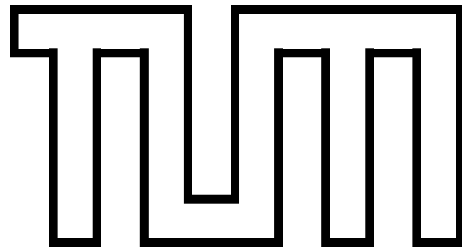


Technische Universität München

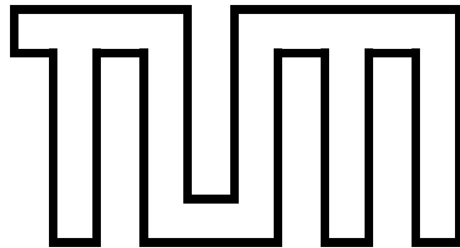
Fakultät für Informatik

Lehrstuhl für Rechnertechnik und Rechnerorganisation

Diplomarbeit

Evaluation of Interoperable Tool Deployment for the Late Development Phases of Distributed Object-Oriented Programs

Alexandros Stamatakis



Technische Universität München

Fakultät für Informatik

Lehrstuhl für Rechnertechnik und Rechnerorganisation

Diplomarbeit

Evaluation of Interoperable Tool Deployment for the Late Development Phases of Distributed Object-Oriented Programs

Alexandros Stamatakis

Themensteller: PD. Dr. Thomas Ludwig

Betreuer: Dipl.-Inform. Markus Lindermeier
Dipl.-Inform. Günther Rackl

Abgabetermin: 15. Februar 2001

Erklärung

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 2001

____
(Unterschrift des Kandidaten)

Acknowledgments

Several people have contributed to this diploma thesis.

I would like to thank PD. Dr. Thomas Ludwig for his help concerning this thesis and especially for supporting my studies at the Ecole Normale Superieure de Lyon, as well as an internship with the Eurocontrol Experimental Center near Paris in earlier years.

Furthermore, I want to thank Dipl.-Inform. Markus Lindermeier and Dipl.-Inform. Günther Rackl for the excellent help and guidance they provided me, as well as for the pleasant and relaxed atmosphere during our meetings.

Finally, I am thankful to Dipl.-Inform. Marcel May, who implemented the C++ version of the realignment application and explained to me the details and the structure of his program.
I would also like to thank Jörn Eichler, who implemented parts of the automatic load balancer within the framework of his diploma thesis and helped me with the integration of the load balancer.

Contents

1 Introduction **1**
1.1 Motivation and Goals . . . 1
1.2 Structure of the Thesis . . . 2

2 The Java-based Realignment Application **5**
2.1 Application Background: Medical Image Processing . . . 5
2.2 Structure of the Sequential C++ Program . . . 6
2.2.1 Input Parameter Format . . . 7
2.2.2 Realignment Algorithm . . . 7
2.3 Technical Background: Java Native Interface (JNI) . . . 9
2.4 Encapsulating the Original Program in Java Code using JNI . . . 10
2.5 Verification and Performance Evaluation . . . 13
2.5.1 Verification . . . 13
2.5.2 Performance Test: C++ versus Java/JNI/C++ . . . 13

3 The CORBA-based Client/Server Realignment Application **15**
3.1 Technical Background: CORBA . . . 15
3.1.1 Object Management Architecture . . . 16
3.1.2 CORBA Object Model . . . 17
3.1.3 Portable Object Adapter . . . 18
3.1.4 The Interface Definition Language (IDL) . . . 18
3.2 Design of the Client/Server Architecture . . . 19
3.2.1 Splitting the Java Program into Client and Server Components . . . 19
3.2.2 Definition of the IDL Interface . . . 24
3.3 Verification and Performance Evaluation . . . 25

4 Integration of the Load Balancer **29**
4.1 Technical Background: Load Management . . . 29
4.1.1 The Automatic Load Balancer . . . 30
4.2 Integration of the Load Balancer into the CORBA Program . . . 31
4.2.1 Basic Version . . . 32
4.2.2 Single and Multiple Object Mode . . . 33

4.2.3 Multi-Threaded Client . . . 33
4.2.4 Cache Architectures . . . 36
4.2.5 Persistent Migration/Replication . . . 37
4.3 Verification and Performance Evaluation . . . 38
4.3.1 Manual versus Automatic Replication . . . 38
4.3.2 Multi-Threaded Clients . . . 39
4.3.3 Migration and Replication . . . 40

5 MIMO and MiVis 45
5.1 Technical Background: Middleware Monitoring Systems . . . 45
5.1.1 General Aspects . . . 45
5.1.2 Existing Technologies . . . 46
5.2 Introduction to MIMO . . . 46
5.2.1 Multi-Layer-Monitoring (MLM) . . . 48
5.2.2 Instrumentation Techniques . . . 48
5.2.3 Generic Events . . . 50
5.2.4 Active Tools . . . 52
5.3 Introduction to MiVis . . . 52

6 Interoperability of MiVis and the Load Balancer 55
6.1 Specification of the Degree of Interoperability and of the Tool Functionality . . . 55
6.2 Instrumenting the Load Balancer and the Application . . . 56
6.2.1 Initial Approach and Associated Problems . . . 57
6.2.2 Multi Layer Monitoring Mapping . . . 59
6.2.3 Visualization Interface Definition . . . 59
6.2.4 Command Interface Definition . . . 62
6.2.5 Extension of the Manual Adapter . . . 62
6.2.6 Instrumentation and Command Implementation . . . 63
6.3 The new MiVis Java Bean . . . 64
6.3.1 Display Design . . . 64
6.3.2 Java Bean Implementation and Improved MiVis Design Proposal . . . 66
6.4 Instrumentation of Load-Balanced Applications . . . 68
6.5 Evaluation of Tool Functionality and Interoperability . . . 69

7 Conclusion 73

Abbreviations 75

List of Figures

1.1 Outline of the Development Process 3

2.1 Realignment Algorithm 9
2.2 Java/JNI Program Structure 12
2.3 Performance: C++ versus Java per Image 14
2.4 Performance: C++ versus Java per Sequence 14

3.1 Object Management Architecture 17
3.2 CORBA Client/Server Development Process 19
3.3 Server Structure and Control Flow (reference data in cache) . . . 22
3.4 Server Structure and Control Flow (reference data not in cache) . 23
3.5 CORBA Overhead per Image 26
3.6 CORBA Overhead per Sequence 27

4.1 Basic Components of a Load Management System 30
4.2 Two Models for a Multi-Threaded Client 34
4.3 Dynamic Thread Model 36
4.4 Multiple versus Single Object Mode 39
4.5 Load Balancement of a Multi-Threaded Client 40
4.6 Object Replication for a Multi-Threaded Client 41
4.7 Migration of Client Objects 41
4.8 Stateful versus Stateless Replication 42
4.9 Stateful Migration for a Cache-less Client 43

5.1 MIMO Architecture 47
5.2 MLM Model 49
5.3 Instrumentation Alternatives 50
5.4 MiVis Architecture 54

6.1 Virtual and Real IORs 57
6.2 Alternative MiVis Design Proposal 65
6.3 Migration Command Execution 67
6.4 Handling of Dynamic IOR Changes 68
6.5 Screen-shot: New Display Layout 70
6.6 Screen-shot: Representation of Replications 71

6.7 Screen-shot: Execution of a Migration Command 71
6.8 Screen-shot: Complex Replications 72

Chapter 1

Introduction

1.1 Motivation and Goals

The introduction of the distributed object-oriented programming paradigm, as implemented for example by the Common Object Request Broker Architecture (CORBA), provides a finer distribution granularity paired with an increase of complexity.

Therefore, a set of powerful tools is required for handling the increased complexity of distributed object-oriented systems and applications.

Firstly, there is a need for tools supporting the development process of distributed object-oriented applications, for monitoring and visualizing interactions, object creations, events etc. This motivates the development of middleware monitoring systems.

Secondly, the parallelization potential offered by the finer distribution granularity should be exploited in an advantageous manner. As a consequence, automatic load balancing mechanisms are required, permitting to handle the complexity induced and to take advantage of the degree of granularity provided.

Thirdly, the monitoring and steering of an automatic load balancer and the associated load-balanced application(s) using a powerful monitoring tool, is important for the improvement of the load balancer and the development of load-balanced applications. In addition to this the possibility to steer the load balancer using a visualization tool, enables for example the system administration to perform maintenance work and to keep track of actions performed. Furthermore, the interoperability of a middleware monitoring system with such a complex system has to be investigated.

Within this context, an automatic load balancer and a middleware monitoring system for distributed object-oriented systems are currently under development at the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR).

The goal of this thesis was to test and evaluate those tools with a real world

application. As indicated by the title, the final phase consisted of analyzing interoperability issues concerning those tools and of developing concepts for monitoring and steering the load-balanced system[1]. This was achieved by designing a new graphical display based on the MIddleware MOnitoring System (MIMO) and the closely related visualization tool Middleware Visualization System (MiVis). A medical application, which had been parallelized in an earlier diploma thesis using C++/PVM and C++/CORBA respectively, was chosen as real world application.

1.2 Structure of the Thesis

The *first* part of this thesis (chapters 2, 3, and 4) describes a series of program transformations, necessary for the integration of the load balancer, leading to an increase of system complexity each time.

With the original sequential C++ program as a starting point at each step a new concept or technology is integrated. The motivation and task of the specific step will be stated in each chapter. Furthermore, a brief description of the new technology introduced and of the evolved program structure will be given. Finally, the results of the program verification and evaluation process will be presented at the end of each chapter.

The *second* part (chapters 5 and 6) introduces the monitoring system MIMO and the visualization tool MiVis. A new visualization tool (including several new components and interfaces) for load-balanced systems (see footnote), based on MIMO and MiVis, will be presented.

As outlined in figure 1.1 the chapters cover the following subjects:
Chapter 2 describes the medical application , the original sequential C++ program, and its transformation into a sequential Java program using the Java Native Interface (JNI).
Chapter 3 gives an introduction to CORBA and describes the transition from the sequential Java code to a sequential Java/CORBA program based on a Client/Server architecture.
Chapter 4 introduces the load balancer and its basic mechanisms and covers the modifications necessary to adapt the initial CORBA program to the restrictions and requirements imposed by the load balancer. It presents results for the degree of parallelization obtained by the application of the load balancer.
Chapter 5 explains the concepts of the MIddleware MOnitoring system (MIMO) and the associated MiVis visualization tool.
Chapter 6 describes the integration of the above tools with the load balancer and the application. It presents the resulting extended MIMO interface and the new MiVis Java Bean, a display which visualizes the actions performed by the load balancer and the interactions of the load-balanced application. An additional

[1]A system consisting of the load balancer and the load-balanced application(s).

