Rule-Based Transactional Object Migration over a Reflective Middleware

Damián Arregui, François Pacull, and Jutta Willamowski

Xerox Research Centre Europe
6, chemin de Maupertuis, 38240 Meylan, France
{Damian.Arregui, Francois.Pacull, Jutta.Willamowski}@xrce.xerox.com

Abstract. Object migration is an often overlooked topic in distributed object-oriented platforms. Most common solutions provide data serialization and code mobility across several hosts. But existing mechanisms fall short in ensuring consistency when migrating objects, or agents, involved in coordinated interactions with each other, possibly governed by a multi-phase protocol. We propose an object migration scheme addressing this issue, implemented on top of the Coordination Language Facility (CLF). It exploits the particular combination of features in CLF: the resource-based programming paradigm and the communication protocol integrating a negotiation and a transaction phase. We illustrate through examples how our migration mechanism goes beyond classical solutions. It can be fine-tuned to consider different requirements and settings, and thus be adapted to a variety of situations.

1 Introduction

Distributed systems use migration to perform load balancing, reduce network traffic and support mobile users. Both the operating systems and the mobile agents communities have extensively discussed the issues around process and agent migration. Current proposed solutions do not however cover all the requirements of today's enterprise distributed applications in domains such as electronic commerce, workflow, and process control. Indeed, such applications are often characterized by complex and dynamic relationships between distributed components. Suspending and later resuming this type of system (totally or partially) while preserving consistency becomes a real challenge. Nevertheless, maintenance operations on the underlying hardware and software infrastructures often require such operations to be performed. In order to ensure the continuous availability of the affected applications some of their components have to be transparently migrated from one node of the network to another.

In this article, we describe how the Coordination Language Facility (CLF) provides advanced support for object migration. It includes the externalization of migration control, and the reflexive use of the CLF middleware capabilities in order to offer transactional and negotiated migration. This enables flexibility and consistency.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 179–196, 2001.

CLF is a middleware platform aimed at coordinating distributed active software components over a Wide Area Network (typically the Internet). Mekano is a set of reusable coarse grain components and component development tools, compliant with the CLF middleware, that have been used in the implementation of various distributed applications deployed across multiple intranets. Complementary information can be found in [2].

Section 2 presents the basic functionalities of CLF used for object migration, section 3 describes the migration mechanism itself, section 4 gives examples of use, section 5 discusses related work, and section 6 concludes the paper.

2 Fundamental Features

We have implemented our approach for object migration on top of the CLF/Mekano platform. The migration facilities we describe could be implemented with other distributed object-oriented platforms but this would impose developing significant additional code in order to mimic some of the features readily provided by CLF. This section presents these features along with the minimum set of information needed to make this paper self contained.

2.1 Object Model

The CLF approach relies on the resource-based programming paradigm [5] that we have extended in order to cope with the requirements of a distributed system. We model objects as *resource managers*, the interactions between them as *transactional resource manipulations* (e.g. *removal* and *insertion* of resources in its simplest form) and the *resources* as tuples of strings. Each string element of

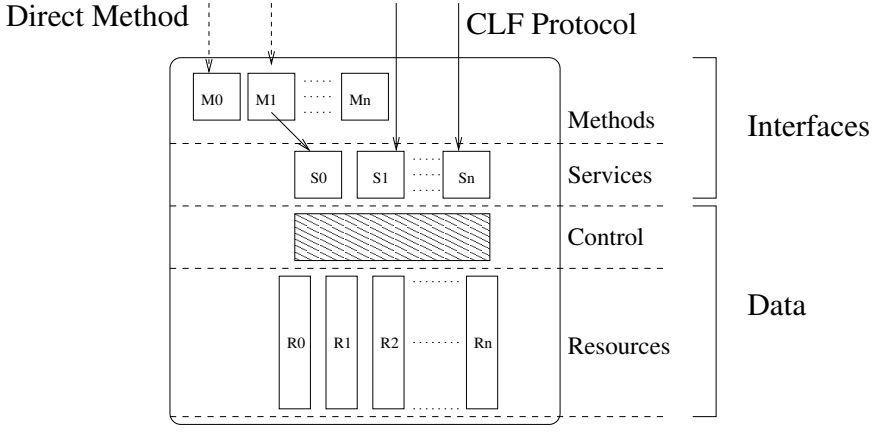

Fig. 1. CLF Object Model.

a resource can either contain text, string-encoded objects (e.g. XML-encoded) or marshalled objects (e.g. through Java [7] serialization).

The CLF object model goes beyond the classical dichotomy between *data* and *behavior*. Indeed, among the data itself we distinguish two separate kinds: the *control* data and the *resources* managed by the object (see figure 1). The control data covers information related to resource management while the resources represent the actual information contained in the object. For instance, the control data contains the structures (e.g. locks) to manage concurrent access to the object resources.

