



**PEPITO**  
**IST-2001-33234**  
**PEer-to-Peer Implementation and TheOry**

**Deliverable no: D3.1**

## **Report on Mobile computation design**

REPORT VERSION: first

REPORT PREPARATION DATE: 2003.12.31

CLASSIFICATION: Public

DELIVERABLE NO: D3.1      DUE DATE: Month 24      DELIVERY DATE: Month 24

PROJECT START DATE: 2002.01.01      PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: KTH

PARTICIPATING PARTNERS: KTH

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK



**Project funded by the European Community under the 'Information Society Technologies' Programme (1998–2002)**

Project Number: IST-2001-33234  
Project Acronym: PEPITO  
Title: PEer-to-Peer Implementation and TheOry  
Deliverable No: D3.1  
Report on Mobile computation design  
Due date: project month 24  
Delivery date: 2003.12.31

Responsible Partner: KTH  
Participating Partners: KTH

23rd January 2004

By Dragan Havelka, Seif Haridi, and Christian Schulte.

## Contents

|           |   |           |
|-----------|---|-----------|
| <b>1</b>  | <b>Goals and motivation</b>                               | <b>3</b>  |
| 1.1       | Vision and approach . . . . .                             | 3         |
| 1.2       | Current achievements . . . . .                            | 3         |
| 1.3       | Relations to tasks in Annex 1 . . . . .                   | 3         |
| <b>2</b>  | <b>Relation to other (sub)workpackages in PEPITO</b>      | <b>3</b>  |
| <b>3</b>  | <b>Survey of current state-of-the-art</b>                 | <b>4</b>  |
| <b>4</b>  | <b>Subworkpackage contributions</b>                       | <b>5</b>  |
| <b>5</b>  | <b>Overview of Deliverable</b>                            | <b>6</b>  |
| <b>6</b>  | <b>Thread Migration Model</b>                             | <b>6</b>  |
| <b>7</b>  | <b>Programming Patterns</b>                               | <b>7</b>  |
| 7.1       | Go: Self Migration . . . . .                              | 7         |
| 7.2       | Pull: Execution Locator . . . . .                         | 8         |
| 7.3       | Push: Execution Mediator . . . . .                        | 9         |
| 7.4       | Summary . . . . .   | 9         |
| <b>8</b>  | <b>Migration Primitives</b>                               | <b>10</b> |
| 8.1       | Migration and Thread States . . . . .                     | 11        |
| 8.2       | Migration Primitive Improvement . . . . .                 | 12        |
| 8.3       | Programming the Abstractions . . . . .                    | 13        |
| 8.4       | Thread References and Thread Migration Protocol . . . . . | 14        |
| <b>9</b>  | <b>Sketch of Implementation Architecture</b>              | <b>15</b> |
| 9.1       | Oz and Mozart . . . . .                                   | 15        |
| 9.2       | Thread Marshaling . . . . .                               | 15        |
| 9.3       | Migration Protocol Implementation . . . . .               | 16        |
| <b>10</b> | <b>Initial Feasibility Test</b>                           | <b>16</b> |
| 10.1      | Application Domain . . . . .                              | 16        |
| 10.2      | Application Scenario . . . . .                            | 16        |
| 10.3      | Results . . . . .   | 17        |
| <b>11</b> | <b>Work Status</b>  | <b>18</b> |

## 1 Goals and motivation

This report is a deliverable related to the task Mobile computation [KTH:12] in workpackage 3 of the Technical Annex.

### 1.1 Vision and approach

The vision of this workpackage is to develop a model for strong mobility with concomitant programming abstractions for *truly* mobile applications. *Strong mobility* allows to migrate execution state in contrast to *weak mobility* where only data and code is subject to mobility. Strong mobility is essential for truly mobile applications where reconstruction of execution states is either difficult or even impossible.

There is a strong connection between mobile applications and peer-to-peer applications in that mobile applications require information on nodes (sites) for migrating computations. We expect a synergy in future work where sites for migration are managed by a peer-to-peer system (such as by distributed hash tables, for example DKS [18]).

Strong mobility is approached by developing thread-based mobility as a model where threads are the units of mobility. This model is a natural extension of a distributed programming model based on automatic synchronization through dataflow variables. The programming language Oz implemented by the Mozart system follows this model of distributed programming.