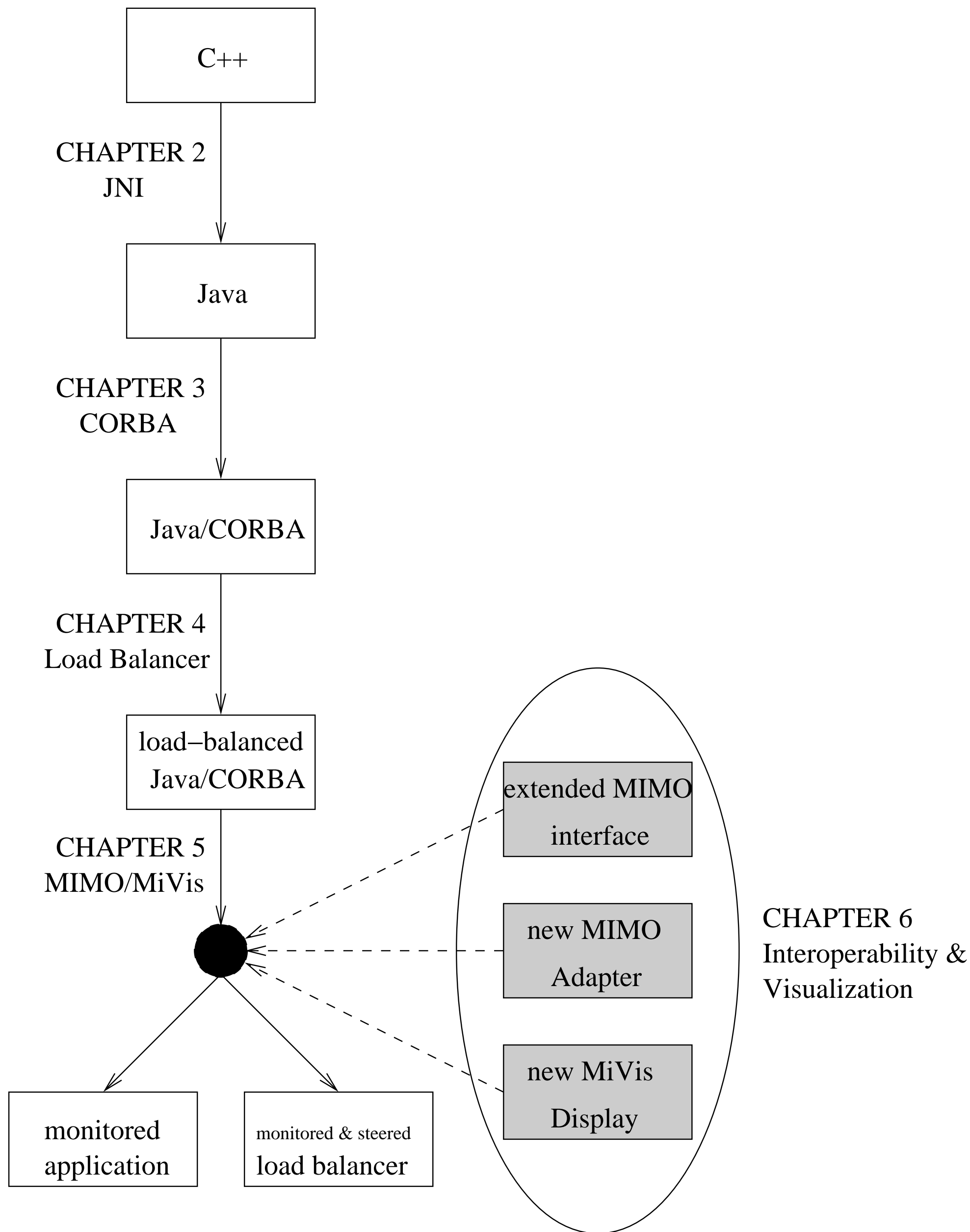


Figure 1.1: Outline of the development process.

feature is a drag and drop function for the execution of user-initiated migrations. Furthermore, it presents a guideline for instrumenting load-balanced applications and proposes an alternative architecture for the MiVis system.
Chapter 7 forms the conclusion of the work conducted.

Chapter 2

The Java-based Realignment Application

The load balancer, which will be integrated into the program later on, only provides services for Java/CORBA programs. Therefore, it was necessary to transform the original C++ realignment code into an equivalent Java program. Since the C++ source code contained well over 5000 lines of code, porting the entire program to Java was not possible due to lack of time and the complexity of the code. An alternative approach was chosen using the Java Native Interface (JNI), in order to encapsulate the C++ source code in Java.
The strategy consisted of transforming the sequential C++ Code into a sequential Java program, always keeping in mind however, that the resulting program had to be split into a client and a server component at the next step. Thus, Java methods had to be provided especially at those points where a remote method invocation would be performed later on.

2.1 Application Background: Medical Image Processing

The realignment process, that has been chosen for this thesis as real world application, comes from the field of medical image processing.
Realignment forms part of the Statistical Parametric Mapping (SPM) application developed by the Wellcome Department of Cognitive Neurology in London (see [4]). SPM is used for processing sequences of images, as obtained for example by functional Magnetic Resonance Imaging (fMRI) or by Positron Emission Tomography (PET). Sequences of such images, spaced at close time intervals are used in the field of neuroscience, for analyzing the activity of different regions of the human brain during cognitive and motoric exercises.
Realignment forms part of the preparatory computations SPM conducts, in order to be able to perform statistical operations on the obtained image data. For the

preparation of a series of functional medical images SPM performs the following 4 computational steps on the raw image data:

1. Registration and *Realignment*
2. Normalization
3. Coregistration
4. Smoothing

Realigning the images means, that the effect of small movements, caused for example by the breath of the patient, are filtered out of the sequence by calculating a 4 x 4 transformation matrix. This process has previously been selected for parallelization, in order to conduct a performance comparison between its Parallel Virtual Machine (PVM) and CORBA versions (see [3]). The realignment process is one of the most cost expensive computations SPM performs and contains few dependencies, i.e. is easy to parallelize.
A sequential C++ version had also been developed, since it was necessary to extract the realignment part from the SPM program which is written in MATLAB (see [5]). This program version was the starting point of this thesis.

2.2 Structure of the Sequential C++ Program

As mentioned in the above section the sequential C++ program only performs the realignment computation, which forms part of the preparatory computations conducted by SPM.

The realignment process was selected mainly due to its cost expensive computations. Medical as well as mathematical details are only secondary for this thesis. Therefore, only a brief overview over the program's structure will be given, neglecting unimportant details, since the program was considered as a set of black boxes, for which several JNI interfaces had to be constructed.
The description is split into two parts, the format of the parameter string passed to the program is specified and an abstract description of the realignment algorithm is given.

2.2.1 Input Parameter Format

The input parameter string consists mainly of one or more sequences of image filenames and some numerical parameters. For the exact semantics of the numeric parameters see [3]. The format of the parameter string is specified by the following BNF like grammar:

```
PARAMETERLIST ::=  HEAD SEP {SEQUENCELIST}
HEAD          ::=  Quality SEP SmoothingCoefficient SEP
                   GridSpacing SEP RTM SEP [ImageWeight]
                   SEP Hold SEP NumOfSequences
SEQUENCELIST  ::=  SEQUENCE [SEP SEQUENCELIST]
SEQUENCE      ::=  NumOfImages {SEP ImageName}
RTM           ::= 0|1
SEP           ::= "|"
```

Example parameter string for one single image sequence with 10 images:

```
0.5|6|4.5|0||-8|1|10|data/01.img|...|data/10.img
```

2.2.2 Realignment Algorithm

There exist two different realignment algorithms depending on the input data, i.e. whether the images are obtained by Positron Emission Tomography or functional Magnetic Resonance Imaging.
The C++ code however implements the algorithm for the realignment of fMRI data only. The input data may consist either of a sequence of images or of a sequence of sequences of images. For each image of the input sequence(s) a real-valued 4 x 4 transformation matrix is calculated.

The algorithm is presented in plain text, in a C-like pseudo-code, and graphically (see figure 2.1).

1. *Case:* Realignment of *one* sequence of images.

   The algorithm consists of three basic steps. Preparatory computations are performed on the first image of the sequence, in order to calculate the reference data set.
   Thereafter the remaining images of the sequence ($2...n$) are realigned relatively to the first image of the sequence, using the previously calculated reference data.
   Finally, the obtained transformation matrices of images $1...n$ are saved to disk.

```
void realignSingle(Image images[])
{
   load(images[1]);
   referenceData = preparatoryComputations(image[1]);
   //Step 1 in figure 2.1

   for(int i = 2; i <= images.length; i++)
      {
         load(images[i]);
         compute(images[i], referenceData);
      }
   //Step 2 in figure 2.1

   saveTransformationMatrices(images[]);

}
```

2. *Case:* Realignment of *multiple* sequences of images.

