Abstract
To fully support distributed processing in any distributed system, it must be possible to perform remote executions. For maximum benefit, this should also be coupled with the movement of data and programs from sites other than those where the execution is to be performed. Such distributed processing requirements are considered in the context of a Virtual Enterprise which uses an Object-oriented Knowledge Base Management System (OOKBMS) to act not only as a trader, but also the repository for a global model and shared data. Here, the concept of objects causes us to reconsider remote execution and data movement as the more general problem of object migration. In this paper, we identify object migration consisting of the migration of three possible components: object definition, object instances, and object method executables and source. We describe in this paper an exhaustive list of thirteen execution and data location scenarios. The discussion in this paper on object method execution covers not only the migration of executable code between homogeneous systems, but also interpretive code, such as Java, and also compilable programs (source code). Given these considerations, this paper finishes with the definition for a migration protocol using KQML. These performatives provide the mechanisms to support a wide variety of migration policies and are shown to support the thirteen distributed execution operation scenarios.

Keywords Data Migration, Legacy Data, Object-Oriented Databases

1. Introduction
In distributed systems that are constructed through the provision of services from many existing systems, there is a concept of binding as in Coulouris [1], also known as trading. This is where the providers of services register their services with the trader (a service itself). Registration includes the service's interface signature, address information and any additional information regarding the service being offered. Other systems wishing to locate a service may query the trader by providing any knowledge specifications that could be matched with the registered services. All information about any matching services are then returned to the requester. The requester can then choose which service they wish to use and communicate directly with the service using the registered interface and addresses. For some distributed systems though, trading of interface definitions is not sufficient.

In virtual enterprises (VE), complex interactions between the cooperating systems may occur that cannot be solved by the simple activation of advertised services. Interactions maybe activated by certain conditions occurring in individual systems or as a result of complex VE-wide queries. Such VE's are constructed by the cooperation of many legacy systems for which enhancements may be limited or restricted, thus the appeal of a more complex trading service.

In the National Industrial Information Infrastructure Project (NIIP)\(^1\), such a complex trading service is accomplished by using an Object-oriented Knowledge Base Management System (OOKBMS) to store the model of the VE and all its systems and services. For this system, a single common modeling language is used for all the cooperating systems. The model used is an object-oriented semantic association model described in Su [5]. This allows for services to be modeled as methods belonging to objects which also contain data components and rules. These rules not only model what occurs within the individual systems but can also be used to model the interactions between the systems. VE-wide queries can thus be accomplished by using the Object Query Language (OQL) provided with the OOKBMS. This system has been implemented at the University of Florida with minimal impact on the legacy systems. To accomplish this, each legacy system has a wrapper which translates actions and data specified in the common language and model into the local language and model and vice versa. In this way, interactions between legacy systems can be conducted directly

---

1 NIIP is an on-going project being carried out by a consortium of industrial companies, universities and government organizations. It is funded by DARPA (see acknowledgment).
using the common model. Possible interactions are shown in Figure 1 below.

![Figure 1 - Trading in a Virtual Enterprise](image)

Figure 1 represents the possible communication scenarios that may arise in the VE. Each legacy system may use its local model to formulate queries for the OOKBMS. These will be translated into the common language and model via the wrapper (shown as the shaded area encapsulating each legacy system). Results and requests from the OOKBMS will likewise be translated in the other direction. Also shown in Figure 1 are two ways in which the legacy systems may communicate between themselves directly. Legacy systems 1 and 2 communicate via the common language and model used by the OOKBMS, whereas legacy systems 2 and 3 may communicate directly with each other. This assumes that they have their own common language and model. Such optimizations of communication protocol are possible and could be used.

It should also be mentioned here that the OOKBMS in this case does not only act as a trader, but also stores information that is common or must be shared. For example, it may be necessary for data relating to relationships between objects from different legacy systems to be stored in the OOKBMS. This may be due to either limitations on additional data being stored in the legacy systems or for efficiency reasons. Even with such an effective trading and sharing mechanism, this existing system still has some limitations.

The current system is limited in its support for more general cases of distributed processing. It assumes that method executables which operate on object instances are always located in the same sites as the instances. In a distributed environment, object instances and method executables may be in different sites which may also be different from the clients that make the request for the operations and the sites that perform the operations. For this reason, remote procedure calls are not sufficient to handle such a problem. It must involve the movement or migration of all or part of the objects. In this paper we define object migration as the migration of three possible object components: object definition (often referred to as the class specification), object instances, and object method executables and source (program files).