We identify a set of abstractions (`Go`, `Pull`, and `Push`) for explicit thread migration. A common implementation infrastructure based on thread replication and migration managers is developed that provides the necessary support for the programming abstractions.

### 1.2 Current achievements

There are three main achievements:

1. A model for thread-based migration together with abstractions for programming applications using strong mobility.
2. An implementation architecture based on thread replication and mobility managers for providing thread-based mobility.
3. An initial feasibility test based on a standard scenario for mobile execution. The scenario is structured as a simple peer-to-peer system.

### 1.3 Relations to tasks in Annex 1

This subworkpackage corresponds to the task on

- ▷ mobile computation

of Annex 1.

## 2 Relation to other (sub)workpackages in PEPITO

**Distributed directory service (WP2) and Oz (WP3)** The test scenario application used to assess feasibility makes two major simplifications. Firstly, nodes where mobile computations can move are assumed to be

completely known. We intend to make more realistic experiments by using the support provided by the Oz-based peer-to-peer platform for maintaining nodes where computations can move.

Secondly, distributed knowledge on locations is assumed to be directly available. Here we will be able to take advantage of the distributed directory service in more realistic experiments.

**Resource control, versioning, and modularity (WP3)** This deliverable focuses on model, programming abstractions, and implementation architecture for thread-based mobility. One aspect that is not tackled so far is how mobile computations can control their resource access while migrating. The work in this workpackage will provide us with models how to extend our approach with resource control and in particular with dynamic binding of resource names to resources.

**Basic services (WP4)** Thread replication as one cornerstone of our implementation architecture uses the distribution support available in Mozart. By that it will directly use what is developed in this workpackage.

### 3 Survey of current state-of-the-art

The current state-of-the-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 the 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.

**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 [14]. 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.

**Obliq.** Obliq is an untyped object-oriented interpreted language with distributed lexical scope developed at DEC by Cardelli [1]. 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.

**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.

**JoCaml, Join Calculus, and Ambient Calculus.** JoCaml [3] is an implementation of the Join-Calculus, which is a reformulation of  $\pi$ -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 [2] are very closely related as pointed out in [5] 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 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.

**D'Agents.** Developed at the University of Dartmouth, D'Agents [7] is a multi-language system consisting of Agent Tcl [6], 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.

**ARA.** Developed at the University of Kaiserslautern, Ara [19] 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.

## 4 Subworkpackage contributions

This subworkpackage makes the general contribution of a model, programming abstractions, and sketch of an implementation architecture for thread-based strong mobility for a distributed dataflow language. In detail, the contributions are as follows:

**Thread-based Mobility.** This deliverable 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 insists on the fact that mobility is under explicit control of the programmer.

**Programming Abstractions and Application Scenarios.** Based on thread-based mobility, this deliverable contributes programming abstractions (`Go`, `Push`, and `Pull`) which capture common programming idioms in the construction of mobile applications. It is shown how the abstractions can be used in prototypical application scenarios.

**Sketch of Implementation Architecture.** All programming abstractions can be implemented by simple primitives for thread mobility. The deliverable identifies thread replication together with migration managers as basic building blocks for an architecture to implement thread-based mobility.

**Initial Feasibility Test.** The programming abstractions have been implemented in a prototype on top of the Mozart programming system. The deliverable conducts a first feasibility test by running a standard test scenario for mobility [8].

## 5 Overview of Deliverable

The next section presents the migration model and details implicit and explicit migration. Section 7 introduces migration abstractions by presenting several application scenarios. The following section presents a migration primitive as foundation for the migration abstractions and how thread states are reflected during migration. An implementation architecture for the model in the Mozart programming system is sketched in Section 9 followed by an initial feasibility test in Section 10.

A paper reporting on this work is currently under submission.

## 6 Thread Migration Model

In this paper 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 statements 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 we refer the reader to [20].

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, we refer the reader to [22, 10, 11].

**Strong Mobility.** Strong mobility is known from operating systems as a process migration. In our model, the unit 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 source site.