   Let m be the number of sequences. During the initial step the first image of sequences $2...m$ (i.e. $2, 1...m, 1$) is realigned relatively to the first sequence's image $(1, 1)$ and its reference data.
   Afterwards each sequence $(1...m)$ is realigned independently by applying the single sequence algorithm.

```
void realignMultiple(Image images[][], int numOfSequences)
{
   load(images[1][1]);
   referenceData = preparatoryComputations(images[1][1]);
   //Step 1 in figure 2.1

   for(int i = 2; i <= numOfSequences; i++)
      {
         load(images[i][1]);
         compute(images[i][1], referenceData);
      }
   //Step 2 in figure 2.1

   for(int i = 1; i <= numOfSequences; i++)
      {
         realignSingle(images[i]);
      }
   //Steps 3 and 4 in figure 2.1
}
```

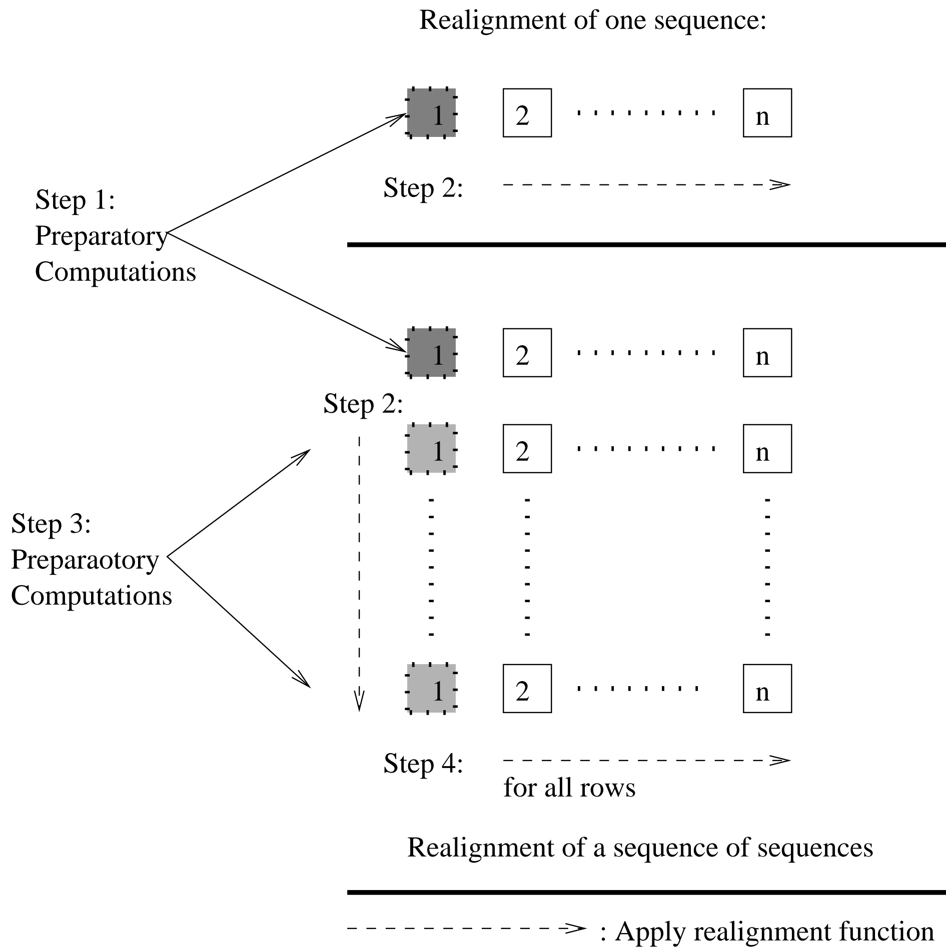


Figure 2.1: A graphical representation of the algorithm for both cases, a simple sequence of images and a sequence of sequences of images.

2.3 Technical Background: Java Native Interface (JNI)

The reason for the utilization of JNI (see [1]) has already been given in the introduction of this chapter. JNI provides a means for integrating native code, usually C or C++ functions into a Java program, by defining *native* methods. This however leads to a loss of one of Java's key features: machine independence. Furthermore, the extensive usage of native methods, as well as passing large amounts of data from Java to the native code and vice versa, leads to performance penalties.
Thus, the overall number of *native* method calls should be reduced to a minimum, as well as the amount of data passed through the interface. For a thorough

analysis of JNI performance issues see [2].

When one wishes to integrate native code into his Java program the steps are as follows:

1. Declare the native method and its signature using the `native` modifier in the Java code.

2. Generate automatically the JNI/C++ header-file, by compiling the Java code with `javah -jni`.

3. Implement the C++ methods defined in the generated `.h` file.

4. Compile the native method implementation and store it as shared library file (e.g. `libName.so` for Solaris).

5. Load the shared library using the Java method `System.loadLibrary()`.

Example:
A classical example for the utilization of JNI is the acquisition of the Process Identification Number (PID). The Java definition would be as follows:

```
public static native int getPID();
```

The following C++ function implements `Java_JNITest_getPID()` which is defined in the automatically generated header file:

```
JNIEXPORT jint JNICALL Java_JNITest_getPID(JNIEnv *, jclass)
{
   int pid = (int)getpid();
   return (jint)pid;
}
```

2.4 Encapsulating the Original Program in Java Code using JNI

The most important point of this part was to select the appropriate C++ function or set of functions which have to be executed by one single *native* method call. This selection and grouping was done using two criteria.

1. *Major criterion*: Provide Java native methods at those points, such that an easy and efficient transition two a Client/Server architecture is possible later on.

2. *Minor criterion*: Reduce the number of Java native method calls and the amount of data exchanged to a minimum for better performance (performance considerations are however only secondary at this point).

Since the program has previously been parallelized (see [3]), it is relatively easy to identify the pieces of C++ code that have to be executed by one single Java method call. In this case it is mainly the `compute()` function which is situated at the inner `for` loop.
The sole dependency is, that a call to `compute()` can only be performed when the respective reference data set is available. Once available, the realignment matrices of all images of the sequence can be computed independently. Thus, the `compute()` method will be the service provided by the server later on and was therefore defined in a separate Java class (`Compute`).

A second Java class (`Realign`) was designed, representing the future client, providing additional native methods for parsing, calculating the reference data set etc. The structure and control flow of the Java/JNI/C++ program for the realignment of a *single* sequence of images is depicted in an abstract manner in figure 2.2.

The encapsulation was performed as follows:

- Define an appropriate native method `parse()` for calling the already existing C++ function which parses the parameter string (see 2.2.1).

- Rewrite the main control loop of the program (see pseudo-code) in Java.

- Define the pseudo-code methods as Java native methods.

- Implement corresponding C++ functions, which transform the JNI parameters and call the original C++ functions.

A difficulty that arises when working with JNI in combination with C++ is the constant switch between two completely different memory management models. Especially with the realignment application, where great amounts of data are passed back and forth through JNI and a lot of memory is allocated dynamically, this lead to difficulties associated with memory allocation in the C++ code.

Furthermore, there was no documentation available clearly describing how the heap and stack of the native code are managed. The conclusion was drawn, that memory issues have to be handled with *extreme* care when writing this type of program.

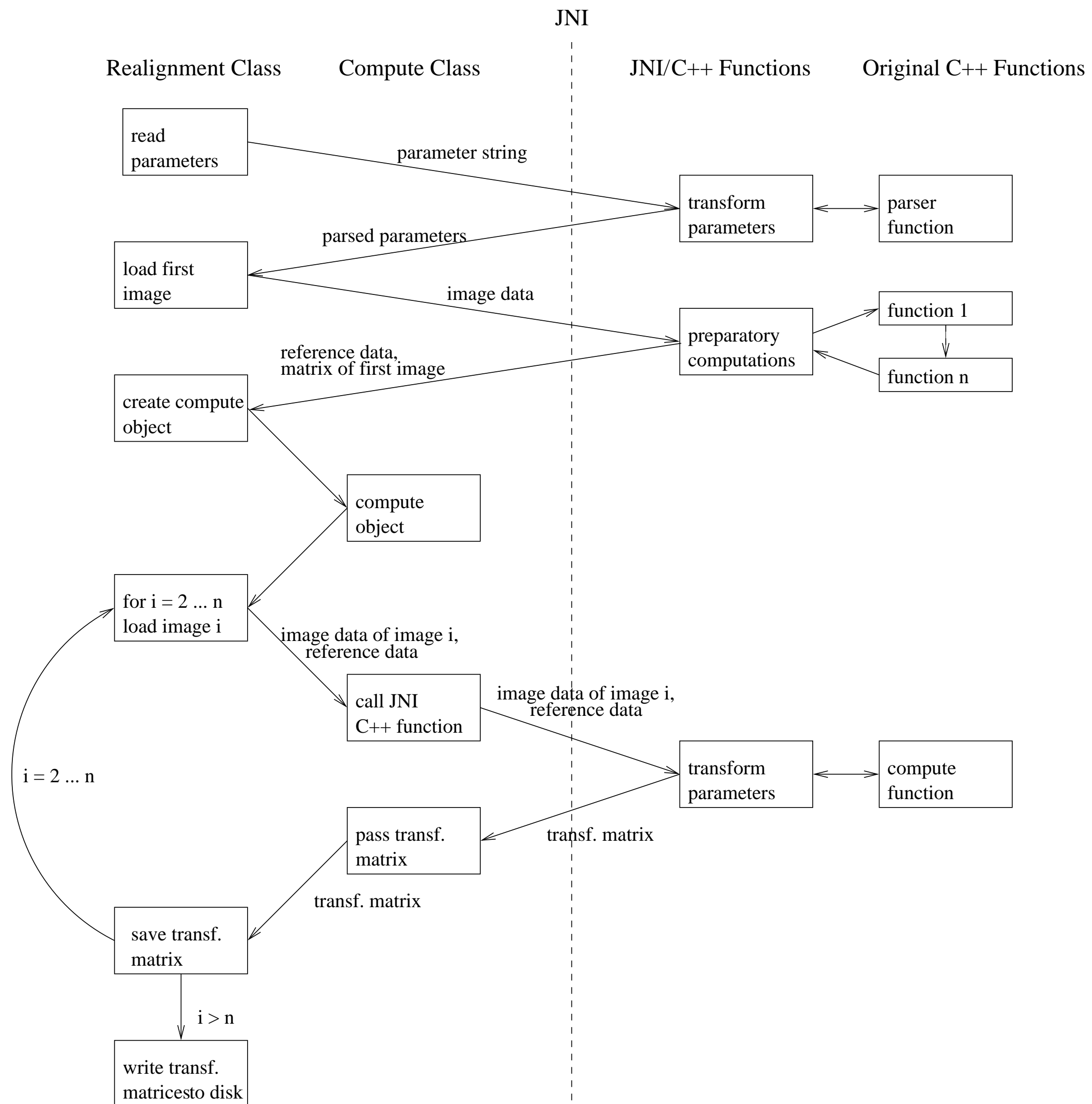


Figure 2.2: The calls and data traversing the Java Native Interface as well as the control flow of the program for a *single* sequence of images are shown.

2.5 Verification and Performance Evaluation

2.5.1 Verification

Program verification, i.e. showing that both programs are equivalent was particularly easy and straight-forward in the specific case. Since the C++ program produces a set of 4 x 4 matrices as output (one matrix per image), it was easy to automatically compare this set with the results obtained by the Java version.
The C++ native code of the Java program had however to be modified, in order to write the output data to a different directory.
A small C function `verify()` was implemented for comparing the matrices. `verify()` expects a sequence of image file names in the same format as specified for the realignment application (see variable `SEQUENCE` in 2.2.1) as input and returns `true` if all files in the Java and C++ result directory are equal, otherwise it indicates which files are erroneous.