CLF enforces the traditional object encapsulation policy: resources are not accessible directly but only through an interface. But, unlike traditional objects, CLF objects offer two kinds of interfaces: *direct methods* and *services*. Direct methods correspond to traditional remote method invocations on a single CLF object while services allow the coordination of access to resources held by multiple CLF objects. Both are described in more detail in the two following sections.

2.2 Direct Methods

Direct methods typically provide user interface functionality for CLF objects, in particular for thin clients such as Web browsers. A direct method has the following abstract signature:

Perform: *input-method-parameters* -> *output-method-parameters*

Direct method invocations are similar to Remote Procedure Calls [14] available in conventional middleware, e.g. CORBA, and renamed "Remote Method Invocations" [8] in the Java world. Being HTTP-based, they can be invoked through a simple URL call, providing various encoding options for the input and output parameters. Initially they were designed to quickly add simple user interfaces to a CLF application using direct methods whose parameters and result encodings are directly supported by standard Web browsers (so called "form-data" or "url-encoding" for input parameters and HTML for the result). The now effective new generation of XML browsers and the standardization of various flavors of XML-RPC[15] makes this approach even more valuable.

Direct methods enable a *synchronous single-phase* interaction with a *single* CLF object. Services on the contrary allow to *transactionally access and consistently modify* the resources held by a *set* of CLF objects.

2.3 Services and Interaction Protocol

A service of a CLF object corresponds to a partial view of the resources the object manipulates. A three phase protocol deployed on top of the object services specifies how to access these services and how to manipulate the underlying resources. It allows in particular to coordinate the interaction within a set of CLF objects. This protocol consists of three phases: *negotiation*, *performance*, and *notification*. Each phase is in turn materialized through a set of corresponding interaction verbs that are invoked as described below.

First phase: negotiation. The negotiation phase asks a CLF service about offers for actions on resources matching a given filter. On such a request, formulated through the **Inquire** verb, the service returns a potentially infinite stream of offers. Therefore the service returns an `InquiryId` allowing the requester to access the corresponding offers one by one through the **Next** verb. For each offer, an `actionId` allows the identification, and to reference it later in the protocol. When no currently available resource matches the inquiry the **Next** operation remains pending until a new offer becomes available. The latter might happen either after internal changes within the object or through the insertion of new resources (see the notification phase). The service can also explicitly close the stream of offers by raising the NO-MORE-VALUE exception. This happens if the service can guarantee that no new offer matching the inquiry will ever become available. The requester can also **Kill** the inquiry if he is no longer interested in corresponding offers. Finally, the requester may, at any time, check the validity of an offer through the **Check** verb. The verbs involved in the negotiation phase have thus the following abstract signature:

Inquire: *input-service-parameters* -> *inquireId*
Next: *inquireId* -> <*output-service-parameters*, *actionId*> | NO-MORE-VALUE
Kill: *inquireId* -> VOID
Check: *actionId* -> YES | NO

Second phase: performance. The performance phase unrolls a classical two-phase commit protocol ensuring the atomic execution of a set of actions found during the negotiation phase. To achieve atomicity, the requester first attempts to **Reserve** the resources corresponding to these `actionId`s. If successful, it enacts all the actions through the **Commit** verb. Otherwise, if any reservation fails, it **Cancel**s all previously executed successful reservations. The verbs of the performance phase have the following abstract signature:

Reserve: *transactionId*, *actionId* -> ACCEPT | SOFT-REJECT | HARD-REJECT
Commit: *actionId* -> VOID
Cancel: *actionId* -> VOID

Third phase: notification. The notification phase allows for asynchronous creation of new resources. The verb **Insert** notifies their creation to the corresponding services:

Insert: *service-parameters* -> VOID

2.4 Scripting Language

The CLF scripting language exploits the CLF object model and services through the above described protocol. It views coordination as a complex block of interrelated manipulations of resources held by a set of objects (called the *participants* of the coordination).

CLF scripts describe, through rules, the expected global behavior of such blocks in terms of resulting resource manipulations, but abstracts away from the detailed sequencing of invocations of the CLF interaction verbs required to achieve such a behavior. This abstraction feature considerably simplifies the design and verification of coordination scripts. It makes them highly platform independent and hence, portable.

In a CLF application, dedicated CLF objects called *coordinators* enact the coordination scripts. As any CLF object, coordinators manage resources, accessible through CLF services: these resources are CLF coordination scripts and the rules which compose them. When a script is inserted in a coordinator, it is immediately enacted. Being CLF objects, coordinators can participate (i.e. occur as participants) in higher level coordinations, thus offering a reflexive model of coordination. Moreover, it is possible to create and insert scripts on the fly.