Here we treat methods as a separate component to that of the object definition. The reason for this is that the granularity of methods must be finer than the set of all methods (and their executables) belonging to a class. Not only is the migration of executable code between homogeneous systems possible, but also interpretable code, such as Java, and compilable programs with different versions may be catered for. Thus heterogeneous systems can also be catered for. Through these considerations, thirteen execution and data movement scenarios can be identified.

Hence, the objective of this paper is twofold. First, the scenarios for remote executions and data movement need to be identified. Initially, this will be done without consideration of an object-oriented model. How this is viewed from an object-oriented perspective can then be covered. The impact this has upon an existing OOKBMS can then be determined and necessary changes to support object migration presented.

Second, present the definition of a protocol to handle object migration in a distributed system. This protocol should be the mechanism to support a wide variety of migration policies without unduly restricting them. We show that this protocol allows us to support all of the thirteen execution and data movement scenarios as they may arise in a distributed system. Protocols to activate the remote executions are not covered in this paper as they are more closely tied with distributed query processing which will be covered in a future paper.

The rest of this paper is structured as follows. Section 2 provides the basis for this paper. It presents the complete set of remote distributed execution operations possible in a distributed system. In Section 3, how this relates to an object-oriented model and more specifically an existing OOKBMS is covered. This includes a background to the structure of the meta-schema for an OOKBMS: the OSAM*KBMS developed at the University of Florida. The subject of object granularity is also discussed here as it has an impact on not only the structure of the meta-schema but also the object migration operations. Section 4 contains the definition of the protocol for the object migration operation specified in the Knowledge Query and Manipulation Language (KQML) defined in Finin [2]. It is shown to be able to support the scenarios defined in Section 2. Finally, in Section 5, conclusions are given with a brief discussion about other work being carried out in this area.
2. Distributed Execution Operations and Data Movement in a Distributed System

Remote Procedure Calls (RPCs) are a well-known aspect of Distributed System (DS), described in Coulouris [1]. In its simplest form, it is assumed that the procedure (usually as an executable) resides at the site where the RPC is being sent. Also assumed in this form is that any data not sent as an argument also resides at that site. These assumptions can limit the effectiveness of a DS. In this section, a more general form of the RPC, the Distributed Execution Operation (DEO), is considered.

DEOs differ to RPCs by eliminating all assumptions regarding the location of both data and executables (and their source) in relation to both the calling and execution sites. That is, an DEO involves the specification of not only where the execution is to occur, but also where the data is coming from as well as where the procedure (or function) to be executed is coming from. This is particularly important in a distributed system that makes extensive use of replication in both data and executables.

Hence, the request to perform the execution of a procedure or function (the executable) on a set of data, can be categorized based on several criteria:

- where is the execution to be performed with respect to the requester:
  - locally (L) - site of the requester or
  - remotely (R)
- if execution is local, where is the required data and executable located:
  - locally (L) - site of the requester or
  - remotely (R)
- if execution is remote, where is the required data and executable located:
  - locally (L) - site of the requester or
  - execution (E) - site where execution is to occur
  - remotely (R) - not local nor the site of execution but another site

To adequately show these exhaustive location combinations, two tables are presented. The first table covers the local execution and the second table covers the remote execution. Labels for these tables are taken from the lists above, namely L, R, and E.

For example, category L4 is the scenario where an execution is to be performed locally, but the executable and the data are both stored at remote sites. This means that the executable and the data would need to be migrated to the local site before execution could begin. In category R5, the scenario is one where the execution is to be performed at a remote site. Here though, both the executable and the data are stored at sites other than the local requesting site and the execution site. They may or may not be stored together.

<table>
<thead>
<tr>
<th>Category</th>
<th>Location of Executable</th>
<th>Location of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>L2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>L4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 - Local Execution Scenarios

<table>
<thead>
<tr>
<th>Category</th>
<th>Location of Executable</th>
<th>Location of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>R2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>R5</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 - Categories for Remote Executions

The environment in which these scenarios are currently being investigated is object-oriented. More specifically, the environment is an OOKBMS. In order to cover the impact of such a model on DEOs, a more thorough discourse on the OOKBMS is required. This and its impact on DEOs is covered in the next section.

3. Remote Execution for Objects

The view taken in this paper is that the entire distributed system can be modeled in an object-oriented manner, even if its components are not all object-oriented in nature. Such an approach has been taken in the NIIP project at the University of Florida. In order to better understand the work on object migration, this section presents in more detail both the architecture and model used, namely the OSAM*KBMS.