When a thread migrates all data values that are accessed by the thread stack are migrated as well. The exceptions are other threads that are referenced by the migrating thread and resources. We define a *resource* as a data structure whose use is restricted to one site (such as file handles, for example). Here, we assume that resources are ubiquitous and dynamically rebound when threads are migrated. A thread is executed at a *location* which we refer here as a *site*. Migration is always performed between two sites: *source site* and *destination site*. The thread migration implies migration of 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 some other thread.

**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.

**Implicit Migration.** Implicit migration is transparent (invisible) to the application programmer. That means that a 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.

**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. Therefore, we consider explicit migration here.

**Thread References.** A thread after creation is known only at the *source* site. Later on, a thread reference can be passed to other sites. Thread references are globally unique. Passing a thread reference between sites will *not* trigger thread migration. Migration is requested by calling one of the migration abstractions to be discussed. That is, distribution of a thread reference does not include the migration of the associated thread.

## 7 Programming Patterns

This section introduces common programming patterns for mobile applications and identifies migration abstractions and primitives. The code fragments used are presented in Oz [9, 21]. To help the reader,  $\{P \text{ Arg}\}$  calls the procedure  $P$  with the argument  $\text{Arg}$ . `thread  $s$  end` creates a new thread which executes the statement  $s$ . By  $x$  `in` a new dataflow variable  $x$  is introduced.

### 7.1 $G_0$ : Self Migration

The abstraction  $G_0$  is useful for pro-active 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.

## 7.2 Pull: Execution Locator

The abstraction `Pull` is used to move execution from the *source site* to the *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 amount of network-traffic produced by the thread). A matching code example is presented below:

```

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

```

Please 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 a site as single argument.

Table 1: Use of thread and site references

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

### 7.3 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}
  HighestLoadSite LowestLoadSite
  LoadList Thr
in
  LoadList = {GetLoads SiteList}
  HighestLoadSite = {Max LoadList}
  LowestLoadSite = {Min LoadList}
  Thr = {ChooseThread HighestLoadSite}
  {Push Thr LowestLoadSite}
end

```

Please note that the `Push` abstraction takes two arguments, the thread to be migrated and the destination site. It can be invoked from any site.

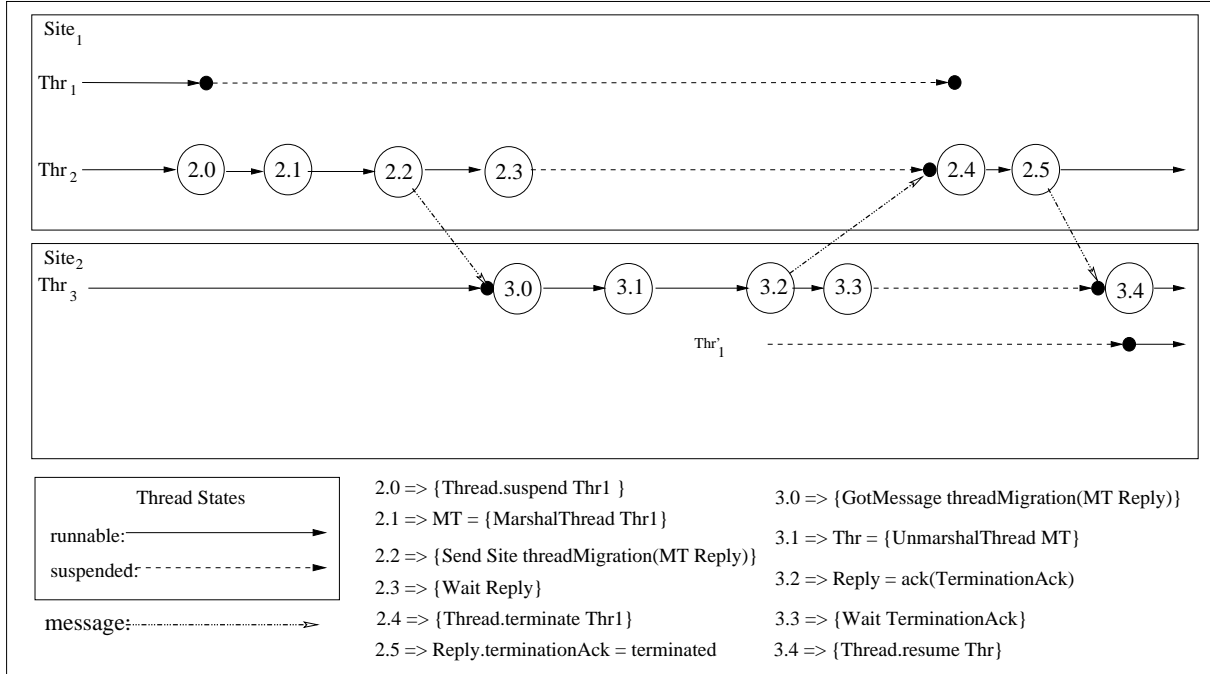
### 7.4 Summary

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

- ▷ 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
- ▷ Destination Site
  - rebuilds the thread on the destination site.

The use of thread and site references is summarized in Table 1. All abstractions have in common that they require representations of both threads and sites in the programming language. Threads are available as thread references (as already discussed in Section 6).

For sites in our model we take the following approach: A *site* represents a process where computation proceeds, for example a virtual machine run by an operating system process. It is represented by a unique identifier created when a process is forked. The *site* identifier can be passed as a function argument or returned as a result from a function call. It can be sent as a message (or a part of the message) to other sites. It can be used in the migration abstractions `Push` and `Go`.

Figure 1: Execution of  $\{ThreadReplicate Thr_1 Site_2\}$ 

## 8 Migration Primitives

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 threads. Thread migration is done by sending and receiving of messages between migration managers. The information needed to perform migration of a thread  $T$  is located at the source site of  $T$  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.

**Source Site** The thread  $T$  is *suspended*, its execution state is collected, serialized, and sent to the destination site.

The migration manager  $MM_s$  waits until an acknowledgment on 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  $MM_d$  is informed that the *replica* can resume.

**Destination Site** When the serialized thread is received by  $MM_d$ , it rebuilds the thread  $T_r$  from the network representation (*unmarshal* the thread). After rebuilding, an acknowledgment message is sent to  $MM_s$ .

Then  $MM_d$  waits on a confirmation that the *original thread* has been terminated. When termination of  $T$  is confirmed,  $T_r$  is *resumed*.

$MM_s$  and  $MM_d$  synchronize twice during thread migration. The first synchronization is on the *replica-thread creation* and the second synchronization is on the *original-thread termination*. Figure 1 shows the

interaction and the synchronization between migration managers.

A code example of thread replication is shown below:

```

proc {ThreadReplicate Thr Site}
  Reply MarshThr
in
  {Thread.suspend Thr}
  /* Send serialized thread and synchronization variable */
  MarshThr = {MarshalThread Thr}
  {Send Site threadMigration(MarshThr Reply)}
  /* Wait on acknowledgment */
  {Wait Reply}
  /* Terminate thread */
  {Thread.terminate Thr}
  /* Acknowledge termination */
  Reply.terminationAck = terminated
end

```

A code example of a migration manager is presented below:

```

proc {WaitForThreads S}
  for Msg in {GetMessage S} do
    case Msg
    of threadMigration(MarshThr Reply) then
      Thr TerminationAck
    in
      Thr = {UnmarshalThr MarshThr}
      /* Inform the source site about replica */
      Reply = threadReceived(terminationAck:TerminationAck)
      /* Wait on termination */
      {Wait TerminationAck}
      {Thread.resume Thr}
    [] migrate(Thr Site Sync) then
      {ThreadReplicate Thr Site}
      Sync = success
    end
  end
end

```

## 8.1 Migration and Thread States

A thread can be in 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 behavior description for each case follows:

**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*,

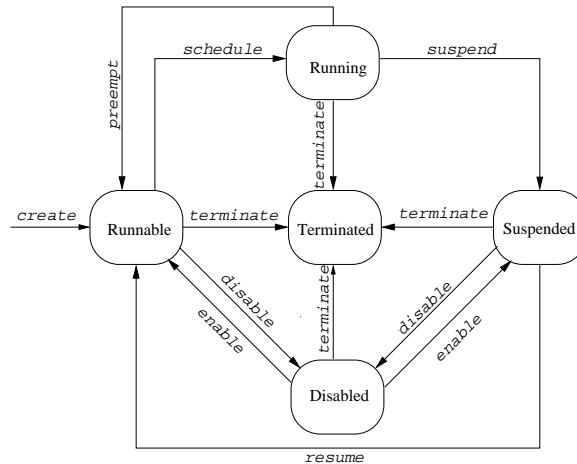


Figure 2: Thread States

and the *replicated* thread which was created in the *suspended* state is *resumed* (that is, added to the runnable queue at the *destination* site).

**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 [10].

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

## 8.2 Migration Primitive Improvement

The model used for the replication primitive *ThreadReplicate* is based on synchronous message passing (*rendezvous*). We use this double synchronization due to dataflow synchronization used for thread synchronization. The migration primitive can be easily improved by avoiding that the destination site needs to synchronize on thread termination.