2.5.2 Performance Test: C++ versus Java/JNI/C++

Another interesting aspect of the first part of this thesis was the possibility to compare the performance of the pure C++ implementation with the performance of the Java/JNI/C++ version of the program and some rather surprising results were obtained. The interesting aspect was, that a particularly large amount of data is being passed via JNI (about 0.8 Mb per image), and therefore a serious performance penalty had to be expected.
The initial tests confirmed what had been expected, a performance 20 to 30 % worse than that of the C++ code. Later on in the project it became evident, that the shared library file had not been compiled using optimization flags, in contrary to the original C++ code. The shared library was recompiled with the same optimization flags and the Java code now ran slightly faster than the pure C++ code, still providing correct results (see figure 2.3 and figure 2.4).

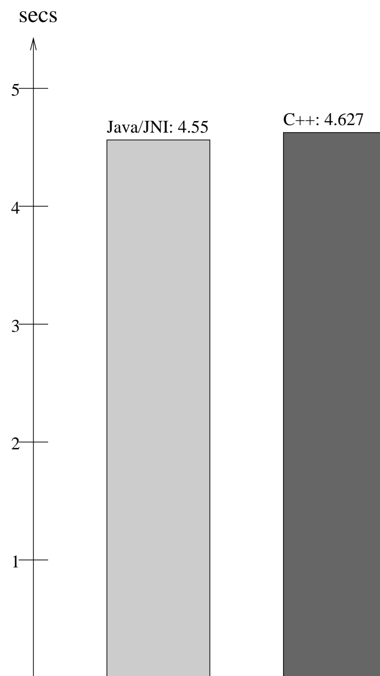


Figure 2.3: C++ and Java average computation times per image.

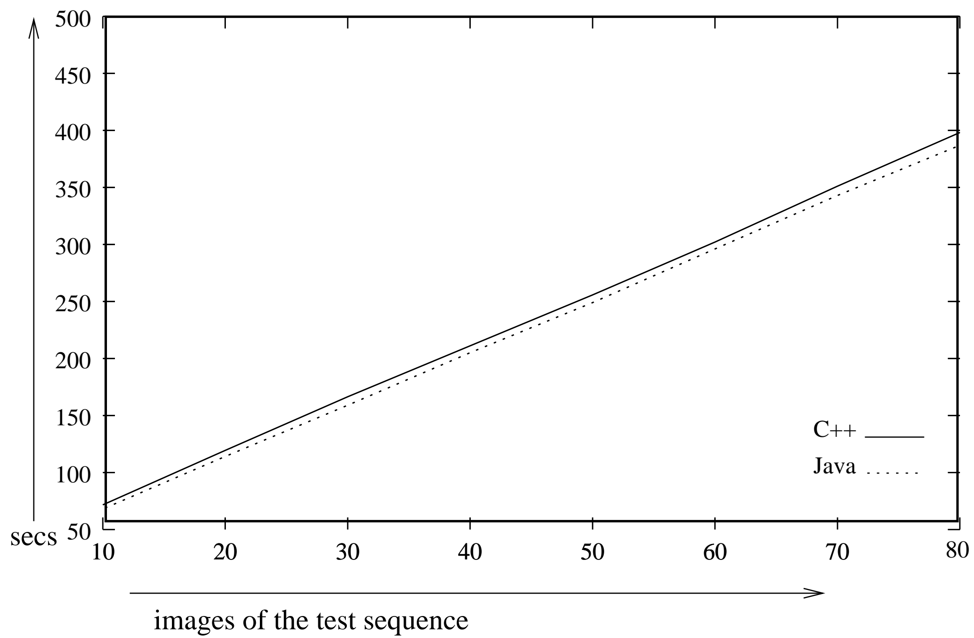


Figure 2.4: C++ and Java average computation times for the realignment of sequences of images.

Chapter 3

The CORBA-based Client/Server Realignment Application

The last step, before the integration of the load balancer, was to transform the Java realignment implementation into a Client/Server application using CORBA, since the load balancer, as well as the monitoring tool, are based on this distributed object-oriented system.
Therefore, CORBA will be presented in this chapter. As mentioned in *chapter 2* the first steps for the division into client and server components have already been made, by means of identifying the service, the server will offer and by providing autonomous classes for the two basic components.

The server architecture, some interesting design aspects, and the definition of the IDL interface will be discussed in this chapter. At the end of the chapter the performance of the CORBA-based program will be evaluated and compared with the C++ and Java versions.

3.1 Technical Background: CORBA

The Common Object Request Broker Architecture (CORBA, see also [6] and [7]) is a standard developed and specified by the Object Management Group (OMG), a group which has been founded by a number of leading companies, such as Hewlett Packard, Sun Microsystems, and American Airlines. CORBA is the effort to establish a common view of the distributed object-oriented programming paradigm, which combines two important concepts of computer science:

- Object-orientation has proved to be a solid concept for developing large applications in a well-structured and efficient manner.
- Distributed computing is a trend that grows stronger in an ever more connected world. Object distribution provides fine granularity for distributing program components among the nodes of a network, i.e. to balance load and

obtain maximum performance out of the existing computing power. Other important features of distributed computing environments are transparency (location transparency, concurrency transparency etc.) and interoperability (interoperability of different platforms and languages).

The CORBA standard is a specification of the services such a system should provide. Implementations are available by various vendors. Two different CORBA implementations were used for this thesis.

1. The JacORB implementation (see [8]) for the application and the load balancer, since the load balancer is based on a modified JacORB version.
2. The ORBacus implementation (see [9]), because the monitoring system and the associated tools are based on it.

The fact that there existed two different CORBA implementations, that finally had to work together, offered the opportunity to investigate interoperability issues concerning CORBA as well.

3.1.1 Object Management Architecture

The Object Management Architecture describes the basic components of the distributed object world, as well as the basic services it should provide. A more detailed description of some of its main components will be given in the following sections.

The distributed world, as specified by the OMG, is made up by the following components:

- The actual application, represented by a set of application objects.
- The Object Request Broker (ORB). It is the most important component, since all CORBA method calls are channeled transparently through the ORB. Thus, the ORB provides the basic communication mechanisms, has to ensure interoperability between heterogeneous platforms, and hides those aspects from the application programmer.
- The object service provides basic low level functions for distributed object-oriented systems, such as methods that allow the client to detect objects of a certain type and a life cycle management, which provides services for moving, copying, creating, and deleting objects within a distributed system. Furthermore, there exists an CORBA event service and a naming service for publishing and subscribing to objects.
- The common facilities offer high level services, such as document management, electronic mail etc.

Those components interact as depicted in figure 3.1.

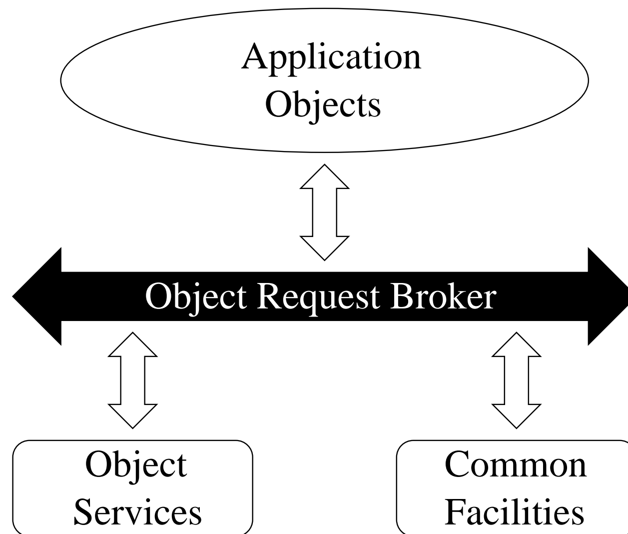


Figure 3.1: The Object Management Architecture, as defined by the OMG. All components of the distributed object world communicate through the Object Request Broker.

3.1.2 CORBA Object Model

Objects in CORBA are defined as identifiable entities with a state and a set of operations, which can be applied on them. What is special about distributed objects is, that they have to use communication mechanisms (the Internet Inter-ORB Protocol IIOP defined by the OMG), in order to perform (remote) method invocations on distant objects. When an object invokes a method offered by a remote object (an object, which is not located on the same host and/or not in the same memory space) it has to be able to detect and identify the remote object and to send and receive data. This explains the need for unique object identifiers called Interoperable Object References (IORs), which are generated for each CORBA object.

Furthermore, the intervention of communication mechanisms has to be hidden, i.e. a remote method invocation should look like a local method invocation (location transparency). Therefore, a proxy object, which is the local representative of the remote object and performs the communication duties, is automatically generated by the IDL compiler (see 3.1.4). Thus, an object call in CORBA consists of a method invocation on the proxy object, which is also called stub. On the server-side the incoming request for a method invocation is handled by the automatically generated server skeleton and the POA (see next section).

3.1.3 Portable Object Adapter

The Portable Object Adapter (POA) is an important component of the ORB, especially within the context of the automatic load balancer (see 4.1.1). The object adapter is the layer between CORBA objects and programming language objects on the *server-side* of the program. It is responsible for servant object creation/deletion and for forwarding method invocation requests to the servant object.

It performs the following tasks:

- Generation of object references.
- Activation/Deactivation of objects.
- Location/Start of servants.
- Dispatch operations.

The desired server policies, such as the threading policy or the request processing policy, can be specified in the POA.

3.1.4 The Interface Definition Language (IDL)

As already mentioned, another key feature of CORBA is, that it provides a framework for assuring interoperability in heterogeneous environments. That means, that for example a Java object running on a PC is able to request a service from a C++ object running on a Sun work station. This type of interoperability is offered by the Interface Definition Language (IDL), which provides the framework for platform- and language-independent remote method invocations based on IIOP.

IDL has a C-like syntax for specifying CORBA methods, events, and complex data types. Each CORBA implementation provides an IDL compiler, which compiles the `.idl` file and automatically generates a series of files, which provide the client stubs and server skeletons for the actual implementation of the specified methods.

An inconvenient when working with two different CORBA versions, such as JacORB and ORBacus, is that there are no standard names for the generated stub and skeleton files, which can lead to confusion. Most providers offer an idl2C++ and/or an idl2Java compiler. Thus, if a Java programmer wants to use a service provided by an C++ object on a server he just has to compile the IDL file specifying the service provided by the C++ object, using an idl2Java compiler and implement the client side of the code. The overall design process of a CORBA application is outlined in figure 3.2.

