Adaptable Object Migration: Concept and Implementation

Wolfgang Lux

GMD, Institute for System Design Technology
D 53757 St. Augustin, Germany

November 7, 1994

Abstract

Migration is one example of the insufficiently used potentials of distributed systems. Although migration can enhance the efficiency and the reliability of distributed systems, it is still rarely used. Two limitations contained in nearly all existing migration implementations prevent a widespread usage: migration is restricted to processes and the migration mechanism, i.e. the way state is transferred, is not adaptable to changing requirements.

In our approach, migration is an operation provided by every object of any type. Triggered by higher level migration policies, the object migrates itself using an object-specific migration mechanism. Changing requirements are handled by higher level migration policies that adapt migration by exchanging the object's mechanisms.

Adaptable migration was implemented within the BirliX operating system. Different migration mechanisms are accomplished by different meta objects, which can be attached to other objects. If an object has to be migrated, the meta object does the migration. Changing environmental requirements are handled by exchanging the meta object. As a result, each object has its own migration mechanism. The approach has been examined by implementing a couple of well-known migration mechanisms via meta objects. This paper describes the meta object implementation of the Charlotte migration mechanism.

1 Introduction

Although efficient networks provide a suitable base for distributed systems, distributed systems do still not exploit all of their potentials. Migration, i.e. dynamically changing the location of objects [1], is one example of these insufficiently used potentials. Migration can be used to enhance the efficiency and the reliability of distributed systems:

- Administration:
  When maintaining or replacing computers, their services can be migrated to other hosts without interrupting usage [TH92].
- Communication support:
  To minimize communication overhead, communication partners are migrated to hosts at close range [Sch90]. Collecting communication partners on one host renders predictable execution times for real time critical applications [RS84].

[1] Within this paper, we only consider migration of coarse-grained operating system objects.

- Fault avoidance:
  To meet security demands, documents that become security sensitive are migrated to trustworthy computers. To enhance the availability of replicated objects, individual replicas can change their location [Fen93].

- Resource usage:
  To balance processor loads, processes are migrated from highly loaded computers to lowly loaded ones [BS85]. To enable mobile computing, objects are migrated from/to mobile stations before decoupling from the system and migrated back after recoupling [BAI93].

Although a lot of applications can profit from migration, only a few systems provide a migration feature. In most cases, these systems are special solutions for load-balancing by process migration. Restricting migration to processes impedes the usage of migration in many applications. For an application with a high degree of file i/o, separating the processes from the accessed files is counter-productive because of the additional communication overhead. Migrating the processes together with all their files avoids this overhead.

Requirement: A system must provide migration for objects of different types.

The Charlotte operating system [AF89] introduced the first step in making migration adaptable to various applications by dividing migration into a migration policy and a migration mechanism. The policy makes the decision about when and where to migrate an object; the mechanism transfers the resources belonging to an object from one host to another. Thus, a policy does not need to care about resources, but can use the object migration provided by the mechanism. The goal is a generic, policy-neutral mechanism supporting various policies. Unfortunately, a single mechanism is not suitable for all applications. For example, files may be migrated in a fault-tolerant way to avoid loss of information. The overhead needed for fault-tolerance is undesirable for fast process migration. An inappropriate mechanism can even be counterproductive, i.e. the Accent [Zay87] lazy migration mechanism leaves residual dependencies, which conflict with fault-tolerance requirements.

Requirement: A system must support the coexistence of different migration mechanisms.

As systems are still evolving, new requirements will arise during their life time.

Requirement: A system must support the dynamic integration of new migration mechanisms.

To get a widespread usage, a migration system must offer sufficient performance. In existing implementations, the whole state of an object is transfered. The use of semantic knowledge about the object state can reduce the the amount of transferred state. For example, the data cached within a database application does not need to be transferred, since the cache can be rebuild at the destination system.

Requirement: A system must incorporate semantic knowledge for migration.

To make a decision, migration policies need information about participating hosts and objects. The necessary information depends on the decision criteria of individual policies. Different policies need different information. Newly developed policies probably need new information.

Requirement: A system must support the dynamic integration of mechanisms for information collection.

Adaptable migration is an approach complying with all of these requirements. Section 2 presents the concept of adaptable object migration. Migration of an object becomes adaptable by attaching a meta object containing an individual migration mechanism. The BirliX operating system which has been used for the implementation is introduced in section 3. Section 4 describes the implementation of adaptable migration. As well as selected interfaces for migration, a migration meta object and some measurements are also presented.