The CLF coordination scripting facility does not specify, per-se, any computing feature. If computation is needed in a coordination (arithmetic computation, string manipulation etc.), it must be handled by a participant. However, in the CLF distribution, a basic stateless computing facility is delivered with the coordinator prototype in order to provide simple computation, verification of assertions and timeout evaluation.

As shown later, we use the features and the power of this scripting language to implement our object migration mechanism.

2.5 Sample Script

The sample script detailed here is intended to show most of the CLF scripting language features used later in this paper. More applied script examples may be found in [3].

```
interfaces:
  ...
  YP(Obj): -> Obj is LOOKUP YellowPages.Objects
  apply(Obj,Serv,Y,Z): Obj,Serv,Y -> Z is DISPATCH

rules:
  S(X) @ P(X,Y) @ 'YP(Obj) @ apply(Obj, 'service',Y,Z) <>- R(X,Z)
```

The tokens `S`, `P`, `Q`, `R`, `YP`, `apply` refer to CLF services declared in the interface of some CLF objects (found and described in the name server, as shown later in section 2.7). For each token, the parameters appear between parentheses. The output parameters are underlined.

The logical name of the service can be statically defined (e.g. `YP` is linked to the service `Objects` of the `YellowPages` object) or dynamically according to the result of the instantiation of some parameters (e.g. the first two parameters of the token `apply` refer to the object and service name, the latter being known here only at run-time).

If such a rule is inserted in a coordinator, it is executed as follows:

1. Resources satisfying properties `S`, `P`, `Q`, `R`, `YP` are found:
   - The token `S(X)` finds some resource satisfying property `S`. The parameter `X` is instantiated accordingly using a value returned from the service that corresponds to `S`;
   - The second token finds a resource satisfying the property `P(X,Y)` for consistent values of `X`,`Y`.
   - The token `YP(Obj)` finds some resource satisfying property `YP`. The backquote (`) before the `YP` prevents the resource satisfying the `YP(Obj)` from being extracted in the transaction phase (if any).
   - The token `apply` is not statically linked to a logical name but will use the first two parameters as the object and the service name that will be used for the lookup. It is what we call the *dispatch* mechanism. The first one, `Obj`, is returned by the previous token `YP` while the second is set to the constant *`'service'`*. It finds consistent values of `Y`,`Z`.
2. For each tuple of consistent values `X`, `Y`, `Z`, `Obj`, the rule is triggered and transactionally extracts the resources satisfying `S`, `P` and `apply`.
3. A resource satisfying `R(X,Z)` is finally inserted.

2.6 Runtime

The CLF runtime provides the basic system facilities required for hosting and managing CLF objects on a site (See figure 2). Among these facilities we use the following for our migration mechanism:

- the managers (Search Manager, Concurrency Manager, Insert Manager) supporting the CLF protocol interaction for the objects, i.e. the asynchronous aspects of Inquiry/Next (SM), concurrency control and deadlock avoidance for Reserve/Cancel/Confirm (CM), event detection for Insert (IM), and garbage collection for Kill/Check (SM).
- the communication blocks encapsulating the communication protocol on top of which the CLF protocol verbs and the direct methods are implemented. These blocks allow for decoupling of the requests from the communication scheme (message passing, RPC) and the communication protocol (HTTP, SSL, XML-RPC, RMI).
- the system facilities allowing remote interaction with a CLF object. These facilities can allow for instance to start and stop an object within an application. They can also trigger the instantiation of a CLF object from a description or on the contrary to reify a CLF object into a description. The latter two cases are normally used for deploying and controlling an application. They are of particular interest in the context of object migration as described in section 3.

The CLF runtime facilities are, in general, directly invoked by various types of clients for deploying, monitoring and controlling CLF applications. However, we have encapsulated some of them in the services of a dedicated CLF object (see section 3.2) to be able to access and coordinate them through rule scripts.

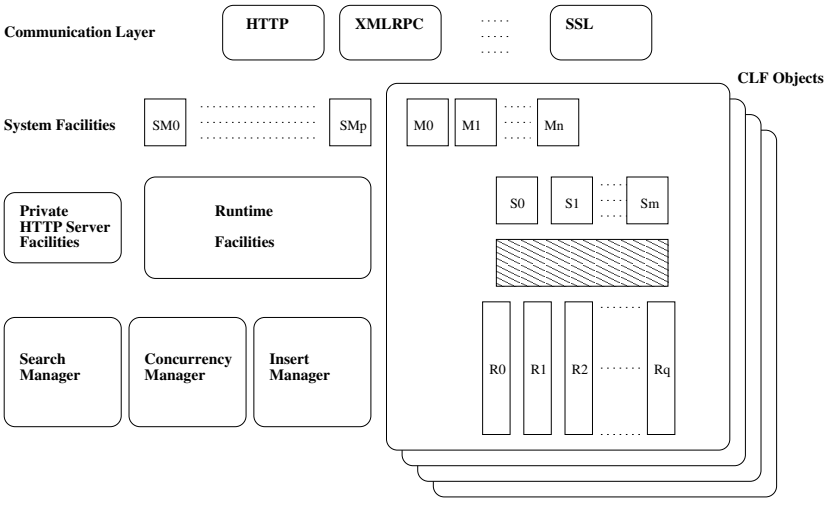

Fig. 2. CLF Runtime.