We introduce a new thread state called *disabled* (see Fig. 2). A thread becomes *disabled* after the thread operation  $\{\text{Thread.disable Thr}\}$  is called. The disabled thread cannot become *runnable* or *suspended* until it is *enabled* with the operation  $\{\text{Thread.enable Thr}\}$ . An *disabled* thread can be *terminated*.  $\{\text{Thread.disable Thr}\}$  and  $\{\text{Thread.enable Thr}\}$  are applied only on the site where the thread is executed (that is, the operations require no support for distribution). For example, if the *replica* is *enabled* on the destination site, the *original* is still *disabled* on the source site.

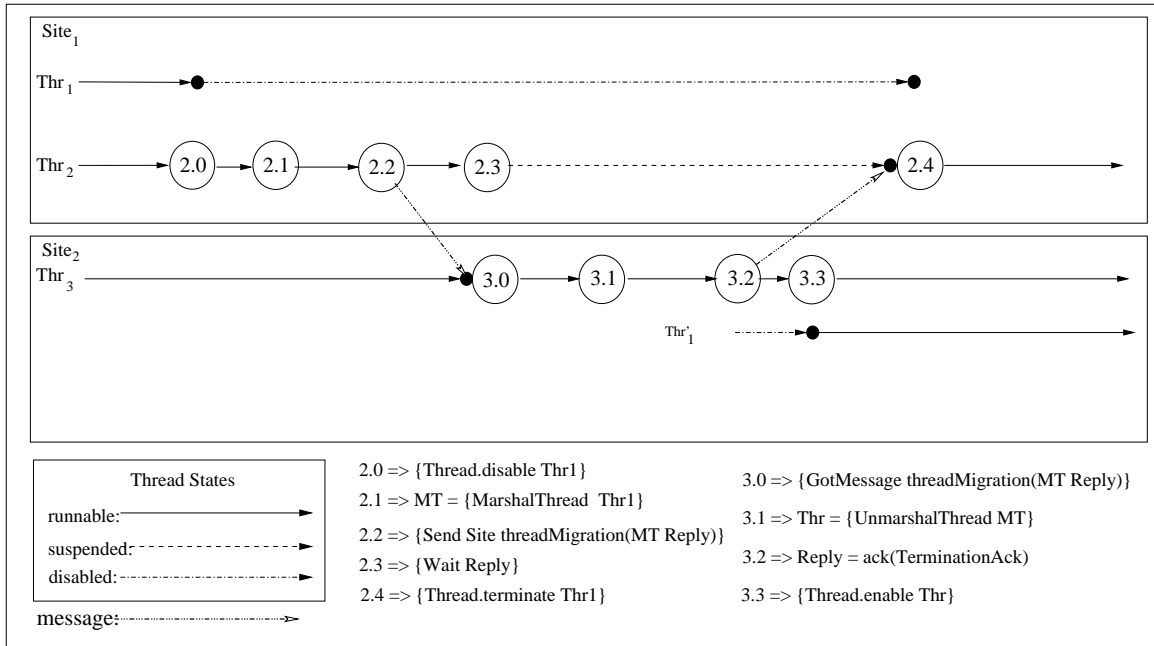


Figure 3: Execution of  $\{ThreadReplicate Thr_1 Site_2\}$  with *disabling*

Instead of using a dataflow variable to suspend a thread, the thread is *disabled*. The source site still waits on the thread reception confirmation from the destination site, but the destination site does not wait on the thread termination confirmation. Thus, the *replicated* thread can be *enabled* and can continue with execution before the *original* thread has been terminated. In the case of high network latency this can lead to considerable improvement (see Fig 3).

On the other hand, there is a drawback of this solution as well, as it might be too optimistic. In case the acknowledgment on thread reception is lost, the source site is not informed that migration has succeeded and raises an exception.

In general, the failure behavior of thread replication needs to be studied in more detail and will be addressed in future work.

### 8.3 Programming the Abstractions

With the help of migration managers and thread replication as discussed above, the abstractions introduced in Section 7 can be programmed easily.

**PUSH Abstraction.** The PUSH abstraction can be used both from the source site or from some third site. The *distributed scheduler* presented in the previous section uses PUSH to migrate a thread that is not on 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 Thr Site}
  ThrHomeSite Sync
in
  ThrHomeSite = {GetSite Thr}
  {Send ThrHomeSite migrate(Thr Site Sync)}

