way as an operating system provides registers for processes.
- **Y registers** which are allocated per procedure call and are used for local variables.
- **G registers** which are allocated per procedure definition.

A thread in Mozart consists of a stack of tasks, an identification tag *id* and a priority level descriptor *flags* (see Figure 5.1). The only way to get a thread's id is for the thread itself to call `ThrRef={Thread.this}`. The attributes *id* and *flags*

are represented as integers and a task stack is represented as C++ class. Tasks on the task stack are executed sequentially following a stack discipline. A task is a closure consisting of a triple (*PC*, *Y*, *G*): *PC* is the address of the next instruction, *Y* is a local environment with a number of registers, and *G* is a reference to the current procedure. Detailed information about the Mozart virtual machine can be found in [28, 24].

```
class Thread
{
  int id;
  int flags;
  TaskStack *thrStack;
  // methods...
  // ...
}
```

```
class TaskStack
{
  StackEntry *tos;
  StackEntry *stackEnd;
  // methods
  int getNrOfTasks();
  // ...
}
```

Figure 5.1: Thread in the Mozart system

5.2 Scheduling

The scheduler controls the thread preemption and guarantees fairness among all runnable threads. The runnable threads are stored in a queue and according to the round-robin policy selected for execution. A preemption of a thread cannot occur during the execution of an instruction. It can occur when a new task is pushed onto the stack (for example, when a procedure is called) and when a task is popped from the stack to be evaluated by the engine.

This preemption scheme has two important properties:

- The size of the state of the run-time system which has to be saved and restored due to thread preemption is small.
- A strong invariant for atomic operation is provided because the execution of a task is never interrupted.

These properties facilitates an efficient implementation of strong mobility: The first property minimizes the size of the data that has to be transferred, and the second property simplifies the implementation.

5.3 Thread Tasks

Thread tasks are divided in three major groups:

1. **A continuation task:** A continuation task, *(PC, Y, G)*, is a closure starting at the address *PC*. *Y* and *G* are the environment for execution of the instructions. Instructions are fetched from the address *PC* and executed using the *G*, *Y*, and *X* registers.

2. **X register saving task:** The MVM provides a single set of *X* registers. The illusion that every thread has its own private set of registers is preserved by saving the *X* registers when a thread is preempted and restoring them when the thread is restarted.

3. **Exception handler task:** A handler task is used for exception handling and they are never executed directly.

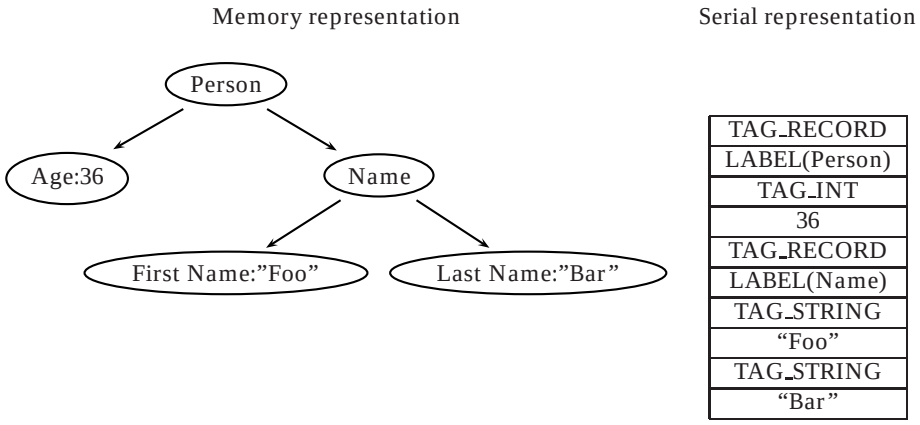

Figure 5.2: Memory and serial representation of data entities

5.4 Marshaling

Here we give a short description of marshaling in the Mozart system. Detailed description of the marshaling model together with an evaluation of the implementation in the Mozart system can be found in [27].

The marshaler consists of two parts: a set of *marshaling methods* with at least one method per data structure and *a traverser*. The traverser passes over a data structure node and applies appropriate marshaling methods for the data structures in the subnodes.

The interface between the marshaler and the rest of the system is described by an example. In the example a message `Msg` is followed from creation at the source site to reception at the destination site:

- Source Site
  - `Message` is created by the application layer at the source site.
  - An operation is performed on `Message`, for example {`Port.send P Message`} where `P` is a port located at the destination site.
  - *DSS* is triggered by the operation and a request for marshaling of `Message` is forwarded to the marshaler.
  - The marshaler marshals `Message` and inserts its serial representation into the serialized buffer. `Message` is a language entity that can be described as a directed graph of nodes. A node contains values and references to other nodes. The serial representation of a language entity consists of a sequence of tokens that represents nodes (see Figure 5.2). The marshaler traverses over `Message`, constructs the serial represen-

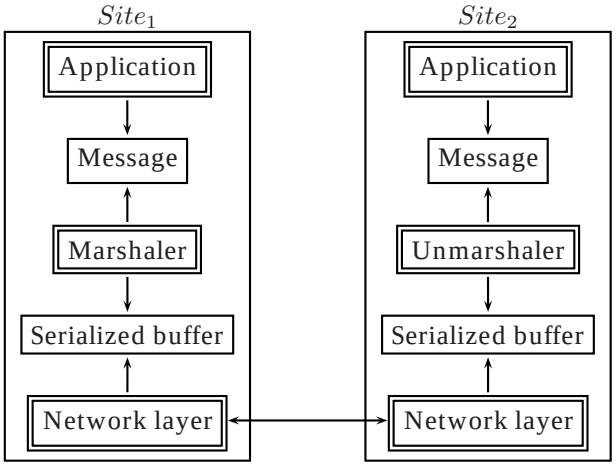

Figure 5.3: Layers in the Mozart distributed system

tation of each subnode and inserts it into the serialized buffer. Figure 5.3 shows that `Message` is shared between the *application layer* and the marshaler and the *serialized buffer* is shared between the marshaler or the unmarshaler and the *network layer*.