2 Adaptable Migration

Our approach is based on the adaptable object model described in [SHKL94]. The execution environment consists of a set of hosts interconnected by a communication system. At each host there is an object management system managing the local objects. Objects have *functional* and *non-functional* properties. Functional properties are defined by the *type* of an object, non-functional properties are defined by the object's *infrastructure*. The type determines, for all of its objects, the internal data structures (attributes) and the operations (methods) used to manipulate the data structures. An infrastructure is a run-time environment for an object and the execution of its methods. It provides communication, encapsulation and non-functional properties. For example, the infrastructure of an operating system object defines non-functional properties such as persistence, protection and migratability. The adaptability of non-functional properties to environmental requirements is achieved by dispersing parts of its infrastructure into a set of *meta objects* [TY92], as shown in figure 1. A meta object is an object containing the implementation of a non-functional property. Non-functional properties of an object are adapted to specific requirements by

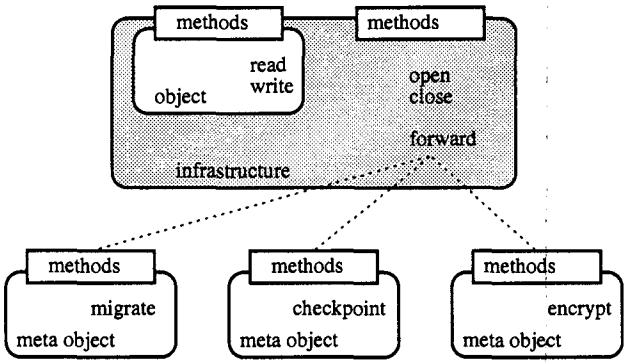

Figure 1: An Object with its Meta Objects

substituting its meta objects.

Migration is the transfer of an object from one object management system to another. The object management system provides a type-independent standard implementation for migration and information collection. Thus, objects of different types can be migrated without support from its type. This is possible because the state of an object is completely represented within its type and its infrastructure. The object management system migrates the object to the specified destination host by:

- Creating a transferable representation of the object at the source host,
- sending the representation to the destination host and
- reconstructing the object from the transferred representation.

Adapting migration to changing environmental requirements is achieved by meta objects. Different migration mechanisms and information collectors are stored within different meta objects. Attaching a meta object to a single object enables object-specific migration and information collection. Higher-level migration policies handle changing requirements by substituting attached meta objects.

When the migrate method of an object is called, its infrastructure forwards the call to the attached migration meta object. To be able to create a transferable representation, the meta object needs access to the internal components of the object. To do this, the object's infrastructure provides an internal interface to the meta object. The offered operations, e.g. suspension of activities and extraction of kernel state, are comparable to those provided by micro kernels.

The information collector collects information about its attached object. As the state of an object mostly depends on its communication, most information can be got by observing the communication. To do this, the infrastructure provides the sequence of called methods to the attached information collector.

In many cases, knowledge about the type implementation can make migration more efficient. One example is reducing the amount of state to be transferred. Adapting an object to the resources of the destination host is another potential of type-specific knowledge. For example, the number of parallel activities within an object is changed to the number of processors at the destination host. To use this knowledge, a type description should provide an internal interface to the meta object. Type-specific migration is an option for a type programmer, he can still rely on type-independent meta migration.

The concept of adaptable migration is open to several implementations; though each implementation must treat the following topics:

- Splitting objects into system/user parts:
  Each object known to the object management system has an anchor within the system, such as its binding to physical resources. Consistency, flexibility and efficiency are criteria used to determine which parts of an object are system defined and which are administrated at user level. For user level migration, it is important to know, which parts of the system state can be extracted by the user part during migration.

- Defining the interfaces of meta objects:
  A mechanism in a meta object depends on the calling parameters, the information provided by the object's infrastructure and the object management system. While the infrastructure must provide operations for accessing the object's resources, the object system must provide operations for transferring the contents of these resources.

- Communication with meta objects:
  Efficiency of communication between an object and its meta objects is important. Meta objects can be implemented as independent objects or integrated into the attached objects. To get high availability, an attached meta object should reside on the object's host, i.e. it should be migrated along with the object.

3 The BirliX Type System

Adaptable migration was implemented within the BirliX operating system developed at GMD. The goal of BirliX is the support of fault-tolerant and secure applications in a distributed environment [KHKL90]. The BirliX kernel is basically an abstract data type management system providing mechanisms for the definition and instantiation of *BirliX types*, the identification of instances [2] and communication between instances. BirliX types are very similar to Eden types [ABLN85]. All BirliX types share a common set of type-independent attributes and type-independent methods inherited from a kernel-defined *primary type*. The type system and the primary type together provide the infrastructure of an instance. The functional extension of a BirliX system is easily done by adding new BirliX types.

An application in BirliX consists of a cluster of communicating instances. Instances communicate via instance calls, which are implemented by network transparent *remote procedure calls* (RPCs).

3.1 Instance Features

BirliX instances are autonomous units, their state can only be changed by calling methods of the instance. The change of state is made by the instance itself.

BirliX instances are persistent. The lifetime of an instance depends on internal activities and references in name servers. The BirliX type system manages an instance as long as there is at least one internal activity or one external reference. Name server instances are used to locate an instance. They map symbolic names of instances to their unique identifiers. Additionally, they provide hints about their current location. At any

[2] The term 'instance' is used instead of 'object' because of the restricted inheritance.

time, an instance resides at one host. Executing the primary method `Migrate` changes the location of an instance.

Security in BirliX is based on the encapsulation of instances and on authenticated message transfer [KH90]. Access control lists and subject restriction lists are the current mechanisms to control access. Access control lists specify access rights of human users, instances and types for instance methods. When receiving a call, the instance checks the sender identification and the called method against the access control list. Subject restriction lists are used to restrict the sight of an instance to a certain set of instances. The authenticity of messages assures that sender identifications are not falsified.