2.7 Name Server

The *name server* is a CLF object whose resources are *access patterns* binding *logical names* of objects to their *physical location*. CLF objects register themselves at the name server. The name server allows the coordinators to lookup the physical location of CLF objects from their logical names. This maps the logical description of the coordination contained in the rule scripts to the actual location of the CLF object service of an application distributed across several hosts.

Ensuring that the information contained in the name server is always accurate is of course essential to implement object migration. The possibility to access the resources manipulated by the name server through a CLF rule provides a simple way to modify them in a consistent manner during ongoing migration.

3 Object Migration Scheme

This section describes how the features presented in the previous section interoperate to provide the global CLF migration mechanism. This mechanism essentially relies on the reflexivity of the CLF platform. The basic idea is to first transform an active CLF component into a static resource, then transport it over the distributed system, just like any other resource, then instantiate it on the destination site.

This section first describes how a CLF object enters a freezable state where it can be safely and consistently transformed into a static resource. It then presents how we encapsulated the CLF runtime into a CLF object, the *object manager*,

providing services relevant to the migration scheme. We show in detail how these services are customized, i.e. how the verbs of the CLF protocol directly trigger the different methods readily provided by the runtime in order to realize the various migration steps. Finally, we illustrate through a very simple scenario how this allows us to manage object migration.

3.1 Reification and Freezable State

Migration requires the target object to enter a *freezable state*, where the data representing the current execution state of the object can be captured. Once this state is reached, we can easily transform the dynamic object into a static description, the *reified object*. From the reified object we can later re-instantiate the corresponding CLF object, possibly at a different location.

Freezable State Definition. Thanks to the clear distinction on one hand between control data (state of the inquiries and the transactions) and on the other hand the resources managed, we can easily define the freezable state of a CLF object as a state where the control data set is empty. In fact, as the resources are tuples of strings, they can be directly stored and recovered. So, for the freezable state we only have to consider the control data set: it contains information about the state of multi-phase interactions involving the object. In CLF, an empty control data set corresponds with the situation where no critical operation modifying the object resources is pending. In this case, the reified object (the set of resources together with the object description) entirely describes the object and can thus be used for object migration. This reified object is exactly the one used when starting objects at deployment time. Thus restarting objects after migration does not necessitate any specific additional code.

Freezing an object. As explained, to reach a freezable state, we have to flush the control data set. We now define in the following a process to guarantee that no new control data appear and that the control data set decreases and finally becomes empty. We therefore have to consider all interactions that modify the control data set of an object. The direct methods represent synchronous single-phase interactions which can only access the resources through the services. We can thus ignore them in this context. The CLF protocol however is a multi-phase interaction protocol that allows for the modification of the object resources. The control data set represents the state of this interaction. To reach a freezable state we therefore have to consider the different phases and verbs of the CLF protocol:

- The verbs belonging to the negotiation phase are *idempotent*: they do not modify the resources. Thus, even if the negotiation phase implements a multi-phase protocol: an Inquiry followed by a potentially infinite number of Nexts. It can be aborted without harm. Indeed, the coordinator will detect that the negotiation phase was aborted, in the same manner as if the object became

unavailable. In this case, the coordinator will restart the Inquiry when the object becomes available again, possibly at another location.
- The verbs belonging to the performance phase are *non-idempotent*: they can reserve and remove resources. They implement a two-phase commit protocol. The execution state of the protocol, in particular the reservation of a resource involved in a transaction, is stored in the object control data ; it might be complex to store and recover. Furthermore, by policy, CLF transactions are guaranteed to be short-lived. A reserved resource is either rapidly consumed through a Commit or released through a Cancel. Thus, the appropriate solution to reach a freezable state is to temporarily reject any new transaction and to wait until all ongoing transactions are completed.
- The notification phase only consists of the Insert verb and implements a synchronous one-phase interaction, that does not impact the control data set. Furthermore, it can be delayed without harm, as the coordinator will repeat pending insertions until they are acknowledged by the object. An object can therefore be frozen independently from pending notifications.

To sum up, the steps to freeze an object are:

1. break the connection for Inquiry/Next verbs simulating somehow a stopped object ;
2. return a *soft-reject* for any `Reserve`. A soft-reject tells the coordinator to retry the reservation later again if still required[1] ;
3. accept Commit/Cancel as usual ;
4. accept Insert invocations, but only until the control data set is empty. After that, apply the same treatment as for Inquiry/Next. Just like for the Inquiry/Next, the coordinator will retry the Insert invocations once the object becomes available again.
5. Once the object is freezable (i.e. the control data set is empty), we close the connection for any subsequent invocation, and freeze its state.