  - The network layer reads the data from the serialized buffer and copies it to the network.

- Destination Site
  - The network layer reads data from the network and writes it to the serialized buffer.
  - *DSS* passes the serialized buffer reference to the unmarshaler and triggers unmarshaling.
  - The unmarshaler reads the data from the serialized buffer and creates the memory representation of nodes which are assembled into larger nodes until complete `Msg` is created.
  - The `Msg` reference is passed to the application layer.

The implementation of the marshaler in the Mozart system is very efficient. The efficiency is based on the following characteristics:

- Marshaling of *a message* is performed concurrently with network delivering of *the same message*. This is important for latency reduction in the case of large messages.
- Marshaling is preemptive. This is necessary to achieve concurrent work of the *marshaler* and the *network layer*.
- Size of the serialized buffer is constant.
- A time slice for each activation of the *marshaler* is limited.

5.5 Marshaling Tags

Marshaler and unmarshaler use *tags* for identification of serialized language entities. Each language entity type has its own marshaling tag. The marshaler begins serialization of a language entity by insertion of the marshaling tag for the corresponding entity type into the serialized buffer. Then it proceeds with serialization of the entity.

The marshaling tags are also used to mark end of serialized entity (`TAG_EOF`) and to mark possible suspensions (`TAG_SUSP`) and continuations (`TAG_CONT`)

which can occur in the case of suspension due to large data structures that not fit in the serialized buffer.

5.6 Resource Marshaling

In the Mozart system resources are entities whose use is restricted to one site and their distribution is handled in the following way:

- All sites have a table where they keep references to the exported resources.
- When a marshaler traverses over a language entity node and discovers a subnode that corresponds to a local resource a new entry in the table is created and a reference to the resource is inserted into the table of exported resources.
- The index in the resource table is marshaled together with the owner site identity instead of the resource.

In this way the information about the exported resource is preserved. Moreover, the exported resources are uniquely represented at remote sites and equality test behaves correctly. When the resource entity is received back from the network, the real resource reference replaces the resource representation used at remote site.

In the current release of the Mozart system threads are considered as resources. Our implementation changes that and enables for distribution of threads.

5.7 Global Names and Distributed Entities

Distributed stateful language entities use global unique names to implement their identity. Thus, all stateful language entities have an attribute that is specifically used for global identification. Globalization of a language entity occurs when its reference is passed to a remote site for the first time. The entity becomes *distributed* and the globalization attribute is bound to a brand new global name *GUID*. Stateful entities used only at the local site do not use global names for identification due to efficiency reasons. The global name is implemented as C++ class which uses combination of a site identity and a large integer number to create globally unique name. It belongs to the system internals and is not exposed to the application layer. In the process of globalization of the stateful data entity the corresponding abstract entity is also created on the source site and the destination site.

5.8 Implementation of Strong Mobility in Mozart

Implementation of strong mobility in the Mozart system presented in bottom-up order consists of the following steps:

- Implementation of the marshaling methods for threads and globalization of threads.
- Implementation of the replication primitive.
- Implementation of the migration manager.
- Implementation of the abstractions.

5.8.1 Marshaling Methods for Threads

In this section we describe the implementation of the marshaling methods for threads. We also describe issues related to the lazy distribution protocol used for distribution of objects in Mozart and we present a solution for the problem.

Oz Threads

In the section 5.6 we said that threads in the Mozart system have been considered as a local resources. An implementation detail which is direct consequence of that is that threads do not have global names, because they never needed them.

Therefore, the first step in making threads distributed is to extend them with global names to ensure their global uniqueness. The standard procedure for globalization of stateful data entities used in Mozart is described in the section 5.7. We have used the same technique for threads. Thus threads are extended with a new attribute, `GName guid`, for global identification. The attribute `guid` is initialized to `NULL` pointer on thread creation. The globally unique name is created in *by need* fashion, that is when a thread reference has been sent to a remote site for the first time.

The second step is to provide marshaling and unmarshaling methods for threads. The Figure 5.4 presents a thread as a directed graph of nodes. Thus a thread is a root node pointing at four descending nodes: an id, a guid, a priority flag, and a thread stack.

Thread's id, `id`, and thread's priority flags, `flags`, are represented as integers which makes their serialization straight forward. Globally unique name, `guid`, is represented as C++ class for which the marshaling method in the system already exists. The last remaining node is a thread stack, `threadStack`. Marshaling of the thread stack is more complex, due to non constant number of stack entries

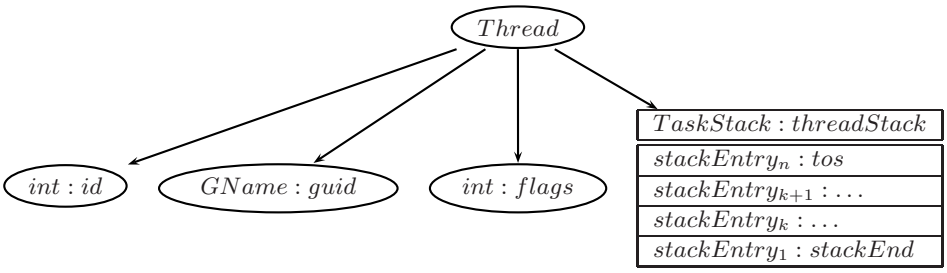

Figure 5.4: Thread Memory Representation

which can lead to suspension of the marshaler. Due to possible suspension the marshaling of threads is divided in two parts:

1. Marshaling of: `threadTag`, `id`, `flags`, `guid`, and `nrOfStackEntries` which is an integer describing the number of stack entries on the thread stack.