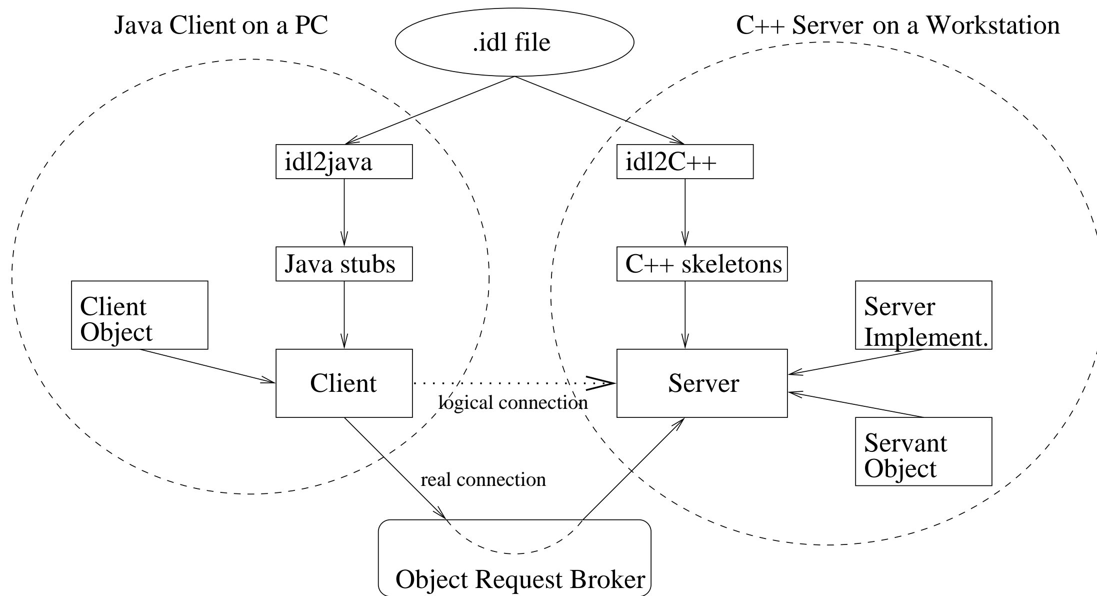


Figure 3.2: The development process of a CORBA Client/Server application for a heterogeneous environment.

3.2 Design of the Client/Server Architecture

This section describes the division of the sequential Java program into its client and server counterparts, the basic server design, and the definition of the IDL interface for the realignment application.

3.2.1 Splitting the Java Program into Client and Server Components

At first it has to be specified, which service should be provided by the server. The service provided by the server in our case should be the most cost expensive part of the application in terms of execution time, since that is the part which will be automatically parallelized by the integration of the automatic load balancer later on.

It has already been stated, that the `compute()` method, which actually calculates the realignment matrix of an image, is the service to be provided by the server, since it is situated within the inner loop of the program and can easily be parallelized. The only precondition for its execution is the availability of the reference data.

The Java program has already been structured with respect to the forthcoming division into client and server components during the previous step. Therefore, the transition to a Client/Server program is relatively easy, because only minor

changes are necessary, in order to build a client and a server out of the `Realign` and the `Compute` classes respectively.

Before defining the IDL interface and encapsulating the `compute()` method into a CORBA method, some basic considerations have to be made concerning the particular design aspects of a server.

The requests for computation jobs a server receives can come from more than one client. The problem with the particular application is that for each calculation of the transformation matrix the reference data for the present image must be available. A straight forward approach would be to pass the correct reference data to the compute function at each call. This is simple, but not very efficient, since the same data would have to be sent along with every compute request. The size of the reference data set is big enough to cause a significant performance decrease.

Furthermore, image sequences are relatively long in practice (usually more than 100 images per sequence). Thus, the overhead for sending the reference data at each request becomes significant. Therefore, the server initially maintained a simple cache (for a thorough discussion of different cache architectures see 4.2.4) for storing the reference data of the client's sequence.

This approach leads to a modified `compute()` function for the CORBA application, which is represented by the following pseudo-code:

```
void corbaCompute(Image image, String clientID, int sequenceID)
{
   if(!ReferenceDataAvailable(clientID, sequenceID))
      {
          org.omg.CORBA.object client = getClientIOR(clientID);
          ReferenceData referenceData =
             client.corbaGetReferenceData(sequenceID);
      }

   compute(image, referenceData);
}
```

The problem was, that the server has to know, which client is requesting the service and for which sequence, in order to be able to issue the request for the specific reference data to the correct client or to look it up in the cache, i.e. a mechanism is needed to uniquely identify clients. Therefore, a unique client identifier was constructed, which is built out of the client's host name concatenated with its Process Identification Number (PID). Alternatively the client's IOR address could be used.

This identifier becomes an additional parameter of the `corbaCompute()` call, in order to enable the server to know to whom he is providing the service. The method `corbaGetReferenceData()` for the acquisition of the reference data is

implemented and provided on the client-side of the system.
The unique client identifier is used for the registration of the client's reference data service at the CORBA naming service. This offers a dual view of the system, because the client may also be considered as reference data server.

Another aspect that imposes modifications on the server-side is, that the server handles requests sequentially. Therefore a *ping-pong* effect, which might occur, when more than one client attaches to the server, has to be avoided. Let there be two clients, which issue alternating requests for the compute service. In this case the correct reference data has to be resent each time. To solve this a reference data buffer was added to the server using a FIFO strategy for assigning new slots.
In more detail the introduction of the cache and the caching strategy (FIFO) lead to some additional administrative components, such as a lookup function for the reference data and a function implementing the FIFO strategy. At this stage these functionalities were implemented on the C++ side of the program, and provided to the Java object, handling the compute request, as *native* methods.

For the case, that multiple clients are attached to the server the C++ implementation offers a performance advantage. When the reference data buffer is handled and maintained in the C++ code, it has only to be passed once through JNI to the *native* code. Whereas, if the buffer is handled by the Java object, the reference data has to be passed at each invocation of the `compute()` method. According to criterion 2 established in 2.4 this should be avoided.

The server has three layers:

1. The CORBA Portable Object Adapter (POA), which unpacks the data and forwards the request, using single-thread mode, to the Java object providing the realignment service.

2. The Java object calls the respective native methods for cache handling and computing the realignment matrix.

3. Cache handling as well as the actual computation are performed in the C++ component of the server.

The control flow and structure of the server are depicted in figure 3.3 for the case, where reference data is already available and in figure 3.4 for the case of a cache miss, i.e. the server has to fetch the data from the client before completion of the compute call.

An alternative to the Client/Server model would be to add a third component to the system, a so-called pool that registers the compute requests and stores the reference data of all clients. Furthermore, the pool would assign identifiers for each computation job and reference data set.

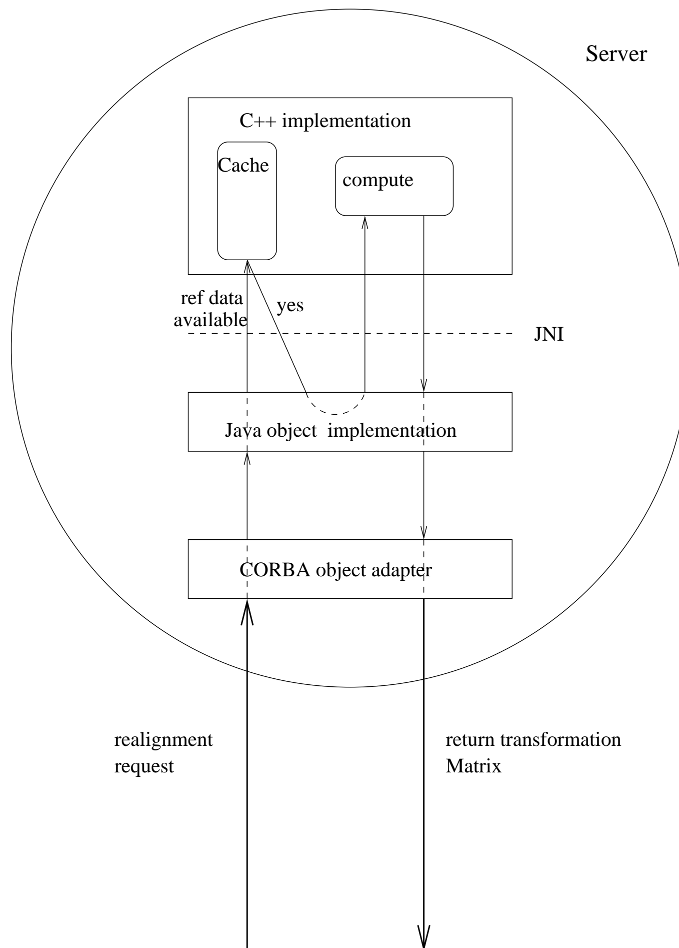


Figure 3.3: Outline of the layers a compute request traverses within the server for the case that the required reference data is already cached.

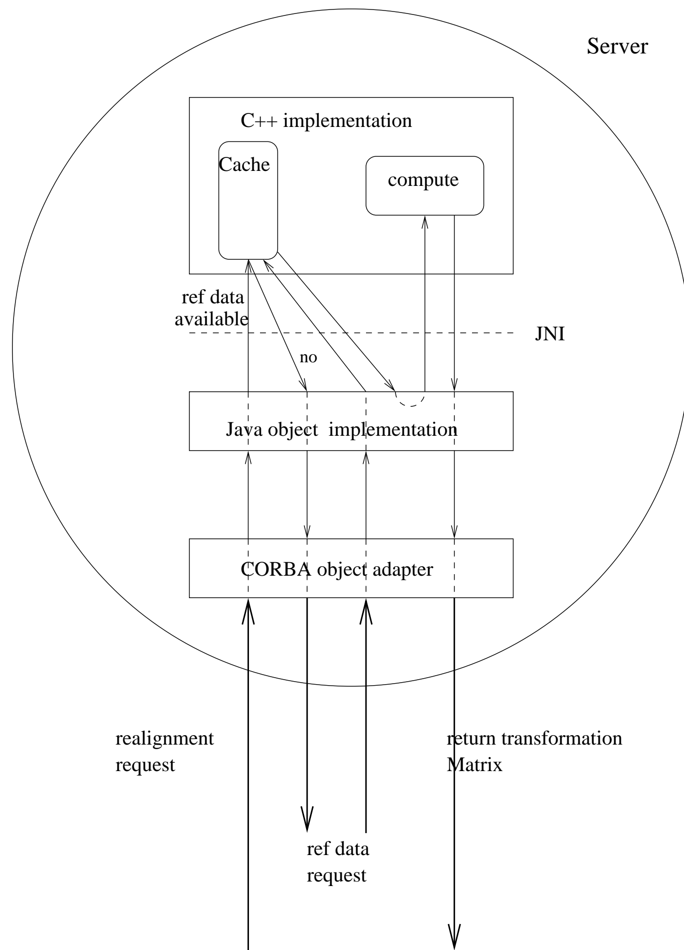


Figure 3.4: Outline of the layers a compute request traverses within the server for the case of a cache miss. An additional *inner* call is made for obtaining the reference data from the client, before completion of the compute call.