The CLF interaction model (services + direct methods) ensures that a freezable state will eventually be reached for every object.

3.2 Encapsulating the CLF Runtime

As described in the previous section the CLF runtime provides methods to start a CLF object and then to control it. Among these methods, the following ones are relevant for the migration process:

- `EnableObjectIsolation(objectname)`
  Triggers the processing of incoming invocations as described in the previous section for reaching a freezable state. At the same time the object is unregistered from the name server, preventing other useless subsequent lookups.

[1] In fact, the transaction is retried only if none of the other participants has returned a hard-reject implying the abortion of the transaction as a whole.

- `DisableObjectIsolation(objectname)`
  Resumes the normal management of incoming invocations and registers the object again on the name server.
- `isObjectFreezable(objectname) -> status`
  Returns a status, either *'ok'*, or *'notOk'*, meaning if the control data set is already empty or not. Once *'ok'*, this is a stable state as long as the isolation is not disabled.
- `ReifyObject(objectname) -> objectDescription,resourceSet`
  Returns the static description of the (frozen) object: object description (object type name and configuration parameters) and resources currently managed.
- `InstantiateObject(objectname,objectDescription)`
  Instantiates a new object according to the object description and registers it under the name `objectName` on the name server.

To benefit from the transactional semantics for object migration, and to fully take advantage of the reflexivity provided by CLF, we encapsulated these methods through a CLF object, the *object manager*. This object offers two CLF services: *reify* and *instantiate*. As we show in the following, these services are implemented in such a way that they call the above described methods, correctly mapping the migration process to the phases and verbs of the CLF protocol. Object migration in CLF is implemented as a coordination of the appropriate services of two object managers.

3.3 Service Reify

Reifying an object consists in transforming an *active object* into a *static piece of data*, typically a resource, that will later allow the re-activation of the corresponding object, possibly at another location. Reification is handled through the CLF object manager containing the object to reify.

The `Reify` service of a CLF object manager manages resources that are tuples of arity three. The first field corresponds with the logical name of the object to reify, the second one with object description (type name and configuration parameters), and the last one with its current set of resources. The values of the second and third fields are XML documents containing the description of the object: the set of modules corresponding to the objects static code, and the set of resources held by the object.

With respect to the CLF protocol described in section 2.3 the `Reify` service implements the following behavior:

- `Inquire`: On this verb the freezing process of the requested object is triggered. The < *objectname* > parameter, provided as input, denotes the object to reify. The second and third parameters will contain as output the reified form of the corresponding object. On such an Inquiry a new thread is launched, triggering the following reification process:

1. invoke the method `EnableObjectIsolation(`< *objectname* >`)` ;
2. loop until the method `IsObjectFreezable(`< *objectname* >`)` returns $'ok'$;
3. invoke the method `ReifyObject(`< *objectname* >`)` to obtain the reified form < *objectDescription* > and < *resourceSet* > of the object ;
4. insert through an internal API the resource `(`< *objectname* >`,` < *objectDescription* >`,`< *resourceSet* >`)` into the service enabling it to respond to a Next.

- `Next`: This verb blocks until the resource corresponding to the reified object becomes available as described above. Then, it returns an `actionId` to access and reserve this resource.
- `Kill`: This verb cancels the object freezing process associated with the `InquiryId` input parameter. The method `disableObjectIsolation` allows to stop this process and to restore the normal processing of external interactions.
- `Check`, `Reserve`: These verbs have their usual meaning with respect to the CLF protocol (see section 2.3).
- `Cancel`: This verb has the same effect as Kill, i.e. interrupting the freezing of the object.
- `Commit`: This verb destroys the frozen object definitively (within this object manager): it removes all remaining data allocated to the object.
- `Insert`: Invoking this verb raises an exception ; it has no sense with respect to the Reify service.

3.4 Service Instantiate

Instantiating an object consists of creating an active object from its reified form. This is handled through the Instantiate service of the target object manager.

The `Instantiate` service has three parameters identical to the Reify service but they are all input parameters.

The particularity of this service is that it can be used in two ways. Used on the right hand side of a rule, in the notification phase (verb Insert), it simply instantiates the object corresponding to the inserted resource. Used on the left hand side of a rule, in the negotiation and performance phases, it verifies possible preconditions for object reification (verb Reserve). As a consequence the whole object migration process can be cancelled and renegotiated as we will see in section 3.5.

With respect to the CLF protocol described in section 2.3 the Instantiate service implements the following behavior:

- `Inquire`: The given object name and reified description are associated with an `inquireId`. An associated resource corresponding to the given object name and the reified description is inserted into the service through an internal API.
- `Next`, `Cancel`: These verbs have their usual meaning in the CLF protocol (see section 2.3).