2. Marshaling of the thread stack, `threadStack`.

The fact that we have divided marshaling of threads into two parts in practice means that a thread stack is marshaled as a separate node. Thus, it must be properly identified in the serialized buffer and a new identification *tag* for the thread stack is introduced `TAG_THREAD_STACK`.

Marshaling Thread Stack

In this section we first present a generic method for marshaling stack entries. Then we proceed with description of issues which are specific for some entry types and present solutions for them.

Generic Marshaling Method for Stack All values are first-class including procedures, objects and classes. A procedure accessed by a thread is transferred only once for each thread. Migration of another thread that has the same procedure reference leads to second transfer of the same procedure. However all distributed stateful entities have globally unique identifiers which means that procedures (and other stateful entities) are represented at most once at each site.

A thread stack is stack of statements. A statement consists of a triplet `(PC, Y, G)` where:

- `PC` is program counter for code to be executed.
- `Y` is set of local variables saved in registers.
- `G` is pointer to the current procedure closure.

`PC` has absolute value which must be translated to relative value with respect to the current procedure.

One important design decision in Mozart, for making marshaling simple, is that only values are marshaled. To comply with that, we collect necessary information to rebuild a stack entry at the receiver site, create an Oz value from it and call the marshaling method for the value. For example:

1. Pop a stack entry from the stack.
2. Create a tuple `ET` with three features as a placeholder for a stack entry.
   (a) Insert the relative offset of `PC` into the first field of `ET`.
   (b) Collect local variables from `Y` into another tuple `LVT`. Insert `LVT` into the second field of `ET`.
   (c) Insert the procedure pointed to by `G` register into the third field of `ET`.
3. Call the marshaling method for tuples, which is already implemented in the system.
4. Repeat the procedure for each stack entry.

Implementation Issues There are several important properties that must be considered when implementing the marshaling method for the thread stack in the Mozart system:

- The amount of space needed for serialization of thread stack entries depends on the type of entry and is not constant.
- Number of stack entries on the thread stack in the moment of serialization is not constant and a variation in size can be significant.
- Size of the serialized buffer is constant.
- The marshaler is able to suspend on the full serialized buffer and resume later on when the buffer has been emptied (that is, the serialized data has been sent).

- Default distribution protocol for objects in Mozart copies state of objects in a lazy fashion. Thus when a reference to an Oz-object is sent to a remote site then: the object is globalized, an abstract entity for the object is created and the globally unique name representing the object is sent to the remote site. That means that the corresponding class definition and the object state are sent later on when the object is called for the first time at the remote site.

  Another important implementation detail concerning Oz objects is that object pointers are directly inserted into the stack entries. It is done this way due to efficiency reasons.

The marshaling of the thread stack begins with insertion of the tag representing the stack (`TAG_THREAD_STACK`) into the serialized buffer. The method continues by serialization of the stack entries iteratively and insertion of their serialized representation into the *serialized buffer*. A control for sufficient space in the serialized buffer is performed in each iteration. The lowest space boundary for the buffer is determined by the size of the largest stack entry. Thus, only complete stack entries are marshaled. In the case that the serialized buffer is full, a special tag `TAG_SUSP` is inserted into the buffer and the marshaling process is suspended. After the network layer finish the sending (that is, copying the data from the buffer to the network) the marshaling process can be resumed. The resumption begins by insertion of the tag `TAG_CONT` into the serialized buffer. The flag `TAG_SUSP` is used to inform the unmarshaler at the destination site about continuation that follows in the next serialized message.

Marshaling the stack entry containing an object pointer The marshaling of the stack entry containing an object pointer results in marshaling of the object stub and not the complete object. This is due to implementation decision in Mozart to distribute objects in the lazy fashion. That means that the object state is migrated only on the receiver demand. The default method marshals only a global name and a dataflow variable (a *by-need* variable) which is used by *DSS* to implement a *lazy* protocol.

Unfortunately, at destination site the complete object is needed to rebuild the correct stack entry. Additionally there is no elegant solution which can be used to inform the marshaler to use the *right* marshaling method without breaking the boundary between *DSS* and *Marshaler*.

Our solution is based on the fact that the destination site triggers migration of the object state. Thus, the unmarshaled thread stack is traversed and object references together with `by-need` variables are discovered and collected into a list. The collected objects are requested from application layer by calling a procedure `{Wait Obj}`. `Wait` procedure triggers the distribution subsystem and activates

the protocol which migrates the object state from the source site to the destination site.

A drawback of this solution is that an overhead is introduced by additional traversing which is needed to collect and request all stack referred objects.

5.8.2 The Replication Primitive: `ThreadSend`

Distribution in Mozart is transparent and the marshaler is hidden from the application layer. Thus, the system does not provide abstractions for marshaling and unmarshaling. Instead, the underlying distribution subsystem, *DSS*, is responsible for activation and instruction of the marshaler.

The marshaler implements at least one method per language entity. When several methods are provided for an entity, *DSS* instructs the marshaler which serialization method to use.