```

```

    {Wait Sync}
end

```

**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:

```

proc {Go RemoteSite}
  Sync Thr
in
  Thr = {Thread.this}
  thread
    {Thread.suspend Thr}
    Sync = done
    {Push Thr 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.

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

```

proc {Pull Thr}
  MySite MyThr
in
  MyThr = {Thread.this}
  MySite = {GetSite MyThr}
  {Push Thr MySite}
end

```

Here, `MyThr` is the thread in which `Pull` executes. `MySite` is the *destination* site.

## 8.4 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 the passed reference can be used to perform network-wide operations on threads. These operations are network transparent and their semantics remain the same as if they were 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. 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). 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:

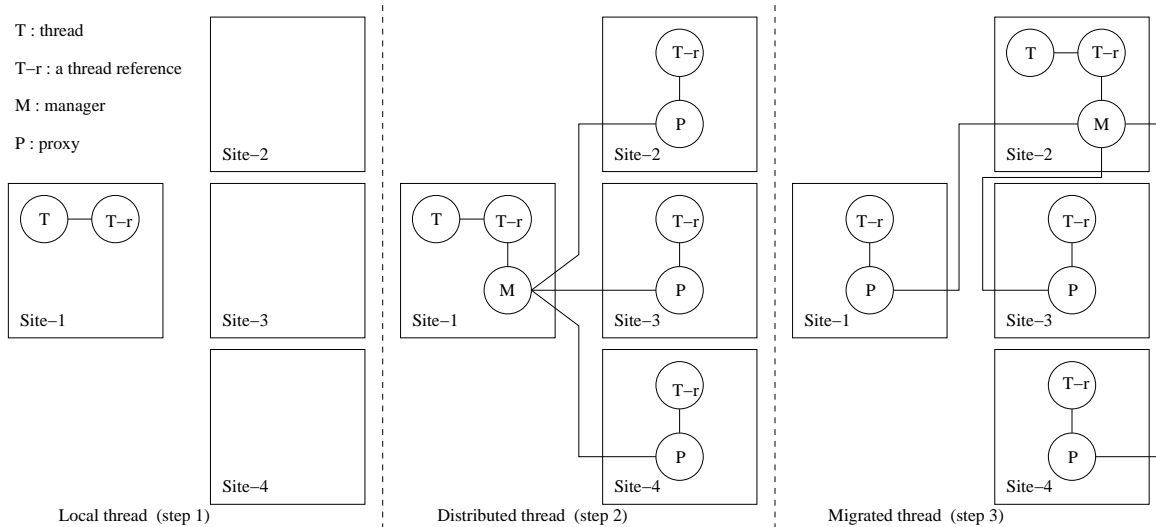


Figure 4: Thread References and Migration

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

## 9 Sketch of Implementation Architecture

In this section we describe how strong mobility can be implemented as an extension to the Mozart programming system [4]. First, we give a short description of the Oz programming language, and the Mozart programming system. Then we describe the implementation of the extensions needed for the replication primitive.

### 9.1 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 [17]. Thus, a distributed application can be developed completely in a centralized setting [11]. 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 [22, 10] soon to be provided by the generic DSS system [16, 15]. When a data entity is sent between two Mozart processes, its memory representation is transferred to network representation at the sender site and the memory representation is created at the receiver site upon the reception. Translation to network representation is called *marshaling* and translation back is called *unmarshaling*. The term *serialization* is also used.

### 9.2 Thread Marshaling

The replication primitive is based on marshaling and unmarshaling of threads. Thus, the system is extended with methods for thread marshaling and unmarshaling. A thread is a stack of statements. Each statement

consists of a program counter for code to be executed and an environment of local variables (basically, a thread is a stack of closures). To increase efficiency of execution, a program counter points to the instruction to be executed next.

One important design decision in Mozart, for making marshaling simple, is that only values are marshaled. To comply with that, program counters in stack entries are translated to a pair of procedure and relative offset of the *PC* from the start of the procedure. This construction ensures that only complete procedures are marshaled and also that absolute and site-specific addresses are translated into relative and hence site-independent values.

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.

### 9.3 Migration Protocol Implementation

Sites have uniquely identified port used for thread migration and we refer to this port as migration port. The distributed thread has an access structure, used for thread migration, which contains a reference to the migration port of the thread's current site. Thus, when a thread migrates the thread's access structure must be updated with the current migration port. Migration ports are implemented on a high level reusing ports in Oz [20] and [4]. Oz ports are influenced by ports in AKL [13], [12].