3.1 OSAM*KBMS Structure

The OOKBMS that has been developed and is under further extension at the University of Florida is the extensible OSAM*KBMS Su [8]. This OOKBMS is based on a semantic association model, OSAM*, as described in Su [5][7] and is an active OOKBMS Su [5]. This system is currently used to coordinate and model the services and design of a VE as described in Section 1. Not only does it model existing interfaces, but it can also be used to generate interface stubs. These can be used to either create further services at the legacy system end or for clients to use to access the remote services. Currently, the OSAM*KBMS
system is made up of several components as shown in
the figure below.

![OSAM*.KBMS Architecture](image)

Figure 2 - OSAM*.KBMS Architecture

Details of these components can be found in Su [8]. What is of more interest at this point is the model
that this architecture implements. The object model
used is defined by a meta-model. A part of the current
version of the meta-model, defined OSAM* is shown
in the diagram below.

![Meta-Model](image)

Figure 3 - Meta-Model

The primary components of the model are classes
and the associations between them. Classes are
defined by attributes, methods and rules. As with
most object-oriented models the concepts of
generalization (labelled by a G) and aggregation
(labelled by an A) are catered for. Here though they
are included with the other possible associations
between classes. They provide a much more rich
foundation to describe relationships between objects.
These include: Generalization (inheritance),
Aggregation and Interaction. Other work at the
University of Florida have added to these three using
a model extensibility technique, but for the purposes
of this paper, this basic set is sufficient. Examples of
Aggregation and Generalization can be seen in the
Meta-Model above. For example, the class Class
is a
generalization of the classes Entity and Domain.
Class has two aggregation associations with domain
classes (i.e., two attributes) and three
aggregation associations with three entity classes. The two string
attributes are schema and name. The other three

aggregation associations are with the classes Method,
Rule and Assoc are of the set type. This allows for
each class definition to contain a name, the schema it
belongs to, a set of methods, a set of rules, and a set
of associations with other classes. Rules in this model
are Event-Condition-Action-AlternativeAction
(ECAA) rules.

In the meta-model above, the class Assoc also has
a set association with the class AssocLink. This shows
that an association can be defined by many links.
Assoc is the generalization of the association classes
Generalization, Aggregation and Interaction. For a
detailed explanation of the association types see Su
[5]. Not shown in this meta-model is the fact that
every object instance stored in the OOKBMS is
identified by an Instance Identifier (IID). This is
made up of two components: Object Identifier (OID)
and a class identifier. In this way, given an object
instance, the KBMS not only can uniquely identify
the object, but also the class to which the object
belongs.

Within this meta-model, each method is defined
separately. In reality, the method definition is but an
interface definition to some functional operation
acting upon the object. The actual executable program
(i.e., the method implementation) may include many
calls to other methods. This is not a problem if the
executables for all the methods in a class are to be
moved for an DEO. Unfortunately, this may not be
desirable. Especially in cases of query processing,
migration may only be temporary and the speed of the
migration a critical factor. If not all methods are
required, unnecessary transfers should be avoided.
Therefore, the granularity of methods and executables
needs further discussion before their migration can be
considered and described.

---

2 As development is ongoing, different extensions to the meta-model are in
existence for many of the current projects at the University of Florida. A
common base model is presented here.
3.2 Granularity and Migration

Using an Object-oriented Model (OOM), the view on DEOs is different to that shown in Section 2. In an OOM, all executables are contained within an object. That is, the interface to any method is accessed via an object and hence defined within the object specification. Such an object specification is defined within a class definition as it pertains to many object instances. Therefore, in an OOM, a remote execution acting upon a set of data is seen as the activation of a method defined within a common class specification for a set of object instances.

Hence, to carry out such remote executions as described in Section 2, requires the movement of not only the object instances, but possibly the object definition if it has not been defined at the destination site. This includes not only the specification of the state information contained in an object instance, but also the method specifications and references to the method implementations. Such references are links to execution entities that are stored external to any OOOKMS. That is, the method executables are stored in the external file system. Therefore it is also necessary to migrate the actual executable code. The form that these executables takes needs further explanation.