For means of simplicity the pure Client/Server approach was chosen. A pool would be an additional centralized component and would therefore lead to a decrease of reliability and an increase of complexity. An implementation of the pool model based on CORBA and the C++ implementation can be found in [3].

3.2.2 Definition of the IDL Interface

Now one can define the IDL interface for the realignment application. As mentioned above two methods have to be defined in the interface, one for the computation of the realignment matrix, which is invoked by the client and one for the acquisition of the reference data of the present image, which is called by the server when in the case of a cache miss.

Definition of the module name:

```
module Realign
{
```

Definition of a data-type for a sequence of doubles.

```
  typedef sequence<double> SeqDouble;
```

Specification of the service provided by the client. It reads (`in`) the client identifier for verification purposes, i.e. to check if the reference data is requested from the correct client and the sequence number to identify the reference data, since one client may wish to realign multiple sequences of images. It writes (`out`) the reference data and the numerical parameters, which can be differing for each client.

The parameters `fwhm` and `hold` are parameters specified in the parameter string of the realignment program (see 2.2.1 and [3]), the remaining variables form the reference data set.

```
  interface C_Realign
  {
      void corbaGetReferenceData(in string clientID,
                                 in long sequenceID,
                                 out double fwhm,
                                 out long hold,
                                 out SeqDouble mat_0,
                                 out SeqDouble A0,
                                 out SeqDouble x1,
                                 out SeqDouble x2,
                                 out SeqDouble x3,
                                 out SeqDouble b);
  };
```

Specification of the service provided by the server. It reads (`in`) the image name, the client identifier, the sequence number, the image data, three image-specific parameters, and it reads and writes (`inout`) the transformation matrix which has been initialized with 0.

```
  interface C_Compute
  {
      void corbaCompute(in string name, in string clientID,
                        in long sequenceID, in SeqDouble img_data,
                        in long dim_x, in long dim_y,
                        in long dim_z, inout SeqDouble img_mat);
  };
};
```

3.3 Verification and Performance Evaluation

The verification procedure remains the same as described in the previous chapter.

An interesting performance issue at this stage, is the evaluation of the overhead produced by CORBA and the introduction of the Client/Server architecture (i.e. the communication overhead). Depending on whether the server runs on the local node, i.e. the client's node, or on a remote one, the overhead produced by the CORBA version lies between 25% to 45%. The additional communication overhead for a remote server is between 15% to 20% (see figure 3.5 and figure 3.6). The performance of a server with cache and without cache is also depicted, in order to justify the introduction of the server cache. The results show, that a cacheless server leads to a 10% increase of the average realignment time per image. This is the time needed for transferring the reference data at each `corbaCompute()` method invocation.

Three different CORBA configurations were considered (see figure 3.5):

1. Configuration: A server with cache running on the same node as the client.
2. Configuration: A server with cache running on a different node.
3. Configuration: A server without cache running on a different node. In this case the communication overhead, caused by the reference data transfer at each call, becomes more evident.

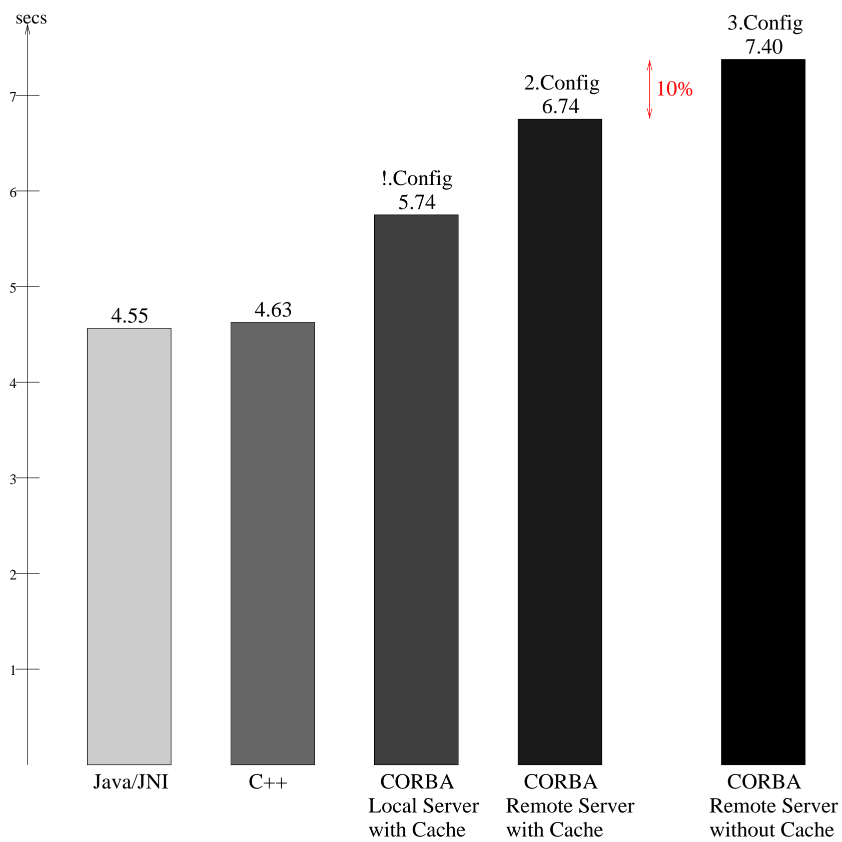


Figure 3.5: Average computation times per image for the Java, C++ implementations, and different CORBA configurations.

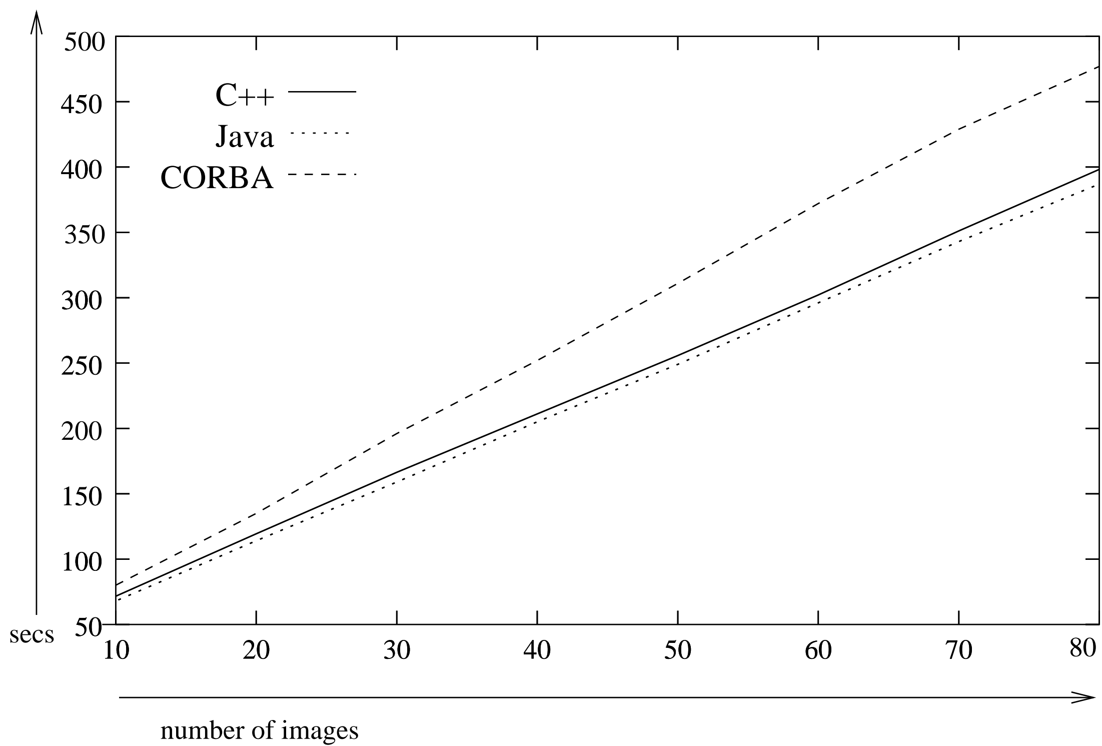


Figure 3.6: Average computation times for sequences of different length. The running time of the realignment algorithm is O(n), where n is the number of images.

Chapter 4

Integration of the Load Balancer

This chapter gives an introduction to load management, presents the automatic load balancer, and describes its integration into the existing CORBA program. Furthermore, the different versions of the realignment application, which have been designed to test specific properties of the load balancer are described. Finally, the results of these test are presented and the load balancer's performance is evaluated.

4.1 Technical Background: Load Management

The high complexity, as induced by distributed object-oriented environments, such as CORBA, leads to performance penalties due to load imbalance. On the other side CORBA offers a high distribution potential, because the object has now become the smallest distributable unit. The increase of complexity paired with a finer distribution granularity motivates the need to concentrate efforts on the design and improvement of load management in such systems.

Load management mechanisms can be classified, according to the layer at which they are implemented. They can be integrated directly into the *application*, they can be handled by the *runtime system* or a separate *load management service* can be offered.
Integrating load management into the application is time and cost intensive and does not provide a general solution for the load distribution problem. It only provides load balancement for the specific application and does not take into account CORBA objects from other applications, which might also produce significant load.
Integrating load management mechanisms into the runtime system, such as the operating system or the middleware, hides the distribution mechanism from the application programmer.
The third possibility, a separate service, combines both approaches, since the ap-

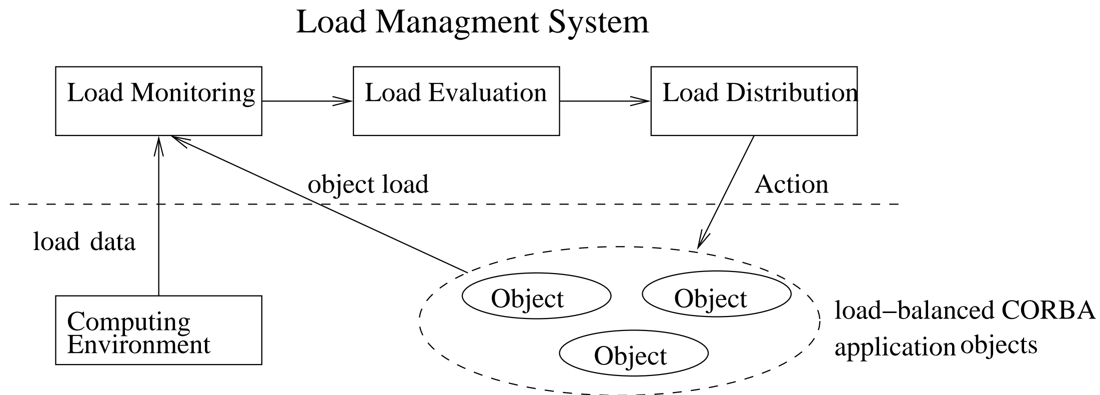


Figure 4.1: Basic Components of a Load Management System.

plication programmer is aware that he is using a load management mechanism, but does not have to bother with details.