In order to support fault-tolerant applications, the state of an instance can be saved in a checkpoint. During recovery, the saved state is restored. Interrupted communication connections are reestablished by a rebinding mechanism in the type system.

3.2 Teams

A BirliX instance is implemented by a type-independent general implementation structure called a *team* (Figure 2). A team provides the storage and computing resources needed by an instance in its active phase. A team is a collection of threads sharing an address space and a collection of segments mapped into that address space. Communication connections to other teams are maintained in *access descriptors.*

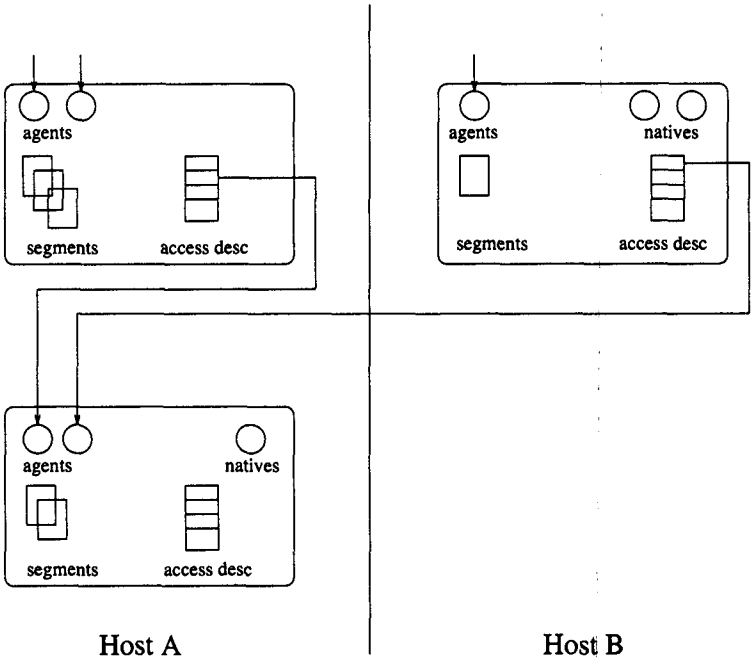

Figure 2: Communicating Teams

Threads are sequential activities running in parallel with each other. Each thread is bound to an address space and several threads can be active at the same time. Within an address space, threads communicate via shared memory synchronized by Hoare monitors. Communication crossing the address space is done by synchronous message passing. Each thread consists of a *kernel coroutine* and *a user coroutine.* The kernel coroutine executes the methods of the type system and the user coroutine those of the BirliX type. Depending on their role, threads are divided into *agents* and *natives.* Each calling instance (*client*) is represented within the called instance by an agent. This agent receives the messages of its client, checks the method against the access control list and executes the method. Natives are used for internal activities running asynchronously

to agents and other natives. For example, a native executes the Unix program in an instance of type Unix process.

Segments are an abstraction of storage resources provided by the memory management system. Segments are comparable to memory objects in Mach [TR86] or segments in Chorus [R+88]. Each segment possesses a unique identifier. The *root segment* is the root of all segments belonging to an instance. It contains a descriptor for the team and has the same unique identifier as the corresponding instance.

Communication connections between instances are managed at the client side in access descriptors. An access descriptor contains a network transparent pointer to an agent and protocol information about the connection. The one-to-one relation between an access descriptor and an agent makes the rebinding of interrupted connections possible.

3.3 Instance Manager

At each host, there is one instance manager, which is responsible for the administration of the local instances. The manager maintains a descriptor for each team. A team descriptor contains information about a team, such as the team state, descriptors for the agents and natives, an access control list, a subject restriction list and identifications of the segments belonging to the team. When an instance is created from a BirliX type, the manager creates a team with one agent. The kernel coroutine of this agent creates an address space and the needed segments. The code segment is built using the type description. Then, all segments of the instance are mapped into the address space. Control is transferred to the user coroutine which performs the type-specific initialization of the instance. Finally, the agent sends a reply message to the client and waits for the next message of the client (figure 3).

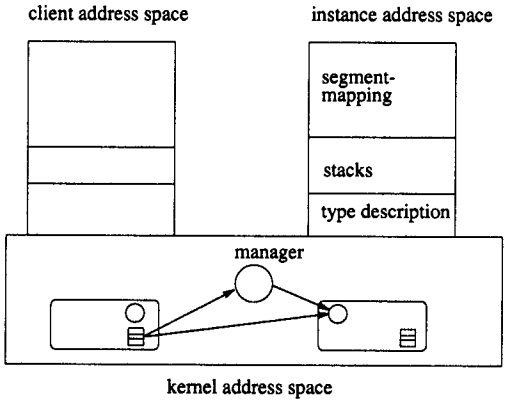

Figure 3: Establishing a Communication Connection

The manager creates a new agent for each additional communication request. On receiving a message, the agent checks the method name in the message against the access control list. If the method name specifies a primary method, it is executed by the kernel coroutine; type-specific methods are forwarded to the user coroutine. When the primary method to delete the communication connection is called, the agent is deleted by the manager. Natives are created and deleted during the execution of type-specific methods. The manager passivates an instance with external references when all agents and natives are deleted. Any further communication request causes the manager to reconstruct the team using the passive representation in the segments. If there is no external reference during passivation, the manager deletes the instance along with all of its segments.