- `Kill`: This verb removes the corresponding resource created through the implementation of the Inquire verb from the service.
- `Reserve:` This verb checks in advance if the object can actually be instantiated from its reified form. This verification can have several facets. For instance, we can verify that all the required libraries or system resources are available at the target site. If any of these conditions are not verified then the associated object migration can be aborted and possibly renegotiated.
- `Commit`: This verb instantiates the object from its reified form. It removes the corresponding resource created through the implementation of the Inquire verb from the service. Once instantiated, the object registers itself under the given name on the name server.
- `Insert`: This verbs triggers the instantiation of an object from the given reified object description, without verifying any preconditions. Once instantiated, the object registers itself under the given name on the name server.

3.5 Migration Scripts

Object migration consists of three steps: object reification, transportation, and instantiation. As discussed in the previous section the CLF object managers provide services for object reification and instantiation. Coordination scripts initiate and handle object migration through these services. They achieve object transportation simply by passing the reified object description from the source object manager Reify service to the target object manager Instantiate service.

Migration scripts can be generated on the fly, either on user request or responding to a particular monitored run-time condition of the distributed system. Once inserted into the coordinator they are automatically enacted.

In the following, we show two elementary migration rules and will provide more practical examples in the next section. The first rule describes the unconditional migration of an object `obj1` from a source object manager `objMgr1` to a destination object manager `objMgr2`:

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) <>-
objMgr2.instantiate(''obj1'',objectDescription,resourceSet)
```

In this case the instantiation of `obj1` on `objMgr2` is taken for granted. Indeed, as the token appears on the right hand side of the rule, the instantiation is handled through the Insert verb of the Instantiate service. Thus no preconditions for instantiating the object on the target object manager are verified.

If preconditions have to be verified before migration, the `instantiate` token has to appear on the left hand side of the rule:

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) @
objMgr2.instantiate(''obj1'',objectDescription,resourceSet) <>-
```

The instantiation is now handled through the Inquire, Reserve, and Commit verbs of the Instantiate service, thus including the verification of basic preconditions for instantiating the object on the target object manager.

As we will see in the next section, several objects can be atomically migrated within a single transaction. Also, further constraints to be verified before commiting the migration can be integrated in the rule.

4 Examples of Use

As shown in the previous section, CLF scripts explicitly describe object reification, transportation, and instantiation and this, externally to the concerned objects. We now illustrate this further through different migration scripts.

4.1 Cloning Objects

Simple Cloning. The first example rule shows how to clone an object and to start the clone on another machine, within another object manager:

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) <>-
objMgr1.instantiate(''obj1'',objectDescription,resourceSet) @
objMgr2.instantiate(''obj2'',objectDescription,resourceSet)
```

Here the original object `obj1` is reified and re-instantiated on the original site `objMgr1` while the clone `obj2` is instantiated on a second site `objMgr2`.

Cloning and Load Balancing. Here, we go one step beyond and split the resources of the original object into two before cloning.

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) @
split(resourceSet,part1,part2) <>-
objMgr1.instantiate(''obj1'',objectdescription,part1) @
objMgr2.instantiate(''obj2'',objectDescription,part2)
```

The token `split` appearing on the left hand side of the rule is implemented via a simple stateless computation service directly enacted by the coordinator (see section 2.4). On the right hand side two `instantiate` tokens instantiate the two clones with separate sets of resources on different object managers.

4.2 Contextual Migration

Our scripting approach also enables more complex migration schemes, allowing to trigger and manage migration transactionally. Contextual migration is an example. Indeed, transactions allow to verify a global context or condition among distributed components: within a transaction, the involved components can agree on the global distributed condition, thus triggering migration accordingly. Obviously, the considered context can be linked to the objects themselves, the application, the user, the distributed system or any combination of them.

As an illustration, consider the common situation where the system administrator of a machine starts a shutdown process with a countdown. This typically

warns other human users of the machine that the machine will be stopped. In reaction, they can stop their activities, save their work, and possibly log in to another machine or just wait until the machine is up again. However, for an autonomous software component, this situation is critical. Without the external help of a human, the shutdown will kill the component. This is an issue for a lot of applications that need to run continuously, e.g. knowledge management, workflow, electronic-commerce, or on-line trading for example.