DSS in its turn chooses between the marshaling methods by looking at the message *MB* that waits to be delivered. *MB* contains information needed to make decision about the marshaling method. The message *MB* belongs to the system internals (see Figure 5.3) and is created by the application layer.

In the case of threads we distinguish between two cases and provide two serialization methods:

- **Sending a thread reference.** The thread reference is sent in three cases:

  1. A thread *T* is referenced by an abstract data type *ADT*. Migration of *ADT* from one site to another do not lead to migration of complete thread state, instead a thread reference, (that is thread's global name) is migrated together with *ADT*.
  2. A thread *T* is sent by using an Oz port, that is the abstraction {`Port.send P T`} [11, chapter 9.6] is called.
  3. A distributed dataflow variable is bound to the thread reference.

  Action: The application layer creates the message *MB*. *MB* contains instructions that only thread reference must be sent. *DSS* checks the message *MB* and instructs the marshaler to serialize only the global name of the thread. Thus, the thread stack is not marshaled.

- **Sending a complete thread.** One of the thread migration abstractions is used.

  Action: The application layer creates the message *MB* which contains instructions that complete thread must be sent. *DSS* checks the message *MB* and instructs the marshaler to serialize the complete thread state including the thread stack.

Replication primitive {`SendThread Site threadMigration(MobThr ?R)`} is a procedure used by the migration manager to perform the thread migration. That is, the replication primitive is used to create the message *MB* which contains instructions for *DSS*.

It is a special case of the procedure `Send` [11, chapter 9.6] provided by the system. {`Send Prt Msg`} sends `Msg` to the port `Prt`. The first argument `Site` represents the destination site and is implemented as Oz port [29] which accepts only thread migration messages. (Oz ports are influenced by ports in AKL [20], [19]). The second argument is a tuple where the first field contains a mobile thread `MobThr` and the second field holds a dataflow variable `R` used for reply.

5.8.3 Distribution Consistency and Resources

Migration of threads raises several issues. For example, updates of thread references after thread migration, handling of resources after migration.

One approach to solve distribution consistency is to extend the distribution subsystem with a consistency protocol for threads like it is done for objects in Mozart. The future release of the Mozart system will provide new modular distribution subsystem that provides a number of consistency protocols which are completely separated from data entities. Thus it is possible to choose between variety of protocols for stateful data structures [22]. Another way is to use a data structure with implemented distribution consistency to achieve distribution consistency of thread.

The resource related problems are not easy to solve. A solution that provides a transparent rebinding of resources demands considerably big changes in the current system. On the other hand transparent rebinding is not always what one would like to have. For example, an open file or GUI referred by a thread can be copied to a destination site together with the thread but it can also be left behind at the source site and referred from the destination site as well. We believe that a solution which supports both options is more desirable.

Our solution for both distribution consistency and resource issues is based on a new data structure, *MobileThread*.

5.9 Mobile Thread

A data structure `MobileThread` is implemented as an Oz object with three attributes:

- `thread`: represented as Oz thread.

- `site`: represented as Oz port. It is an incoming communication channel used by the local migration manager.
- `res`: represented as Oz record. It holds information about resources used by `thread`.

An instance is created by calling `MT = {New MobileThread init(Thr Site ResLs)}` where `Thr` is a thread, `Site` is a reference to the migration manager at the current site, and `ResLs` is an Oz record.

Resources In the current implementation, we provide only for rebinding of ubiquitous resources. These resources are represented on all Mozart sites, for example standard libraries and the system modules like graphical user interface or standard output.

Before an instance of `MobileThread` is created several steps must be performed by the application programmer:

- Identify all resources which are used by the thread.
- Write a resource handler function for each identified resource. The functions are used by the migration manager at the receiver site to dynamically replace the resource references from the source site. In the case that a function is not provided for some of the resources the migration manager uses default replacement procedure.
- Create an Oz record with the features that lexically match names of the used resources and that has values bound to corresponding functions.
- Get a reference to the source site.
- Get a reference to the chosen thread.

The implementation is based on the slightly changed marshaling method for resources. The description of the current marshaling method for resources is presented in 5.6. New marshaling method extends the old one by adding a resource tag (represented as Oz atom) to the serial representation of resources. The resource tag lexically match the name of the resource.

Unmarshaling in the Mozart system consists of two actors: *unmarshaler* and *builder*. The former reads data from serialized buffer, token by token, constructs the memory representation of nodes and passes those nodes to the latter one which gather the nodes into toplevel structural nodes. When unmarshaler discovers a resource specific tag in the serialized buffer it passes the resource tag to

the builder and the builder inserts the tag on place where the resource reference suppose to be.

The migration manager at the receiver site upon reception of a thread performs the following procedure to dynamically rebind resources:

- Traverses the thread stack and finds all resource tags.
- Calls the corresponding function for each resource tag. The function definition is either provided by the programmer and is placed in `res` attribute or the default function is used. The function call returns the valid reference to the local resource.
- The returned resource reference is inserted at the place where the resource tag use to be.

`MobileThread` provides basic methods needed for retrieval and update of `site` and `resList` attributes. It also provides `go(Site)` method which is basically a wrapper around `Push` abstraction.

The abstractions `Push` and `Pull` now operate on a data structure of type `MobileThread` instead of type `thread`.