4 Migration in BirliX

4.1 Primary Type Migration

The BirliX type system provides type-independent migration of instances. Using the primary method **Migrate**, an instance is migrated within a network of homogeneous BirliX hosts. Migration is transparent to the user and to the type programmer: a client uses a single method to migrate instances of different types and the programmer of a type does not need to care about migration. To aid the decision about which instance should be migrated, each instance provides an additional primary method **MigInfo**. **MigInfo** returns relevant information about the instance. Like all primary methods, the execution of **Migrate** and **MigInfo** is protected by the access control list of the instance.

During Migration, the current state of an instance is transferred to the destination host. As this state is given by the corresponding team, instance migration is done via team migration. Figure 4 shows the effect of team migration applied to figure 2. For team migration we need a team checkpoint, which is a

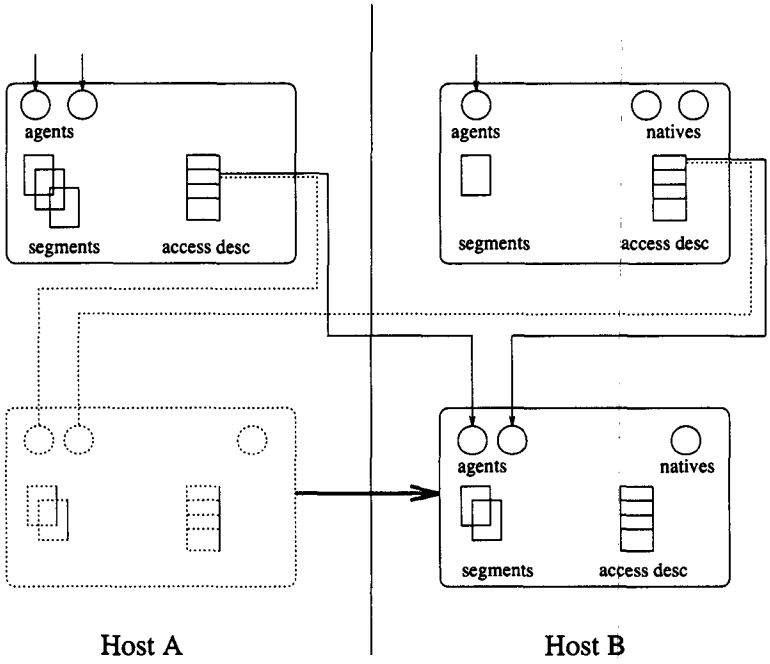

Figure 4: Team Migration

representation of its state on backup storage. Team checkpointing is an existing mechanism in BirliX used to support fault-tolerance. As most of the state is contained in segments, checkpointing mainly consists of collecting the kernel state in the root segment. Thus, migration could be implemented easily by combining already existing kernel functionality:

- Create a team checkpoint at the source host,
- transfer all segments of the instance to the destination host,
- recover the team state from the checkpoint,
- rebind the communication connections of the team.

The primary type provides a standard implementation for type-independent migration [LHK93]. The migration protocol is responsible for the communication between the agents and manager as shown in figure 5:

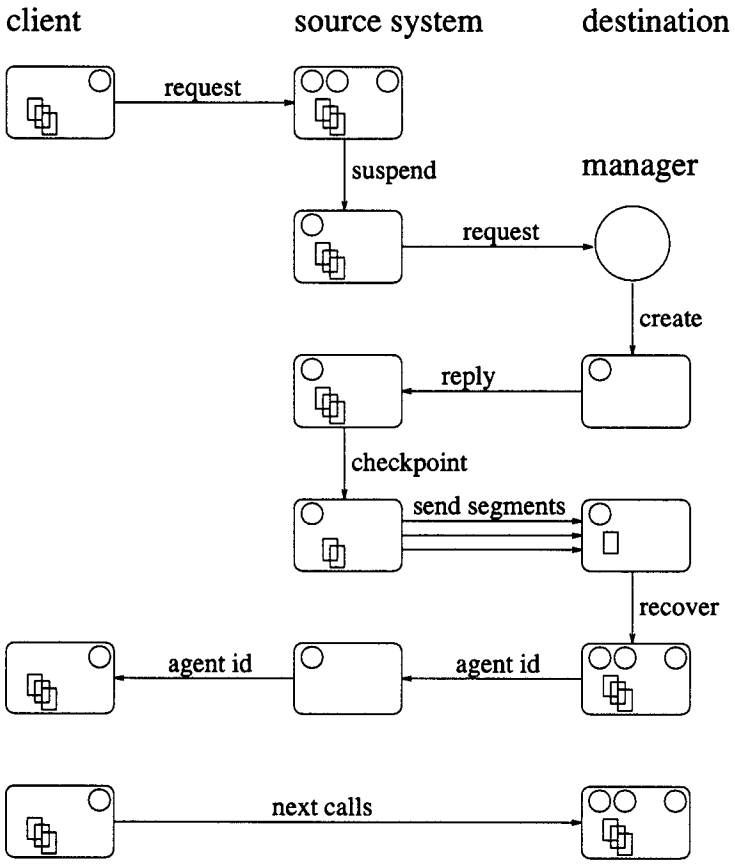

Figure 5: Migration Protocol

- A client sends a migration request to his agent (src-agent).
- Src-agent blocks the team communication (ingoing and outgoing), suspends all threads and sends a migration request to the manager at the destination host.
- The manager creates an agent (dest-agent) and forwards the request to dest-agent.
- Dest-agent announces the instance to the destination host by allocating a team descriptor and sends an acknowledgment to src-agent.
- Src-agent creates a checkpoint and sends all segments to dest-agent.
- Dest-agent recovers the team from the received segments and sends the new identification of the agent to src-agent.
- Src-agent enters a hint for the new location of the instance, releases all resources at the source host and sends the new identification of the agent to the client.