## 10 Initial Feasibility Test

In this section, we present a first feasibility test of the model implemented in the Mozart programming system. We compare the performance of the mobile agent model built on top of thread-based mobility with the client-server model. First, we discuss a possible application domain that is well suited for the mobile agent model. Then we describe the application scenario used for evaluation and present and discuss results.

### 10.1 Application Domain

It is clear that agent migration is a time consuming task and that it costs much more than RPC, RMI, and synchronous message sending. On the other hand, agent migration implies transfer of not only data but "intelligence" as well. The major part of distributed applications are client-server based. Servers provide usually a set of low level operations which can be invoked independently from each other. Clients are applications that combine these low level operations to provide a specific service. Each server operation has to be requested separately. Total time needed to perform some specific task contains of the time that the server needs to perform each low level task and the time needed to deliver requests and replies. Complex clients tasks contain many requests and the total time is directly proportional to network latency. Mobile agents can avoid network latency by adopting servers to perform specific tasks without compromising servers generality.

### 10.2 Application Scenario

Section 7 presents several examples of mobile applications. In the first example a mobile agent moves between sites and collects site-specific information. A similar scenario is used for evaluation. The scenario is inspired by the paradigmatic mobile agent evaluation (see [8]).

The task of the application is to collect at a customer site  $S_c$  a list of hotels with phone numbers in one town. Two database servers must be consulted. The hotel database server  $H_{ds}$  at the site  $S_{H_{ds}}$  to obtain a list

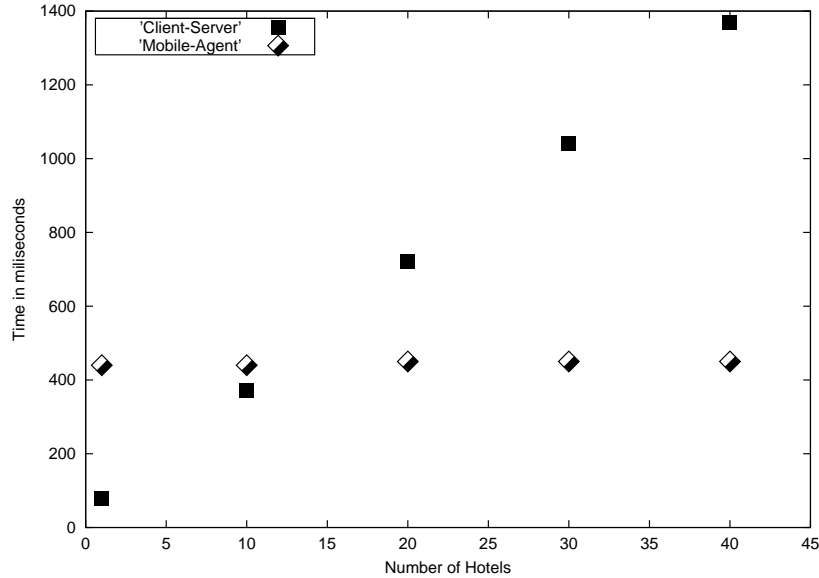


Figure 5: Mobile Agent vs. Client-Server in the Information Collection

of available hotels in the specified town, and the phone database server  $P_{ds}$  at the site  $S_{P_{ds}}$  to obtain a phone number of each hotel from the list, one at a time.

**Client-Server Solution.** In order to get the list of hotels for a specified town, the client sends a request ( $req_h$ ) to the server  $H_{ds}$ . After the list has been received, the client sends one request ( $req_p$ ) per hotel to the server  $P_{ds}$  to obtain telephone numbers. That means that the client sends as many requests as hotels in the list plus one. Consequently, the total time  $time_{total}$  needed to perform the complete task is defined by the time for  $req_h$  plus the time for  $n * req_p$ , where  $n$  is the number of hotels.

**Mobile Agent Solution.** The mobile agent moves from the customer site  $S_c$  to the site  $S_{H_{ds}}$  and requests a list of hotels locally. After the list has been received, the agent moves to the site  $S_{P_{ds}}$  and queries the server  $P_{ds}$  for telephone numbers. When all phone numbers are collected the agent moves back to the site  $S_c$  and returns the result.

### 10.3 Results

In the feasibility test, we assume that the database operations have constant cost. The customer is sited on a computer in Germany (Universität des Saarlandes) and the servers are sited on computers in Sweden (Swedish Institute of Computer Science, SICS). We change the number of returned hotels and measure the total time  $time_{total}$  in both solutions. Figure 5 summarizes the results. We see that only in the case with very small numbers of hotels, the Client-Server solution is more efficient than the mobile agent solution. In all other cases the mobile agent solution is much more efficient and it does scale.