In general a load distribution system has to provide three different services:

1. Initial Placement: Selection of the most suitable host, in terms of load and efficiency, on which a new servant object should be started.

2. Migration: Moving an object from the present host to a less-loaded one for accelerating its execution.

3. Replication: Creating a copy of an object on a remote host (a copy of an object on a remote host is called replica) for accelerating the execution time by redirecting a part of the requests to the new replica.

4.1.1 The Automatic Load Balancer

The automatic load balancer (see [13]) can be considered as an extension of the CORBA standard and provides a load distribution *service*, as described in the above section. CORBA objects interact with the middleware through the POA and the ORB (see 3.1). Therefore, the load management infrastructure is integrated into those two basic CORBA components.

The following three basic components of a load management service system have to be integrated into the CORBA middleware (see figure 4.1):

1. *Load Monitoring*
   Data about the load levels in the distributed system has to be collected by the load balancer. This includes load information on the distributed objects, as well as on the typical resources of a distributed environment

such as CPU load or network load. The necessary resource information is collected via the Simple Network Management Protocol (SNMP, see [19]), which is an adequate and flexible means for gathering the required data within a heterogeneous system. The object-related information has to be acquired by the middleware itself. The ORB and POA have been modified for extracting relevant data, such as the object request rate, the request waiting time, the data volume of requests etc.

2. *Load Distribution*
   The CORBA standard is extended in several ways, mainly concerning object creation, initial placement, and request forwarding (when target objects have been migrated or replicated and received a new IOR address). This led to the introduction of a `ServantFactory` for creation and deletion of servant objects, which has to be provided by the programmer. Furthermore, initial placement and starting of servant processes has to be handled by an additional component, the `GenericFactory` which provides the `create_object()` and `delete_object()` services to the application programmer. Initial placement and other associated actions are performed transparently by this component. The last component of interest to the application programmer is an extension of the `ServantFactory`, the `PersistentServantFactory` in which the methods for servant state extraction and insertion have to be implemented.

3. *Load Evaluation*
   The load evaluator consumes the information provided by the load monitoring component and makes decisions about initial placement, migrations, and replications. Furthermore, it is responsible for redirecting requests in case of migration or replication.

   This work is performed using the mechanisms provided by the load distribution component. Different load balancing strategies can be implemented in this component.

4.2 Integration of the Load Balancer into the CORBA Program

This section deals with the integration of the load balancer into the realignment application and the different execution modes it offers. The *rationale* for the introduction of each execution modus as well as important *implementation* issues will be presented and discussed in the corresponding sections.
In order to give an overview of the different configurations and also to provide a small user's manual the command options for the client and server are explained.

Client Command Options:

```
J_Realign [ -tc | -tl ] [ -so | -do ] [ -NoClientCache ]
          [-help] ImageParams
```

where `ImageParams` is the parameter string of the realignment application as described in 2.2.1.

The meaning of the command options is given in the following tables:

`-tc`	Multi-threaded client: Thread model with parallel access to redundant data.
`-tl`	Multi-threaded client: Thread model with synchronized access to single data.

If neither `-tc` nor `-tl` is specified, multi-threading is disabled.

`-so`	New client attaches to an already existing servant object, if available.
`-do`	New client creates its own, private servant object.

If neither `-so` nor `-do` is specified, the client runs in `-so` mode.

`-NoClientCache`	The client does not use a cache to store its actual reference data.
`-help`	Displays a help message similar to this tables.

Server Properties:
The server properties are specified in a file called `Realign.imr`. Apart from the standard load balancer properties, the cache model the server uses can be specified by setting the variable `CacheStrategy` either to `C++Cache`, to `JavaCache` or `NoCache`. The default setting is `C++Cache`. The setting is overridden when the standard variable `Stateful` is set to `true` and `CacheStrategy` is automatically set to `JavaCache` (for details see section 4.2.4).

4.2.1 Basic Version

The basic adaptation of the CORBA realignment application to the requirements imposed by the load balancer was straight forward. Thus, a basic version of the load-balanced realignment application was soon available.
The server had to be slightly redesigned and the client had to use the methods provided by the `GenericFactory` implementation in order to create a new servant object. Furthermore, the `PersistantServantFactory` had to be adapted to the application, initially handling stateless [1] replications and migrations only.

[1] Stateless denotes an object replication or migration where the object state is not transferred

The reference data service `corbaGetReferenceData()`, provided by the client, is still handled via the CORBA naming service and is not load-balanced, since only few such requests are expected when using a server with cache. A reference data request might occur at the first image of a new image sequence, after a stateless replication or migration, or at intervals depending on the size of the reference data buffer, if the number of clients is greater than the buffer size in single object mode, or the C++ cache is used in multiple object mode. Furthermore, a reference data request consists mainly of transferring large data amounts and not of computations (Except when a cache-less client is used, but this option is only available for demonstrating the usefulness of the persistency mechanism).

4.2.2 Single and Multiple Object Mode

There are two alternatives for a client to detect and access a servant object at start-up. It can either check if such an object is already available and attach to it or create it, if it is not available (single mode). On the other hand each client can create its own, new object irrespectively of what other clients are doing (multiple mode).

Rationale:
Those two configurations are of interest for the scenario where many single-threaded clients issue compute requests. With a sufficient number of nodes available, each client in single object mode should at the end of an initial distribution phase be attached to a distinct replica running on a distinct host. In multiple object mode the replication has already been performed manually at client start up and the objects have to be distributed among the machines using migrations (only if initial placement put several objects on the same host). Therefore, it was important to test and compare the behavior of the load balancer for both modes under equivalent circumstances, i.e. automatic and manual replication.

Implementation:
The implementation of multiple object mode is straight forward since every client simply creates a new servant object by calling the appropriate `GenericFactory` method. In single object mode the client reads the IOR of the existing object from a file (for the sake of generality in a file on a www-server) or writes the IOR of the newly created object to the file, if it previously did not exist.

4.2.3 Multi-Threaded Client

Among the application scenarios considered was the case, that there might be only one single *heavy* client performing computation requests. At present there would be no means of taking advantage of the features the load balancer provides,

without introducing an additional mechanism on the client side, that somehow simulates multiple clients.

Rationale:
The option to start a multi-threaded client was introduced in order to permit the parallelization of a single client. The key feature of this extension is that the *distribution granularity* offered by the *client* is *improved dramatically*, because the different threads might get forwarded their realignment requests to distinct replicas by the load balancer. This can only be achieved however, if each thread maintains its *private proxy object*.

This offers the possibility to take full advantage of the available computation power on the one hand, and to test the automatic parallelization the load balancer provides in this specific configuration on the other hand.

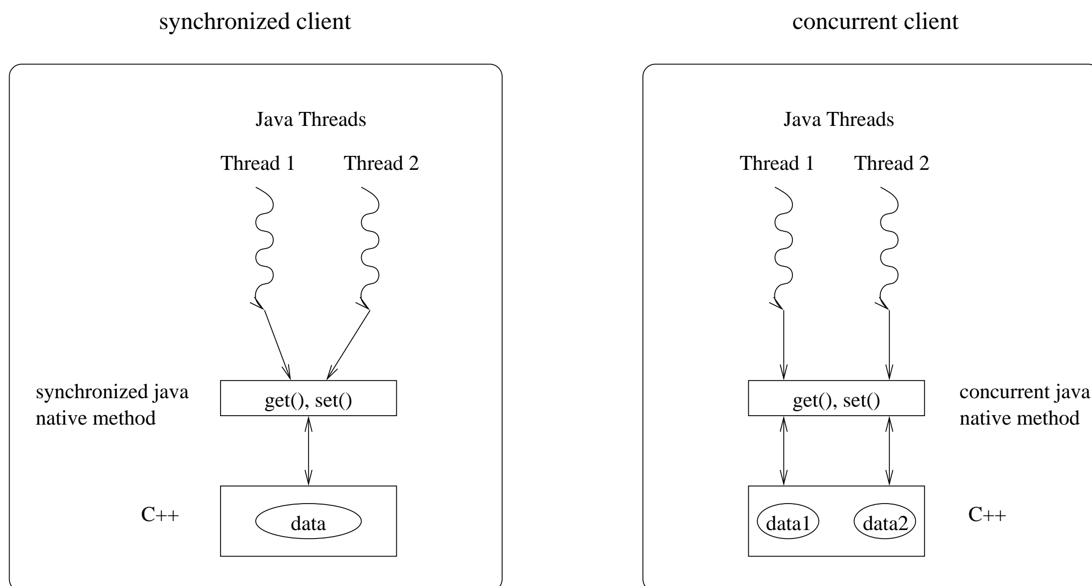


Figure 4.2: Data access structure of the synchronized and concurrent multi-threaded client.

Implementation:
Thread synchronization concerning simultaneous read/write data access had to be considered, because in the sequential model only one image at a time is loaded into memory. Two thread models were implemented:

1. A concurrent model with private memory space for each thread.
2. A synchronized model where threads obtain exclusive data access through a lock.

In the synchronized version the client threads read and write the image and reference data via a Java `synchronized` method, which ensures that only one thread at a time can use these methods. The concurrent version maintains an array of private data segments for each thread in the C++ part of the client code (see figure 4.2). This is a tradeoff between time and memory. The methods to `get()` and `set()` image and reference data have however a neglectible impact on the overall computation time, such that the synchronized model does not induce any measurable performance penalties. Therefore, the synchronized version should be preferred for the sake of reduced complexity and memory consumption.