- Dest-agent resumes all threads and unlocks the team communication.

This standard implementation supports migration of multi-threaded instances of any type. Nevertheless, the unchangeable kernel implementation is not able to adapt migration to changing requirements.

4.2 Adding Adaptability

BirliX is based on a simple, but extensible object model: if the kernel implemented primary methods are not satisfactory for a user, he can substitute them with user-level implementations. Applied to migration, special requirements concerning state transfer and efficiency can be fulfilled by adding functionality to the standard implementation or by substituting it with a new one. For example, one can add encryption to the standard protocol, or even use a completely different protocol. In special cases, the amount of transferred data can be reduced by using type-specific knowledge. By changing the implementation, it is possible to adapt migration to changing execution environments and to use type-specific knowledge for migration. The type-independent interface for the client still remains unchanged.

4.2.1 Meta Migration

The adaption of a primary method is done using meta instances. A meta instance is an instance containing the implementation of a primary type method. Using the method **AttachMeta**, a meta instance is attached to an instance. When the corresponding primary method is called, the primary type activates the meta implementation instead of the kernel defined one. Since attaching a meta implementation is a security sensitive action, **AttachMeta** itself is a primary type method protected by the access control list. Applied to migration, different migration implementations are stored in different meta instances. Depending on the application-specific requirements, higher-level migration policies attach suitable meta instances to individual instances.

The availability of meta instances is a problem. To get maximal availability, an instance and its meta instances must reside at the same host. On the other hand, it should be possible to use a meta implementation by different instances at different hosts. In our solution, each instance has an additional segment containing primary type methods. During execution of **AttachMeta**, the implementation is transferred from the meta instance to the primary segment. Thus, each instance possesses its own implementation which is migrated together with the other segments.

At the start of migration, the primary type maps the primary segment to a reserved range of the instance's address space and calls the meta method. This meta method handles the migration protocol at the source host. As the meta implementation must also work at the destination host, there must be some initialization at the destination host. Therefore, the migration request message specifies if the standard or the meta implementation is used. If the meta implementation is used, the manager at the destination host maps the first received segment containing the meta implementation to a newly created instance address space and calls the meta method. The meta methods at the source and destination hosts handle then the remaining migration protocol.

4.2.2 Type-specific Migration

Adapting migration to types is in apparent contradiction to the decision not to encumber a type programmer with migration. A BirliX type does not need to support migration. Rather, a type programmer can use his knowledge about the implementation to make migration faster. Restartable threads and currently unused storage resources do not need to be transferred. An empty stack can be migrated by transferring the information "stack is empty".

The BirliX concept is not powerful enough to reduce an empty stack to this kind of information. A BirliX instance is too heavy-weighted, it has a team descriptor and communication connections. Therefore, type-specific migration is aligned to resource compression. If a data type supports compression, the primary

type calls the compression method before migration at the source host and its inverse method after migration at the destination host.

4.2.3 Cooperation of Mechanisms

For migration there are potentially three different mechanisms: standard , meta and type-specific. Cooperation of the different mechanisms is determined by the primary type as shown in figure 6. On a migration

```
PROCEDURE MigrateSrc((*InOut*) AgentId:  AgentIdT;
                     (*InOut*) Location:  LocationT)
                     :ResultT;
BEGIN
   call type-specific migrate for source host;
   IF meta implementation exists THEN
      activate meta implementation
   ELSE
      call standard implementation
   END;
END MigrateSrc;

PROCEDURE MigrateDest(Message:  MessageT)
                      :ResultT;
BEGIN
   IF meta implementation exists THEN
      activate meta implementation
   ELSE
      call standard implementation
   END;
   call type-specific migrate for destination host;
END MigrateDest
```

Figure 6: Cooperation of Mechanisms

request, the agent at the source host executes the operation **MigrateSrc** and the agent at the destination host executes **MigrateDest**. State transfer is bracketed by the execution of the type-specific calls. If there is an attached meta object, the meta implementation is used in preference to the standard implementation.

4.2.4 Interfaces

Adaptable migration is done by the cooperation of a client, the primary type and the meta implementation. The communication between these parties is specified by three migration interfaces.