Distribution Consistency Distribution consistency of `MobileThreads` is now much easier to provide because `MobileThread` is an Oz object and the system provides consistency protocol for objects. The only concern that must be taking care of is that `site` attribute is properly updated upon migration by the migration manager at the destination site. Figure 5.5 shows the MobileThread class interface and a code example for `go(Site)` method is presented below:

```
meth go(Site)
   Sync Thr
in
   {self getThread(?Thr)}
   thread
     {Thread.suspend Thr}
     Sync = done
     {Push self RemoteSite}
   end
   {Wait Sync}
end
```

Figure 5.5: MobileThread Class

5.9.1 The Migration Abstractions and Migration Manager

The implementation of the migration abstractions strictly follows the model presented in the section 4.3 and the migration manager is slightly modified, thus taking into account migration of lazy objects referred by the thread stack.

```
// Migration Manager
proc {MigrationManager Id Ticket LocalRes}
   P S T
in
   P = {NewPort S}
   T = {Connection.take {Pickle.load Ticket}}
   T.Id = P
   thread
      {WaitForThreads S LocalRes}
   end
end

proc {WaitForThreads S LocalRes}
   for Msg in {GetMessage S} do
      case Msg
      of threadMigration(MobThr Reply) then
         Thr
         LocalSite
```

```
         in
            LocalSite = {GetSite}
            {MobThr getThread(?Thr)}
            /* Update the current site of MobThr */
            {MobThr putSite(LocalSite)}
            /* Try to rebind resources */
            if {RebindResources MobThr} then
               /* In the case of success resume the thread */
               Reply = success
               /* Enforce migration of objects referred by */
               /* the thread stack */
               {PullLazyObjects Thr}
               {Thread.resume Thr}
            else
               /* In the case of failure terminate the thread */
               Reply = failure
               {Thread.terminate Thr}
            end

         [] migrate(MobThr Site Result) then
            /* If migration is successful Result is bound */
            /* to 'success' otherwise to 'failed' */
            Result = {ThreadReplicate MobThr  Site}
         end
      end
   end
```

5.10 Summary

In this chapter we have first presented details of the Mozart implementation which are important to understand decisions we made in our implementation. After that we presented implementation of strong mobility by extending the Mozart virtual machine. The extensions are more or less straightforward and small in size (around 800 lines of C++ and 200 lines of Oz). The system keeps backward compatibility and no applications need to be rewritten.

Minor issues concerning the lazy behavior of the Oz objects together with thread migration have been identified and solved. The solution thus introduce additional overhead but the first tests show that the size of the overhead in most cases can be neglected.

Chapter 6

Evaluation

In this chapter, we present the evaluation of the model implemented in the Mozart programming system.

The evaluation scenario used here is inspired by the paradigmatic mobile agent evaluation (see [15]). First, we compare the performance of the mobile agent model built on top of thread-based mobility with the client-server model. Then we compare the performance of two implementations of the mobile agent model: one is based on the Mozart implementation and the other is based on the NOMADS [30] implementation.

The first test is performed on a single processor machine (Pentium 750 MHz, 384 MB RAM) running the Linux Gentoo 2.4.19-gentoo-r5 operating system. It is repeated 10 times and the standard deviation was smaller than 0.2 percent.

The second test is performed on a single processor machine (AMD Athlon XP 1700+, 512 MB RAM) running the Windows 2000 operating system (the NOMADS system is only available for the Windows platform). It is repeated 10 times and the standard deviation was smaller than 0.3 percent.

The task of the application is to collect a list of hotels with phone numbers in one town at a customer site S_c. Two database servers are consulted: The hotel database server H at the site S_H to obtain a list of available hotels in the specified town; the phone database server P at the site S_P to obtain a phone number of each hotel from the list, one at time.

6.1 Client-Server vs Mobile Agent

Client-Server. The client at the customer site sends a hotel list request req_h to the H server. After the list has been received, the client sends one request req_p per

Table 6.1: Mobile Agent vs. Client-Server in the information collection

(a) *On a WAN*

	Number of hotels				
	1	10	20	30	40
Client-Server (ms)	80	370	710	1040	1380
Mobile Agent (ms)	410	410	410	410	440

(b) *On a single computer*

	Number of hotels						
	10	20	30	40	50	100	200
Client-Server (ms)	3	6	10	13	17	35	82
Mobile Agent (ms)	6	7	7	8	9	12	24

hotel to the server P to obtain telephone numbers. The total time for the task is equal to the time for req_h plus the time for $n * req_p$, where n is the number of hotels.

Mobile Agent. The mobile agent moves from the customer site S_c to the site S_H and requests a list of hotels locally. After the list has been received, the agent moves to the site S_P and queries the server P for telephone numbers. When all phone numbers are collected the agent moves back to the site S_c and returns the result.

Results. We assume that database operations have constant cost. We have performed two measurement:

- The customer is sited on a computer in Germany[1] and the servers are sited on computers in Sweden[2].
- The customer and the servers are sited on a single machine in three processes.

We vary the number of returned hotels and measure the total time in both solutions. Tables 6.1(a) and 6.1(b) summarize the results. We see that in the WAN

[1] Universität des Saarlandes

[2] Swedish Institute of Computer Science, SICS

Table 6.2: Mobile Mozart vs. NOMADS in the Information Collection

	Number of hotels			
	10	20	30	40
Mozart (ms)	210	220	220	220
NOMADS (ms)	3609	4625	5656	6766

case for small numbers of hotels the client-server is more efficient than the mobile agent. In all other cases the mobile agent is much more efficient and it scales.

The second test shows that the client-server performs better only if the number of hotels is less than 20. The test also shows that in the case of client-server the time increases linearly with the number of hotels, which is not the case with the mobile agent. In the case of the client-server, communication is performed between operating system processes. In the case of the mobile agent communication is performed between threads inside one operating system process.