In the OSAM*.KBMS, there is no restriction on the form that these executables can take, so long as an interface can be specified and a "hook" to the executable. This allows a wide variety of execution types to be accessed, from new interpretive languages such as Java in SAMS [4] and Flanagan [3] to older legacy systems implemented in COBOL or C and C++. Such a range highlights the point that we may not wish to restrict the migration of executables to binaries. In the case of Java, it is in the form of interpretive byte format. This could be extended in legacy systems to include portable source code. In this manner, source code could be migrated and compiled before use. Such generalizations are currently not supported by the meta-schema of OSAM*.KBMS. To cater for such extensions the following updated meta-schema is proposed.

Additions to this meta-model have been shaded for ease of identification. The first addition is the Site class. It identifies sites where not only are objects defined, but also where method source code and executables can possibly be stored. The unit of storage for these methods is not at the individual method level as this is too fine a granularity. As it may be necessary to move a group of related methods together, an artificial unit of storage is created. That is, a collection of related methods to be allocated together is needed. The container for such a collection is specified by the class MethodAlloc. This class is defined by an aggregation association with the Method class. It defines a set of Methods to be treated as a unit. Actual storage can now be identified by the classes MethodExec and MethodSource. Both classes are created by the interaction association between Site and MethodAlloc, where the interaction association models the relationship between sites and sets of methods.

In these two classes, MethodExec identifies where the executables are stored. MethodSource identifies where the source files are stored. In the case of MethodSource the compiler method is also identified by an association. This too may need migrating. These specifications can also be extended to cater for different versions of source code and executables for different sites. This allows for heterogeneous architectures in a VE.

Given the meta-model shown in Figure 4, we are now able to specify the identifiers needed in an DEO call. All object instances are identified by IIDs. This
includes the objects specified in the meta-model, the object specifications, such as the object Class or its specializations. As stated in Section 3, it is possible to identify an object type, or class, by its IID.

Further work is needed in specifying precisely how objects and their states are marshaled and packed for migration, but given an object-oriented model, the use of such features as virtual functions in this process would not be unreasonable. The emphasis here though is on identifying the major components for migration. Based on the focus of this paper being on DEOs and the discussion in Section 2, three major categories of objects present themselves. They are object specifications (class definitions), object instances (their state values) and program files (either executable or source). Given these, the protocols to handle their migration can now be specified.

4. Migration Protocols

The objective of this section is to define protocols that facilitate the migration of objects in a DS. In this work these protocols will be used by Object Managers or wrappers at various sites in the VE. It is not the subject of this paper to determined which objects need to be moved between sites. This is the subject of future papers on Distributed Query Optimization and Processing. Here, protocols are specified using the Knowledge Query and Manipulation Language (KQML) Finin [2].

4.1 Protocol Interfaces

Given the requirements of object migration as described above, the set of protocols defined here allow for the interaction between three conceptually different sites. These sites are: the Requesting Site (RS), the Storage Site (SS) and the Destination Site (DS). The migration process is achieved via messages in the form of KQML performatives. An outline of the process is as follows:

1. The RS sends a request for migration to the SS
2. The SS processes the request by finding the required objects
3. The SS migrates the requested objects to the DS
4. The DS stores the migrated objects and replies once this has been successfully done
5. Once the SS receives a success status from the DS it sends a similar message to the RS

Error messages will be sent should problems arise. Much of the explanations of the individual KQML performatives and their parameters are contained in the comments attached to their specifications.

In the migrate-request performative, the :content parameter specifies an expression that can be evaluated by the SS to form a set of IIDs. This may be in the form of either IIDs or a query. The next three parameters specify the components of the objects to be moved. That is, if :class is specified as T then the class definition is migrated. Similarly, if :instance is specified as T then instances of the class are to be migrated. For :method, three options are available. NIL causes no methods to be migrated. If SRC or EXEC are specified, then the external file containing the source or executable specified in the next parameter, :method_specification, will be migrated.

The next parameter, :persistence, indicates to the OOKBMSs what further operations are required after the object is migrated. If Move is specified, then the object is deleted from the SS once the migration is completed. If Copy is specified, then the OOKBMSs must maintain consistency between the copy at the DS and the copy at the SS. Finally, if Snapshot is specified, then only a temporary copy is being made. No deletion or consistency maintenance is required. The object will be deleted once its usefulness is at an end.

For an acknowledgment of the migrate-request message, the :in_reply_to parameter can be set to T. If it is set, then once a successful migration is completed an acknowledgment message will be sent using the expression passed in the :reply_with parameter. Any errors messages will also be sent with this expression. It is assumed that if the :in_reply_to parameter is not set, then no acknowledgment will be sent. Even if an error occurs, no message will be sent unless the :in_reply_to parameter is set. This allows for migration policies that do not want such reply messages. This may be the case when the same request are sent to several sites and only the first migrated object is used.