Client - Primary Type

The primary type of an instance provides its clients with the interface shown in figure 7. After establishing a communication connection via **InstanceOpen**, a migration policy can use the operations **AttachMeta**, **InstanceMigrate** and **InstanceInfo**. At the system interface of BirliX, these operations are provided via procedure calls. Using the call parameters the type system generates a corresponding message and sends it to the agent. The agent calls the requested operation, puts the return parameters into the message and sends the message back to the client. If the operation could not be executed correctly, the message contains an error code.

```
PROCEDURE AttachMeta((*In*) AgentId:  AgentIdT;
                     (*In*) OpCode:  OpCodeT;
                     (*In*) MetaInstance:  UniqueIdT;
                     (*In*) MetaLocation:  LocationT)
                     :  ResultT;
(* AttachMeta attaches the meta implementation for the method OpCode to the      *)
(* instance identified by AgentId. The implementation is stored within the instance  *)
(* MetaInstance at host MetaLocation.                                            *)

PROCEDURE InstanceMigrate((*InOut*) VAR AgentId:  AgentIdT;
                          (*InOut*) VAR Location:  LocationT)
                          :  ResultT;
(* InstanceMigrate migrates the instance identified by AgentId to the host specified  *)
(* by Location. After execution AgentId contains the new identification of the agent  *)
(* and Location contains the location of the instance.                           *)

PROCEDURE InstanceInfo((*In*) AgentId:  AgentIdT;
                       (*OUT*) VAR Info:  InfoT)
                       :  ResultT;
(* InstanceInfo returns in Info information about the instance identified by AgentId. *)
```

Figure 7: Operations provided by the Primary Type

Primary Type - Meta Implementation

Figure 8 shows the operations which a meta instance must provide for the primary type. **MigrateSrc** and **MigrateDest** are implemented within a meta instance for migration and **CollectInfo** and **GetInfo** within a meta instance for information collection. The programmer of a meta instance must respect these interfaces to get the correct call parameters. When calling the operation, control is transferred from the kernel coroutine (primary type) to the user coroutine, which executes the meta implementation. After execution control is changed back.

```
PROCEDURE MigrateSrc((*Out*) VAR AgentId:  AgentIdT;
                     (*InOut*) VAR Location:  LocationT)
                     :  ResultT;
(* MigrateSrc executes the migration protocol at the source host. The destination  *)
(* system is specified by Location. After execution AgentId contains the new       *)
(* identification of the agent and Location contains the location of the instance.  *)

PROCEDURE MigrateDest((*In*) Message:  MessageT)
                      :  ResultT;
(* MigrateDest executes the migration protocol at the destination host. Message    *)
(* contains the message received by the manager.                                   *)

PROCEDURE CollectInfo((*In*) Message:  MessageT);
(* CollectInfo collects information using the message Message.                     *)

PROCEDURE GetInfo((*Out*) VAR Info:  InfoT);
                  (*In*) Initialize:  BOOLEAN);
(* GetInfo returns in Info the information collected by the meta implementation.  *)
(* Initialize specifies, if collecting should be initialized.                      *)
```

Figure 8: Operations provided by the Meta Implementation

Meta Implementation - Primary Type

To migrate an instance, the meta implementation needs access to the primary type. Since many different meta implementations may exist, it is difficult to find an interface suitable for all. A low-level interface preserving the consistency of the primary type and type system seems to be most flexible. Figure 9 shows the current interface. On calling the operations, control is transferred from the user coroutine to the kernel

```
PROCEDURE SuspendExternalRequests();
(* SuspendExternalRequests suspends establishing and deleting of communication    *)
(* connections to the instance.                                                    *)

PROCEDURE ResumeExternalRequests();
(* ResumeCommunication rescinds the lock for establishing and deleting of         *)
(* communication connections.                                                      *)

PROCEDURE SuspendTeam();
(* SuspendTeam suspends all threads of the team (except the calling one)          *)

PROCEDURE ResumeTeam();
(* ResumeTeam activates the suspended threads.                                    *)

PROCEDURE CollectKernelState();
(* CollectKernelState collects the team state and puts it to the root segment.    *)

PROCEDURE DistributeKernelState();
(* DistributeKernelState distributes the team state stored in the root segment    *)
(* to the system tables.                                                           *)

PROCEDURE ReleaseResources();
(* ReleaseResources releases not yet needed resources of the instance.            *)

PROCEDURE SegmentDup((*In*) OrigUniqueId: UniqueIdT;
                     (*Out*) VAR DupUniqueId: UniqueIdT);;
(* SegmentDup creates a duplicate from a segment. OrigUniqueId is the name of the *)
(* original segment and DupUniqueId is the name of the duplicate.                 *)

PROCEDURE SegmentDelete((*In*) UniqueId: UniqueIdT);
(* SegmentDelete deletes the segment UniqueId.                                    *)
```

Figure 9: Operations provided by the Primary Type

coroutine, which performs the desired action. After execution control is returned to the user coroutine.

4.3 Reliable Migration via Meta Instance

After integrating adaptable migration into the BirliX system, a couple of different migration mechanisms were implemented [Lux94]. This paper shows the implementation of *reliable migration* developed within the Charlotte operating system. Charlotte supports reliable and efficient process migration: "*Migration may fail in case of machine or communication failures, but it should do so completely. That is, the effect should be as if the process were never migrated at all, or, at worst, as if the process had terminated due to machine failure* [AF89]." To rescue a migrating process from failure, the Charlotte implementation makes three arrangements:

- Responsibility for the migrating process changes as late as possible to survive failures of the destination host.
- The process is detached completely from the source host to survive failures of the source host.