The evaluation shows that the mobile agent model is not only well adjusted to the distributed applications that run over WAN, but for applications that run over cluster, grid, and LAN as well.

6.2 Mozart vs NOMADS

In the second part of the evaluation two implementations of the mobile agent model are compared, our implementation in Mozart and the NOMADS system implemented in Java running on the virtual machine Aroma. The test is performed on a single machine. In the case of Mozart the same agent implementation is used as in the test described above.

The agent written in Java uses RMI for communication with H and sockets for communication with P. Sockets are used instead of RMI because the RMI implementation in Aroma does not work with mobile agents.

Table 6.2 shows that our implementation has much better performance and that it scales much better than the NOMADS system.

6.3 Strong Mobility vs Weak Mobility

In this section we compare two implementations of the mobile agents in the Mozart programming system: Weak mobility based and strong mobility based.

For the evaluation purposes we have written a mobile agent package that mimics behavior of mobile agent used in the previous section. That is, only dif-

ference is a new {Go Site Agent} abstraction and new migration managers that perform migration of agents.

In this case the agent cannot continue its execution from the next instruction. The abstraction {Run Agent Site} is called to restart the agent after migration. The second argument in Run abstraction is used by the agent to make decision about the next step in execution. A code examples representing the migration managers, Go abstraction, and the mobile agent follows:

A code example of a migration manager is presented below:

```
proc {WaitForRequests Requests}
   for Query in Requests do
      case Req
      of go(Site Agent Ack) then
         Thr     // the thread ID
      in
         Thr = {GetThreadId Agent} // get the thread ID
         {Thread.terminate Thr}
         {Send Site incoming(Agent)}

      [] incoming(Agent State) then
         thread                  // Create a new thread
           {Run Agent Site}      // and start the agent
         end                     // with updated site info

      end
   end
end
```

A code example of Go abstraction is presented below:

```
proc {Go Site Agent}
   Ack
in
   {Agent.pack}
   {Send Agent.manager go(Site Agent Ack)}
              // Suspend on unbound variable 'Ack'
   {Wait Ack} // this line will never execute
end
```

We have tried to minimize difference between implementation of the mobile agents using strong mobility and weak mobility. The main difference is in the way the mobile agent continues after migration. A code example representing a restart of the mobile agent using the weak mobility is presented below:

```
proc {Run Agent Phase}
   case Phase
   of 1 then
      {Go Agent.hotelSite Agent}
      // There is no return from Go here

   [] 2 then
      {Agent.collectHotelNames}
      {Go Agent.phoneNrSite Agent}


   [] 3 then
      {Agent.collectPhoneNrs}
      {Go Agent.homeSite Agent}

   [] 4 then
      {Agent.returnResult}
   end
end
```

Run procedure is called when the agent thread is spawned. The second argument (Phase) is used to instruct the agent what to do next. This is a consequence of the fact that execution cannot continue at the next instruction when weak mobility is used.

Run procedure for the mobile agent using strong mobility has a structure that is much easier to understand. A code example is presented bellow:

```
proc {Run Agent}
   {Go Agent.hotelSite}
   {Agent.collectHotelNames}
   {Go Agent.phoneNrSite}
   {Agent.collectPhoneNrs}
   {Go Agent.homeSite}
   {Agent.returnResult}
end
```

Here, the execution continues at next instruction after migration.

6.4 Summary

In the first part of evaluation we have compared two programming models: the mobile agent model and the client-server model. The comparison clearly shows that applications producing moderate amount of network traffic benefit directly from the mobile agent model. Our tests show that this is true even in the case when the communicating applications are located at the same machine, because inter process communication is much more expensive than communication between threads running inside a virtual machine.

In the second part we have compared performance of mobile agents based on the weak and strong mobility. The Mozart system supports for weak mobility and for the test purposes we have written abstractions that strictly follow the model we used to implement strong mobility. Tests showed that performance of the systems is approximately the same, but the presented code examples show that the weak mobility creates “unnatural” non-modular programming style which is difficult to reason about and very hard to debug.

Chapter 7

Related Work

The current state-of-art for system supporting the programming of mobile applications can be best classified with respect to the following main criteria: what is the unit of mobility for the approach; whether strong or weak mobility is supported; whether migration control is implicit or explicit; what is connection to support for distributed programming; how is mobility implemented (or what is the underlying implementation architecture for mobility).

Which unit of mobility is chosen in a certain approach typically coincides with the preferred abstraction of structuring programs in a certain programming language. Current approaches choose either objects, agents, operating system processes, or active objects combining objects and threads. Even though we are mostly interested in strong mobility here, we also survey approaches which are interesting from the distributed programming aspect but only support weak mobility.

Prominent approaches are the following, where we highlight what decisions are made for the above mentioned criteria.

7.1 Telescript

Telescript was developed in middle nineties by General Magic. Telescript is an object oriented language created for the development of secure distributed applications. One of the main driving factors in the language design was a focus on mobile agents and strong mobility. Main focus of the system was

Two kinds of execution units: *places* stationary, and *agents* The system is not longer available.

7.2 Emerald

The Emerald programming language and system has been developed in the late 80's and 90's at the Department of Computer Science, University of Copenhagen. Emerald has been the first system that offers *fine-grained* mobility [21]. The Emerald language is an imperative object-oriented language where all data is represented as objects. In Emerald, mobility is integrated in the language. Migration is provided by the *move* statement which moves an object to another site. Any thread executing in a moving object is moved together with the object. In the Emerald migration model, threads follow objects around as the objects are moved.