The final four parameters relate to the identification of sites, both sending and receiving. They cater for situations when requests may be passed onto other sites but the original site identifiers are needed. This is consistent with many of the standard KQML performatives.
Once the migrate-request message has been received by the SS and the objects found, they are then packed into a :content expression for sending using the migrate performative defined below:

migrate
: content <expression> ; packed object values
: packed_as <expression> ; packed how e.g. zipped
: class <expression>; include definition NIL or T
: instance <expression>; include instances NIL or T
: method [NIL SRC EXEC]; methods and form
: method_specification <expression>; methods
: persistence {Move, Copy, Snapshot}; delete
: original, maintain this copy or not
: in_reply_to <expression> ; reply expected NIL or T
: reply_with <expression> ; reply with this
: sender <word>; if different to "from"
: receiver <word>; if different from "to"
: to <word> ; object/s sent to
: from <word>; where are objects stored

Many of the parameters of this performative are the same as those in the migrate-request performative. The ones that are not the same require explanation.

As the data being sent may include definitions, state values, or source and binaries, they may be packed differently in the message. Hence, the :packed_as parameter details what method has been used to create the expression being passed in the :content parameter. This may include the specification of external applications or even simple comma delimited, quoted text.

Given these performatives, their coordinated use can now be discussed.

4.2 Performative Use

The migration performatives must be able to support each of the scenarios defined in Tables 1 and 2. In each case, the three types of objects - definition, instance, and file - may need to be migrated. To show that each case can be handled, we describe what would need to be done for the most complicated scenario. All other scenarios are then simplifications of this.

The most complicated scenario arises in category R5. Here, the execution is to be performed at a remote site. Neither the data nor the executable is stored at that site. To complicate this further, assume that the object definition is not at this execution site either. Therefore, all three types of objects must be moved from a remote SS to a remote DS where the execution is to be carried out.

As the process of storing instances and external files in an OOKBMS may be dependent on the class definition, this should always be moved first. Once this definition has been successfully migrated, both the instances and external files can be migrated sequentially or in parallel.

For each of these objects, including the definition, the order of operations is depicted in Figure 5. This is consistent with the outline presented at the start of

![Figure 5-Sequence of Migration protocol method calls](image-url)
Section 4.1. Variances may occur when erroneous situations arise mid-operation. For example, if the SS does not have the required object or is unable to successfully transfer the required object, then appropriate error response performatives may be issued. These performatives may or may not include detailed error codes depending on the problem.

In all of the other scenarios described in Tables 1 and 2 for DEOs, some object or operation is performed locally rather than remotely. The migration requirements for these are therefore simplifications of that in Figure 5 and are achieved by eliminating some or all of the performatives. For example, the requesting site may also be the storage site, hence the migrate-request performative can be eliminated. Only the migrate performative would be required along with any acknowledgment performatives.

5. Conclusions and Future Work

In this paper, the problem of supporting the object migration requirements for distributed execution operations in a distributed environment was presented. This was first presented as a non-object-oriented problem with an exhaustive list of thirteen possible DEO scenarios. The problem of moving data and executables to support such operations was then viewed from an object-oriented perspective. This was necessary as the major problem being considered is the support of distributed processing in a VE. Development of such support is currently being carried out at the University of Florida. Having presented the support requirements for the DEO problem in an object-oriented manner, a solution was presented. This solution was in the form of KQML performatives to carry out the migration of objects and their three constituent parts: definition, instances and external program executables and sources. These performatives were then shown to be sufficient to support the thirteen categories of DEOs.

Work is currently under way to use this support to aid in Distributed Query Processing and Optimization (DQPO) using the OSAM*.KBMS as an advanced trader and data sharing environment. This work is based on using the OSAM*.KBMS to support all aspects of DQPO including: support data for query optimization, query plan specification and query execution control.

Acknowledgment

This work is supported by the Advanced Research Project Agency under ARPA Order #B761-00. It is part of the R&D effort of the NIIP Consortium. The authors have benefited from discussions with the NIIP consortium members, Art Goldschmidt of IBM and Herman Lam of the University of Florida. The ideas and techniques presented here are those of the authors and do not necessarily represent the opinion of other NIIP Consortium members.

During the time of this research George Semeczko was on Professional Development Leave from the Queensland University of Technology.

References