The following rule allows us to manage the necessary migration process:

```
systemEvent(''shutdown'',objectMgrSrc) @
objectname(objectMgrSrc,''Objectname'',obj) @
reify(objectMgrSrc,''Reify'',obj,objectDescription,resourceSet) @
possibleDestination(objectDescription,objectMgrDst) @
instantiate(objectMgrDst,''Instantiate'',objectDescription,resourceSet) @
generateMsg(obj,''shutdown'',objectMgrSrc,objectMgrDst,msgTitle,msgBody)
<>-
email(''clf@xrce.xerox.com'',''appMgr@xrce.xerox.com'',msgTitle,msgBody)
```

Captured system events of type *shutdown* trigger this rule. As soon as a resource corresponding with such an event becomes available along with the involved `objectMgrSrc` (first token), we fetch all objects it hosts (2nd token). Here the use of the dispatch mechanism that fixes the object and the service name linked to the token `objectname` only at run-time (the token `systemEvent` instantiates the object manager variable `objectMgrSrc` and the service name is a constant *"objectname"*). For each object hosted by the concerned object manager, we initiate reification (3rd token, also a dispatch) and then fetch a possible destination (4th token). For each possible destination the token `instantiate` (dispatch) checks that instantiation is effectively possible. Finally, the last token builds a notification e-mail message through a simple computation service.

As soon as a global solution for all tokens on the left hand side of the rule is found, the respective operation of the services is performed, effectively migrating the corresponding object. The `email` token encapsulating a simple e-mail client, on the right hand side of the rule notifies the application administrator that the object has migrated. Note that the reified object is a single resource, meaning that as soon as it has been consumed, no other instantiation of the rule can be applied. This ensures that each object migrates only once.

4.3 Transactional Migration of a Set of Objects

Transactional migration of a set of objects at the same time is another interesting case easily covered by our approach. Consider, for instance, the case of a powerful server, hosting several dozens of objects, that has to be stopped. Here, we would like all objects to migrate to other hosts of the system. In this context several issues arise. First, as all the objects need to migrate at the same time, we have to ensure that the migration process does not lead to overloaded hosts, e.g. that the first possible host in the list of available ones receives all the migrating objects. Thus, a solution relying on an unconditional migration is not suitable.

Secondly, for performance issues we have to keep certain groups of objects collocated. Their migration has to be managed via a single rule ensuring that they either all migrate to the same site or none of them migrates (see section 3.5). In such cases, as the scripts are very specific to a given problem, they have to be dynamically generated. Dedicated rules detecting shutdown events can trigger their generation on the fly and insert the resulting rules into a coordinator enacting them.

5 Related Work

The migration problem in itself is not new and some existing systems provide mechanisms enabling process or object migration. In this section we compare our approach to three classes of systems providing similar facilities: operating systems, object-oriented systems, and mobile agent platforms.

5.1 Operating Systems

In the operating systems [9] community, process migration is a well-known research topic: it is used to support load balancing, increase reliability and avoid communication overhead by exploiting locality. Operating systems implement migration at a lower level than the one addressed in this paper. But the issues to solve and the decisions on design are similar.

A first problem is to capture the execution state of a process to suspend its execution and to later resume it on a different node. This task is usually complex. In our approach, the CLF resource-based programming paradigm and the associated object model allow the definition of a freezable execution state, as described in section 3.1.

A second issue is the control of the migration process. Depending on the intended use of the migration capabilities, different operating systems adopt different solutions on who controls this mechanism: the kernel, a manager process, the migrating process itself, the system administrator or any other user. In some systems [4], the source and the destination nodes can negotiate the migration, on the basis of the available resources. In our approach, as the CLF rules control the migration process (see section 4) we allow any entity of the system to create and enact such rules via the coordinator. Furthermore, with the CLF protocol, negotiation naturally comes into the picture.

A third major topic is to ensure the availability of the process environment and resources (in the operating system sense) after migrating to a different machine. We do not address this issue. Providing a generic solution to migrate the legacy systems, databases or Web services we encapsulate with CLF objects is beyond the scope of this paper.

Finally, the biggest issue concerns Inter-Process Communication, i.e. how to cope with communication channels which are open when migration is requested, messages which arrive during migration and the new, different process location after migration. The various adopted solutions include buffering messages and

handling the process location through name servers. In our approach, we first empty the control data set, i.e. the communication state, of the migrating object (see section 3.1). During migration, the coordinator buffers further interaction requests. Afterwards it tries to perform them again, automatically retrieving the new, updated location of the migrated object from the name server.

5.2 Object-Oriented Systems

The object-oriented paradigm seems particularly well-suited to migration [1]: it provides data encapsulation. The encapsulated data in principle directly represents the object state. It also standardizes the address scheme through global namespaces. Both features directly support object migration. Still, coherently stopping and migrating an active object engaged in interaction with several other objects remains difficult. With respect to this problem, our model goes one step beyond, distinguishing between control data and resources as described in section 2.

Some existing systems provide language-level support for migration [13]. This approach requires the addition of new keywords to the language, and hence to build new compilers. Besides this, it often requires additional code such as marshalling/unmarshalling routines and to explicitly define which objects can migrate. Our migration mechanism does not impose such constraints: within CLF a component developer does not need to do anything special to benefit from object migration.