7.3 Obliq

Obliq is an untyped object-oriented interpreted language with distributed lexical scope developed at DEC by Cardelli [6]. Obliq supports weak mobility. Thus, remote execution is provided and when remote execution is requested the code representing a procedure is sent and executed at the remote site. Remote execution is function-call based and suspends until execution returns.

7.4 Erlang

Erlang is declarative language developed for programming concurrent and distributed systems developed by Joe Armstrong [2] et.al. at the Ericsson and Ellemtel Computer Science Laboratories. Erlang has a process-based model of concurrency.

Concurrency in Erlang is explicit and the user can precisely control which computations are performed sequentially and which are performed in parallel. Message passing between processes is asynchronous, that is, the sending process continues as soon as a message has been sent. Code loading primitives allow code in a running system to be changed without stopping the system. Thus, the system supports for weak mobility. Erlang processes do not share anything which simplifies the model considerably. That makes the Erlang system a perfect candidate for strong mobility, but to our best knowledge there is no publications explaining work done on implementation

7.5 Sumatra

Sumatra developed at the University of Maryland is a Java extension. It implements strong migration by extending the Java class library and by modifying the Java runtime. The unit of migration is a new abstraction primitive *object-group*. This primitive allows the application programmer to customize the granularity of migration. Object-groups are migrated between execution-engines (that is, an Java interpreter executing on a host). An execution-engine may host several threads, but multiple threads are scheduled in a *run-to-completion* manner.

7.6 JoCaml, Join Calculus, and Ambient Calculus

JoCaml [9] is an implementation of the Join-Calculus, which is a reformulation of π-calculus with explicit notion of places of interaction.

The programming model is based on *locations* and *channels*. *Locations* gather both agents and sites in single abstractions. Sites are toplevel locations and agents are nested locations. A location can contain threads, channels, and sub-locations. The unit of mobility is a location. Thus, migration of a location implies migration of all sub-locations as well. Threads are not part of the language semantics. *Channels* are communication links and are maintained during location migrations. The Join calculus and the Ambient calculus [7] are very closely related as pointed out in [12] where the Ambient calculus is implemented in JoCaml. There is a very close relation between *locations* and *ambients*. An ambient is a *bounded* place where computation happens. In the Ambient calculus locations are not uniformly accessible and they are not identified by globally unique names.

The major difference between the Join Calculus and the Ambient Calculus is that in the Join calculus movement may happen directly from any active location to any other known location. In the Ambient calculus locality and control have strong connection. Each ambient is a box, and interactions can occur only between processes that are in adjoining ambients. Thus, interaction cannot happen without proper consideration of boundaries and their topology. The unit of mobility is an ambient and agents are confined to ambients.

7.7 D'Agents

Developed at the University of Dartmouth, D'Agents [14] is a multi-language system consisting of Agent Tcl [13], Agent Java, and Agent Scheme. The first two have support for strong mobility (Agent Java is based on the Sumatra system). The unit of migration in D'Agents is a process. The system provides a migration

abstraction *agent_jump*. A D'Agent server must be running at each cooperating site. When an agent calls *agent_jump*, the complete state of the agent is captured and marshaled to the target machine. The D'Agent server on the target machine on reception creates a new process running the Tcl interpreter.

7.8 ARA

Developed at the University of Kaiserslautern, Ara [26] is a multi-language system that provides strong mobility. A migration unit in Ara is an agent. Agents are managed by a language independent system core and interpreters for supported languages (C, C++, Tcl, Java). An Ara agent cannot share anything and resource bindings are removed before migration.

7.9 JavaThreads

[5] is a JVM extension. The portable thread state is provided by a type inference technique and thread serialization by combination of type inference and dynamic de-optimization techniques. Migration in JavaThreads depends on the deprecated $stop()$ method in $Java.lang.Thread$. The unit of mobility is a thread. The migration is provided by the $go()$ statement.

7.10 NOMADS

[30] is a Java-based agent system. It uses a custom virtual machine, known as Aroma, with the ability to capture thread execution states. Java threads are mapped to native operating system threads. The unit of mobility is a thread. Migration is provided by the $go()$ statement.

7.11 Summary

In this section we summarize related work. Table 7.1 sorts the systems by: kind of mobility they provide, programming language, the provided mobility abstractions and unit of mobility.

Table 7.1: Code Mobility Systems

System	Mobility	Language	Abstractions	Unit of mobility
Emerald	strong	Emerald	move	object
Telescript	strong	Telescript	go	agent
JoCaml	strong	JoCaml	go	location
Sumatra	strong	Java	go	thread
JavaThreads	strong	Java	go	thread
NOMADS	strong	Java	go	thread
ARA	strong	C, C++, Tcl, Java	go	agent
D'Agents	strong strong weak	Java, Tcl, Scheme	jump	thread, process, code
Java Aglets	weak	Java	dispatch, retract	code
Obliq	weak	Java	none	code
Erlang	weak	Erlang	none	code

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis presents a model, an implementation and an evaluation of strong mobility in Mozart / Oz.

The model is based on migration managers and thread replication. In the model we choose as unit of mobility a new data structure, MobileThread, which encapsulates a thread, resources used by the thread and the current site. The useful migration abstractions `Go`, `Pull`, and `Push` are identified. These abstractions create a powerful programming package for explicit thread migration that covers all migration aspects. Thus, migration can be initiated from *source site*, *destination site*, and from a third site. In addition, the migration can be initialized by the thread itself or from some other thread.