## 11 Work Status

We have presented a model and an implementation of strong mobility in a distributed lightweight concurrent system based on dataflow variables. Important 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. A first sketch of an implementation architecture for the Mozart programming system is described and a first feasibility test is presented.

## References

- [1] L. Cardelli. Obliq: A language with distributed scope. Technical report, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA, Nov. 1994.
- [2] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.
- [3] S. Conchon and F. L. 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.
- [4] D. Duchier, L. Kornstaedt, T. Müller, C. Schulte, and P. Van Roy. *System Modules*. The Mozart Consortium, [www.mozart-oz.org](http://www.mozart-oz.org), 1999.
- [5] C. Fournet and A. Schmitt. An implementation of ambients in JoCaml. In *Proceedings of the 5th ECOOP Workshop on Mobile Object Systems (MOS'99)*, Lisbon, Portugal, 1999.
- [6] R. S. Gray. Agent Tcl. *Dr. Dobb's Journal of Software Tools*, 22(3):18–??, 1997.
- [7] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. Mobile agents: Motivations and state-of-the-art systems. Technical Report TR2000-365, Dartmouth College, Hanover, NH, Apr. 2000.
- [8] D. Hagimont and L. 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.
- [9] S. Haridi and N. Franzén. *Tutorial of Oz*. The Mozart Consortium, [www.mozart-oz.org](http://www.mozart-oz.org), 1999.
- [10] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.
- [11] S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.
- [12] S. 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.
- [13] S. Janson, J. Montelius, and S. Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, USA, 1993.
- [14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.
- [15] E. Klintskog, Z. El Banna, P. Brand, and S. Haridi. The design and evaluation of a middleware library for distribution of language entities. In V. A. Saraswat, editor, *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference*, volume 2896 of *Lecture Notes in Computer Science*, pages 243–259, Mumbai, India, Dec. 2003. Springer-Verlag.

- [16] E. Klinskog, Z. El Banna, P. Brand, and S. 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, Jan. 2004. IEEE Computer Society Press. To appear.
- [17] Mozart Consortium. The Mozart programming system, 1999. Available from [www.mozart-oz.org](http://www.mozart-oz.org).
- [18] L. Onana Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In *The 3rd International Workshop on Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID 2003)*, pages 344–350, Tokyo, Japan, May 2003. IEEE Computer Society.
- [19] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In R. Popescu-Zeletin and K. Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, Apr. 1997. Springer Verlag.
- [20] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
- [21] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2003. Available March 2004.
- [22] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, Sept. 1997.