5.3 Mobile Agents

In the mobile agents community migration is obviously a central issue. In fact, the aim is to optimize locality through agent mobility: rather than communicating with other agents remotely, a mobile agent decides to travel close to the agent or service with whom it wants to communicate and then communicates locally. Mobile agents are in general considered to be self-contained and autonomous. A typical example is the commercial agent traveling on behalf of a client from one electronic market place to another, negotiating each time with the locally available providers. This approach of mobility seems rather restricted, in contrast with the consistent migration of objects involved in multiple complex interactions with others, as provided by our approach.

The OMG "Mobile Agent Facility" specification [11] defines mobility aspects for agent systems. Roughly an agent can travel from one location to another having its class and state serialized and sent to the destination location. There the information is deserialized and the agent resumes execution according to its state. This specification contains no particular considerations about consistent migration. Voyager [10] and Aglets [6] for instance implement a corresponding mechanism. However, Voyager only guarantees consistent handling of "synchronized" methods for mobile agents. The programmer has to ensure that no other methods are active when an agent receives a travel request. With Voyager and Aglets the programmer has furthermore to ensure that all entities referenced

by a possibly mobile agent are serializable. Thus, just like in object-oriented systems, in mobile agents platforms the programmer has to provide specific additional implementation aspects needed for mobility. Within our approach, we reuse intrinsic mechanisms of the platform and thus, we can say that migration comes almost for free.

6 Conclusion

In CLF, as opposed to most existing systems, migration control is explicit. CLF rules, external to the migrating object(s), initiate and control the migration process. These rules can be very simple or rather complex depending on the migration conditions to express. This allows for high flexibility and control of the migration process at different levels:

- *initiating the migration*: the migrating object itself, the user, the application, or the underlying system.
- *triggering the migration*: the verification of local or distributed application and system conditions, the capture of external events, or predefined milestones.
- *negotiated migration*: migration can be simply imposed, or negotiated, e.g. between the source and the destination object managers. Also the destination of the migration process can be negotiated with a site broker.
- *transactional migration*: migration can concern only a single object or a set of objects as a whole.

In CLF the migration logic of an application is specified independently of the application logic itself. This allows systems-oriented developers to take care of the migration logic whereas application-oriented developers may design the application logic.

Our migration scheme goes beyond classical ones, in the sense that we do not migrate code but the description of an object that is used to re-create it at the destination. Thus, we consider several migration axes:

- *space*: an object travels across space in the underlying distributed system. This is classical migration.
- *versions*: an object travels across versions in the sense that it can be reified in one version and instantiated later on in an updated version. Moreover, the migration script could modify the object description on the fly, during migration.
- *environment*: an object can travel from one implementation language to another, e.g. initially running in a Python [12] virtual machine and then with the same resources transferred to a Java virtual machine. This is of particular interest when some components have to be migrated to portable devices imposing a particular system/language combination.

In order to complete our migration facilities, we plan to build tools (and in particular visual tools) to better harness the power of the CLF scripting language

for controlling object migration. The goal is to enable both application designers and system administrators to use higher level concepts to configure the migration mechanism.

Acknowledgements. The authors would like to thank Eric Forestier for his contribution in implementing an earlier migration mechanism on top of the CLF infrastructure, and Irene Maxwell for her helpful comments on the final version of this paper.

References

1. P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski". Transparent object migration in COOL2. In Yolande Berbers and Peter Dickman, editors, *Position Papers of the ECOOP '92 Workshop W2*, pages 72–77, 1992.
2. J-M. Andreoli, D. Arregui, F. Pacull, M. Riviere, J-Y. Vion-Dury, and J. Willamowski. CLF/Mekano: a framework for building virtual-enterprise applications. In *Proc. of EDOC'99*, Mannheim, Germany, 1999.
3. J-M. Andreoli, D. Pagani, F. Pacull, and R. Pareschi. Multiparty negotiation for dynamic distributed object services. *Journal of Science of Computer Programming*, 31(2–3):179–203, 1998.
4. Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47–58, 1989.
5. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
6. IBM. Aglets. http://www.trl.ibm.com/aglets/.
7. Sun Microsystemns. Java. http://java.sun.com/.
8. Sun Microsystems. Java Remote Method Invocation specification. Technical report, Sun Microsystems, 1997.
9. M. Nuttall. Survey of systems providing process or object migration. Technical Report 94/10, Imperial College, London, UK, 1994.
10. ObjectSpace. Voyager. http://www.objectspace.com/products/voyager/.
11. OMG. Mobile Agent Facility specification. http://www.omg.org/cgi-bin/doc?formal/2000-01-02, January 2000.
12. Python. http://www.python.org/.
13. M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, pages 191–204, Nottingham (GB), July 1989. Cambridge University Society.
14. Sun Microsystems. RPC: Remote Procedure Call protocol specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.
15. XML-RPC. http://www.xmlrpc.org/.