The Mozart virtual machine is extended with thread serialization and replication primitive which is foundation for migration abstractions. The extended system is backward compatible.

An evaluation of the system shows that strong mobility and systems with lightweight concurrency and dataflow synchronization are successful combination. It shows that strong mobility is cheap in contrary to the conclusions drawn based on the implementations of strong mobility in Java [30, 14].

8.2 Future Work

The current model needs to be improved in several aspects, specially in the area of resource description.

8.2.1 Resources

Resource description and identification is crucial for application using strong mobility. In continuation we will work on enhanced resource control. Thus the development of the model which is briefly presented in 5 is in the progress. The current model is limited only on dynamic rebinding of ubiquitous resources, but it is possible to extend the model so that it handles even other types of resources. The future implementations will continue in the Mozart system because it provides first class module managers. This will considerably simplify implementation of the resource model.

Another important issue related to strong mobility, particularly in the case of network partition avoidance and failure avoidance is communication channels. For example, the current implementation of the distribution protocol for communication channels in Mozart (that is Oz ports) is based on the stationary manager. Manager and proxies are part of *DSS* and are responsible for keeping consistency. The stationary manager has as a consequence a third part dependency. An example with a mobile agent MA and three sites S_1, S_2, and S_3 that illustrate the dependency problem follows:

MA is created at the site S_1 and later on a reference to the input channel P_{MA} (that is an Oz port) is passed to an agent A at the site S_3. This will create a distributed access structure for P_{MA}. That is a manager and a proxy for the port P_{MA} is created at S_1 and S_3 respectively. After while MA makes decision to migrate to the site S_2 due to planned upgrade at the site S_1. The manager for P_{MA} stays at the site S_1 and a proxy structure for the port P_{MA} is created at S_2. The site S_1 is disconnected and the access structure for the port P_{MA} is lost even if MA has survived.

The future work includes implementation of distribution protocols for communication channels with mobile managers that do not have dependencies on the third site.

Bibliography

[1] Anurag Acharya, Mudumbai Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[2] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, Englewood Cliffs, NY, USA, 1993.

[3] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. The MIT Press, Cambridge, MA, USA, 1991.

[4] J.K. Boggs. IBM remote job entry facility: Generalize subsystem remote job entry facility. *IBM Technical Disclosure Bulletin 752*, August 1973.

[5] Sara Bouchenak and Daniel Hagimont. Zero overhead java thread migration. Technical Report 0261, INRIA, 2002.

[6] Luca Cardelli. Obliq: A language with distributed scope. Technical report, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA, November 1994.

[7] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[8] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, Ma., May 1997.

[9] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[10] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In *Mobile Object Systems: Towards the Programmable Internet*, pages 93–110. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[11] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment*. The Mozart Consortium, `www.mozart-oz.org`, 1999.

[12] Cédric Fournet and Alan Schmitt. An implementation of ambients in JoCaml. In *Proceedings of the 5th ECOOP Workshop on Mobile Object Systems (MOS'99)*, Lisbon, Portugal, 1999.

[13] Robert Gray, David Kotz, George Cybenko, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zyda, editors, *Mobile Agents: Explanations and Examples*, chapter 4, pages 58–95. Manning Publishing, 1997. Imprints by Manning Publishing and Prentice Hall.

[14] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. Mobile agents: Motivations and state-of-the-art systems. Technical Report TR2000-365, Dartmouth College, Hanover, NH, April 2000.

[15] Daniel Hagimont and Leila Ismail. A performance evaluation of the mobile agent paradigm. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–313. ACM Press, 1999.

[16] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[17] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[18] Dragan Havelka, Christian Schulte, Per Brand, and Seif Haridi. Thread-based mobility in Oz. In Peter Van Roy, editor, *Multiparadigm Programming in Mozart/Oz: Second International Conference*, volume 3389 of *Lecture Notes in Computer Science*, Charleroi, Belgium, October 2004.

[19] Sverker Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994. SICS Dissertation Series 14.

[20] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, USA, 1993.

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

[22] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The design and evaluation of a middleware library for distribution of language entities. In Vijay A. Saraswat, editor, *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference, Mumbai, India, December 10-14, 2003, Proceedings*, volume 2896 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2003.

[23] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The DSS, a middleware library for efficient and transparent distribution of language entities. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[24] Michael Mehl. *The Oz Virtual Machine: Records, Transients, and Deep Guards*. Doctoral dissertation, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany, 1999.

[25] Mozart Consortium. The Mozart programming system, 1999. Available from `www.mozart-oz.org`.

[26] Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.

[27] Konstantin Popov, Vladimir Vlassov, Per Brand, and Seif Haridi. An efficient marshaling framework for distributed systems. In Victor E. Malyshkin, editor, *Parallel Computing Technologies, 7^{th} International Conference (PaCT 2003)*, volume 2763 of *LNCS*, pages 324–331, Nizhni Novgorod, Russia, September 15–19 2003. springer. A revised version to appear in Future Generation Computer Systems, 2004.

[28] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Doctoral dissertation, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany, December 1998. In German.

[29] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[30] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. Jeffers. Strong mobility and fine-grained resource control in nomads. In *The Second International Symposium on Agent Systems and Applications / Fourth International Symposium on Mobile Agents*, Zürich, Switzerland, September 2000. Springer-Verlag.

[31] Peter Van Roy, Seif Haridi, and Per Brand. *Distributed Programming in Mozart - A Tutorial Introduction*. The Mozart Consortium, `www.mozart-oz.org`, 1999.

[32] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.