- The migrating process is protected from failures of other hosts by automatically destroying links to crashed hosts.

Charlotte does not rescue migrating processes under all failures, since this would need complex recovery protocols with large overhead.

Within BirliX, the arrangements are transformed in the following way:

- The responsibility for a migrating instance is bound to the root segment. Sending the root segment within the last message changes responsibility as late as possible.
- By sending all segments to the destination host [3] and releasing the kernel resources, an instance is detached completely from the source host.
- The crash of a host participating in migration is recognized by the BirliX message system.

```
PROCEDURE MigrateSrc((*Out*) VAR AgentId:  AgentIdT;
                     (*InOut*) VAR Location:  LocationT)
                     :  ResultT;
BEGIN
   suspend external requests and suspend the team;
   IF NOT send migration request THEN
      resume team and resume external requests;
      RETURN Error
   END
   duplicate all segments of the instance;
   collect kernel state;
   IF NOT send original segments THEN
      rename duplicates of the sent segments to the original ones;
      delete the duplicates of the segments not sent;
      resume team and resume external requests;
      RETURN Error
   END
   register hint to new location;
   delete duplicated segments;
   release resources;
END MigrateSrc;

PROCEDURE MigrateDest(Message:  MessageT) : ResultT;
BEGIN
   announce the instance;
   IF NOT receive all segments THEN
      delete all received segments;
      RETURN Error
   END
   distribute kernel state;
   resume team and resume external requests;
END MigrateDest
```

Figure 10: Reliable Migration

Figure 10 shows the implementation of reliable migration by a meta instance. To be able to rescue an instance from destination host crashes, a duplicate of each segment is created before sending it. If migration fails, at the source host the duplicated segments are renamed to the original ones and at the destination host the received segments are deleted. The links to crashed hosts are handled by the rebinding mechanism using the hints. Our current meta implementation (figure 10) does not have a complex recovery protocol. However, it is not very difficult to build a meta instance with a recovery protocol. It is then possible to choose between both meta implementations depending on the overhead tolerance.

[3] The message system deletes a segment at the source host as soon as an acknowledgment for the complete arrival is received.

4.4 Performance

Measurements were made to compare adaptable migration to other migration systems and to find points for optimization. Comparing migration is difficult because of the different hardware used. Our measurements were done on a network of Sun-3/60 machines connected by a 10 Mb Ethernet. An empty RPC to another host needs 3.55 msec. Figure 11 shows the migration times for the standard implementation and a meta implementation by varying the number of agents and the size of state. The figure shows a fixed part for the

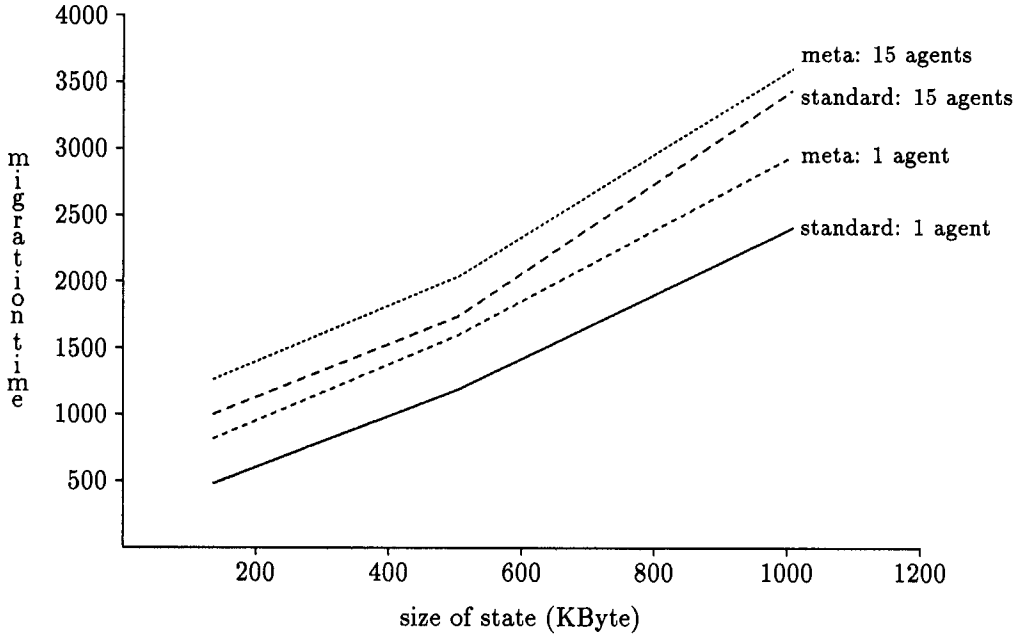

Figure 11: Migration Times in msec

meta implementation and for each agent and a linear effect for the size of the state.

Detailed measurements concerning the following topics are presented in [Lux94]:

- Migration times for different types in the Unix emulation.
- Execution times for the partial operations during migration.
- The effect of migration in a distributed application.
- A comparison with other migration systems.

Altogether, measurements have shown that adaptable object migration is not worse than special solution migration and that there are still a couple of points for optimization, such as improving RPC performance and reducing the size of a team's state.

5 Related Works

Nuttall [Nut94] gives a nearly complete overview about existing migration mechanisms. As mentioned in the introduction, process migration plays an important role and most of the systems are not able to adapt migration to changing requirements. Task migration [Mil93] supports multiple migration mechanisms via different migration servers. Adaptability can be achieved by selecting a suitable migration server. Although both task migration and adaptable object migration fulfil our requirements for migration, their handling of adaptability is quite different.

6 Conclusion

There are a lot of distributed applications which can profit from migration. Unfortunately, the existing migration systems are insufficient, as they are designed as special solutions. A universal migration system requires migration of objects of different types and must be adaptable to changing requirements. Adaptable object migration is a concept which meets these requirements. In this concept, each object has its own migration mechanism contained in an attached meta object. Changing meta objects renders object-specific adaptation.

In order to show its practicability, adaptable object migration was incorporated into the BirliX operating system. BirliX is an abstract data type management system providing mechanisms for the definition and instantiation of BirliX types, the identification of instances and communication between instances.

To integrate adaptable migration, the type system was extended. Migration became one of the kernel provided primary type methods inherited by all instances. The type system provides a standard implementation which migrates an instance by transferring a checkpoint and recovering the instance at the destination host. Adaptability is added by using meta instances containing the implementation of user level mechanisms. Attaching a meta instance enables instance specific migration depending on the current execution environment. Thus, adaptable migration in BirliX has the properties:

- The primary type supports the migration of instances of different types.
- Different migration mechanisms are available by different meta instances.
- By attaching a meta instance, migration mechanisms can be exchanged dynamically.
- Semantic knowledge can be obtained by calling the BirliX type.
- Mechanisms for information collection can be integrated dynamically.

The usability of the approach has been shown by implementing some well-known migration mechanisms. This paper has described the implementation of reliable migration. The performance analysis has shown, that the performance of adaptable migration is comparable to special solution mechanisms.

References

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *Transactions On Software Engineering*, 11(1), January 1985.

[AF89] Y. Artsy and R. Finkel. Designing a Process Migration Facility The Charlotte Experience. *IEEE Computer*, pages 47–56, September 1989.

[BAI93] B. R. Badrinath, A. Achary, and T. Imielinski. Impact of Mobility on Distributed Computations. *ACM Operating System Reviews*, 27(2):15–20, April 1993.

[BS85] A. Barak and A. Shiloh. A Distributed Load Balancing Policy for a Multicomputer. *Software - Practice and Experience*, 15:901–913, September 1985.

[Fen93] I. Fenske. *Virtuelle verteilte Objekte: eine Gruppenarchitektur für verteilte Systeme.* PhD thesis, Technische Hochschule Darmstadt, 1993.

[KH90] O.C. Kowalski and H. Härtig. Protection in the BirliX Operating System. In *Proceedings 10th International IEEE Conference on Distributed Computing Systems*, Paris, May 1990.

[KHKL90] Winfried E. Kühnhauser, H. Härtig, O.C. Kowalski, and W. Lux. Mechanisms for Persistence And Security in BirliX. In John Rosenberg and J.Leslie Keedy, editors, *Proceedings of the International Workshop on Security And Persistence*, Bremen, May 1990. Springer.

[LHK93] W. Lux, H. Härtig, and W. Kühnhauser. Migrating Multi-Threaded, Shared Objects. *Proc. IEEE HICSS 26 II*, pages 642–649, Jan 1993.

[Lux94] W. Lux. *Adaptierbare Objektmigration und eine Realisierung im Betriebssystem BirliX.* PhD thesis, submitted at University Hildesheim, 1994.

[Mil93] D. Milojicic. *Task Migration on Top of Mach.* PhD thesis, Universität Kaiserslautern, 1993.

[Nut94] M. Nuttall. A Brief Survey of Systems providing Process or Object Migration Facilities. *ACM Operating System Reviews*, 28(4):64–80, October 1994.

[R+88] M. Rozier et al. CHORUS Distributed Operating System. *Computing Systems, Vol. 1, No. 4*, 1988.

[RS84] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *Proceedings of the 4. International Conference on Distributed Computing Systems*, pages 96–107, May 1984.

[Sch90] A. Schill. *Migrationssteuerung und Konfigurationsverwaltung für verteilte objektorientierte Anwendungen.* Informatik Fachberichte 241. Springer Verlag, 1990.

[SHKL94] S. Sonntag, H. Härtig, W. Kühnhauser, and W. Lux. Adaptability Using Reflection. In *Proc. IEEE HICSS 27*, Jan 1994.

[TH92] M. Theimer and B. Hayes. Heterogeneous Migration by Recompilation. *Xerox Corporation Technical Report CSL-92-3*, March 1992.

[TR86] A. Tevanian and R.F. Rashid. MACH - A Basis for Future Unix Development. In *EUUG Conference Proceedings*, Manchester, 1986.

[TY92] M. Tokoro and Y. Yokote. The New Structure of an Operating System: The Apertos Approach. *5. ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, 1992.

[Zay87] E. Zayas. Attacking the Process Migration Bottleneck. *Proc of the ACM-SIGOPS 11th Symp on OS Principles*, pages 13–24, November 1987.