The images due for realignment are split equally among the threads, i.e. each thread is initially assigned a constant number of images for realignment. One could think of a dynamic image distribution strategy, because during the initial phase of load balancement, when replicas are created and distributed among the hosts, significant differences in execution speed occur. This option was not implemented due to lack of time and because multi-threading has been added for proof-of-concept reasons only. Different image assignment policies would be of interest within the context of work associated with application scenarios for the load-balanced realignment application. In this case absolute performance and speed become important, in contrary to relative speed which is being considered here.

A solution is however proposed at this point. The aim of a dynamic image distribution strategy is to keep all threads and associated replicas busy until the end of the computation, i.e. to assign more realignment jobs to a fast thread.

The client maintains an array of image numbers for the actual image sequence and an index to the next image due for realignment. When a client has finished its computation it requests a new image number via a `synchronized` method `getNext()` which returns the next image number and increments the index (see figure 4.3). One might argue that the `getNext()` method might become a bottleneck of a system with a great number of threads. The time however for reading an array entry and incrementing an index is neglectible and furthermore it is an asynchronous system, i.e. the probability is low that many threads will be requesting a new image number at the same time and therefore be blocked.

Another interesting aspect is the number of threads to be created, i.e. if this should be done statically, i.e. by creating a fix number of threads at initialization depending on the number of images, or dynamically by observing the present threads behavior, i.e. if the computation time per image is below a certain threshold for all existing threads. Due to lack of time only the first approach was implemented.

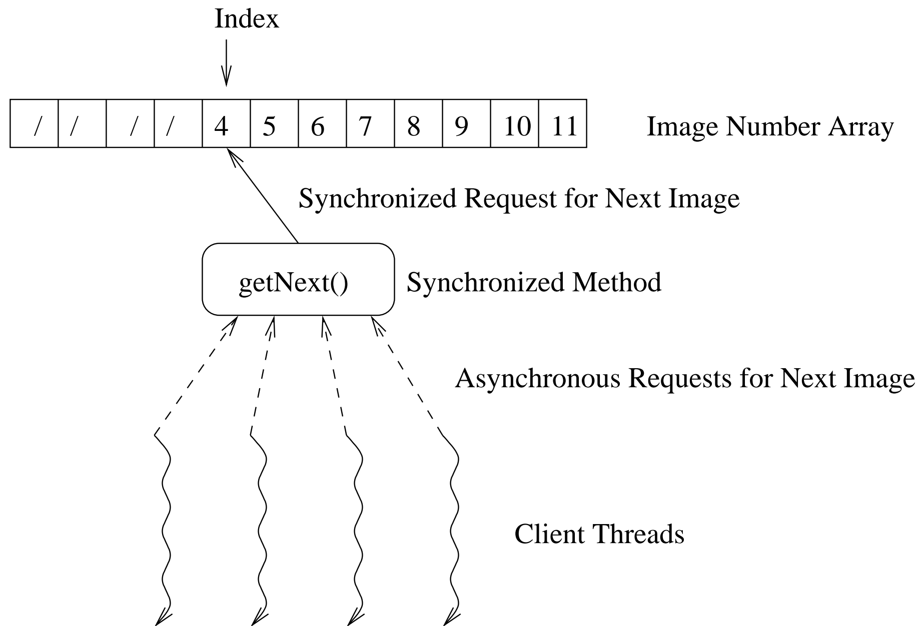


Figure 4.3: Outline of the possible design for a dynamic thread model.

4.2.4 Cache Architectures

As already mentioned the server provides three different cache models.

1. A C++ cache which is controlled by the server and handled by *native* method calls.

2. A Java cache which is integrated into the servant object, i.e. each object maintains its own private cache.

3. The server provides no caching at all.

Rationale:
Apart from the performance considerations which have already been discussed a cache architecture is of interest at this stage, in order to test the *persistent* replication and migration mechanism the load balancer provides. Thus, the state associated with each object due for *stateful*[2] replication or migration has to be available. An appropriate interface for extracting and setting the state is required. In the case of the realignment application the state of the object is its reference data buffer at the time the migration/replication is initiated.

[2]Stateful denotes an object replication or migration where the object state is also transferred and reestablished

Implementation:
For the above reasons a Java cache version was introduced since cache handling has now to be conducted independently by each servant object. A separate class `J_ReferenceImageCache` was introduced which manages the cache.

4.2.5 Persistent Migration/Replication

The automatic load balancer offers the infrastructure to migrate and replicate *persistent* objects. In cache-based systems such as the realignment application this approach offers some advantages.

Rationale:
The ability to maintain and transfer the state of an object, when migrating or replicating it, saves additional CORBA calls for the restoration of its cache. Thus, if persistent migration/replication is implemented efficiently, i.e. the overhead produced by transferring the state is small, compared to the overhead for cache restoration, this leads to an performance advantage over the cache-less system.
The interesting case within this context is a cache-less client, i.e. the client does not hold the reference data of the current image sequence in a buffer and therefore has to recalculate it each time a cache miss occurs on the server-side. The average time for the computation of the reference data is 4 to 5 times higher than that for an image realignment. Therefore, a stateless migration or replication obliges the clients attached to the new servant object to recalculate their reference data and may lead to serious performance penalties, i.e. cache restoration is expensive.

Implementation:
An interface for obtaining and setting the state of a servant object has to be provided and integrated into the `PersistantServantFactory` class handling persistent migration and replication. This class is an integral component of each load-balanced application wanting to make use of the persistency mechanism.

The state for the realignment application is defined as the contents of the `J_ReferenceImageCache` class which is an attribute of the `Compute` class and can therefore be easily obtained. The state of a replicated or migrated object is set by the introduction of a second constructor in `Compute` which takes an `J_ReferenceImageCache` object as argument.
It has to be mentioned that all classes belonging to the hierarchy forming the state have to include `implements java.io.Serializable` in their class definition since the object data has to be serialized in order to be packed and sent via CORBA.

Unfortunately the size of the servant object's state is relatively great (2 Mb) and this lead to serious performance problems. Therefore, three different methods for handling the size have been considered:

1. The standard method provided by the persistent factory implementation.

2. Additional data compression within the `PersistantServantFactory` class using `java.util.zip` was performed.

3. Writing and reading data into and from a file of the distributed file system avoiding the usage of CORBA for state transfer.

The standard method caused an unexpected and unacceptable overhead and delay in the computation, probably due to non-linear behavior of the Java garbage collection and memory allocation mechanisms at high load and memory requirements. Thus, initially an attempt was made to compress and decompress the serialized object data, which lead to a slight improvement, the performance remained still prohibitive however.
As a final solution and in order to obtain results, showing that persistent objects are advantageous in certain cases, the state was not transferred using CORBA communication including packing, unpacking data etc., but using the Network File System (NFS).

The generic persistency mechanism proved to be flexible enough to permit an efficient implementation and evaluation of different methods for transferring the object state.

4.3 Verification and Performance Evaluation

The verification procedure remains the same as in *chapters 2 and 3*, since the output matrices calculated by the load-balanced realignment application have to be compared with those of the original program.

The performance evaluation sections deal with the load balancers performance for different Client/Server configurations.

4.3.1 Manual versus Automatic Replication

Two test runs in *stateless* mode for the same image sequence and with a low network and host load level have been run. A pair of clients has been started in single object mode (Client 1S, Client 2S) and a second pair in multiple object mode (Client 1M, Client 2M).

The requests of Client 1S and Client 2S are assigned to the same object initially, which becomes overloaded. The load balancer reacts to the high object load by replicating the servant object at about the 9th image of the test sequence.

The servant objects created by Client 1M and Client 2M are initially placed on the same host, since they do not start to produce load immediately. Once again the load balancer responds correctly to the high host load and initiates a migration at about the 11th image of the test sequence.

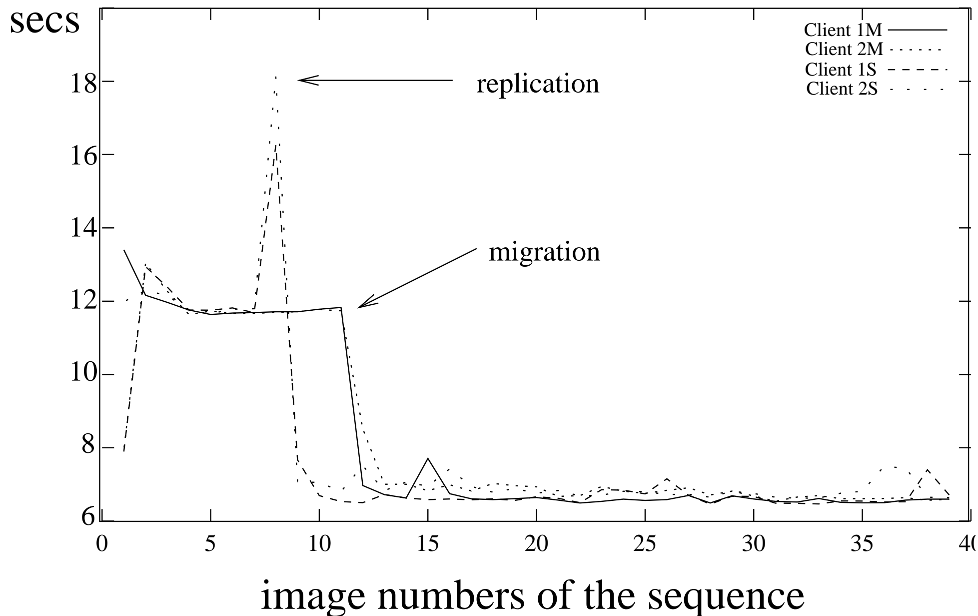


Figure 4.4: Computation times per image for two clients in double object mode and two clients in single object mode.

The comparison shows that manual and automatic replication (see 4.2.2) render nearly identical results. The requests of one client are reassigned to a migrated or replicated servant object on a new host. The only notable difference, which has been confirmed by a series of tests, is that migration is faster than replication, probably due to the augmented overhead associated with object replication.

4.3.2 Multi-Threaded Clients

A client with four threads (Thread 1,..., Thread 4) has been started in *stateless* mode within an unloaded network of four equivalent nodes controlled by the load balancer. As shown in figure 4.5 the load balancer correctly distributes the load created by each of the four threads, i.e. towards the end of the computation each thread issues computation requests to a servant object located on a separate machine. At the first load balancing action the requests of one pair of threads are redirected to a a new replica on a separate machine. At the second and third step the remaining pairs of threads obtain a proper server on a new machine each. Thus, the speedup of the average realignment time per image improves from 1 through 0.5, 0.33 to 0.25. The distribution and speedup obtained at the end is optimal for a four-node network. The jags which occur before each reduction of the computation time per image, i.e. at the replication, represent the overhead caused by the replication.

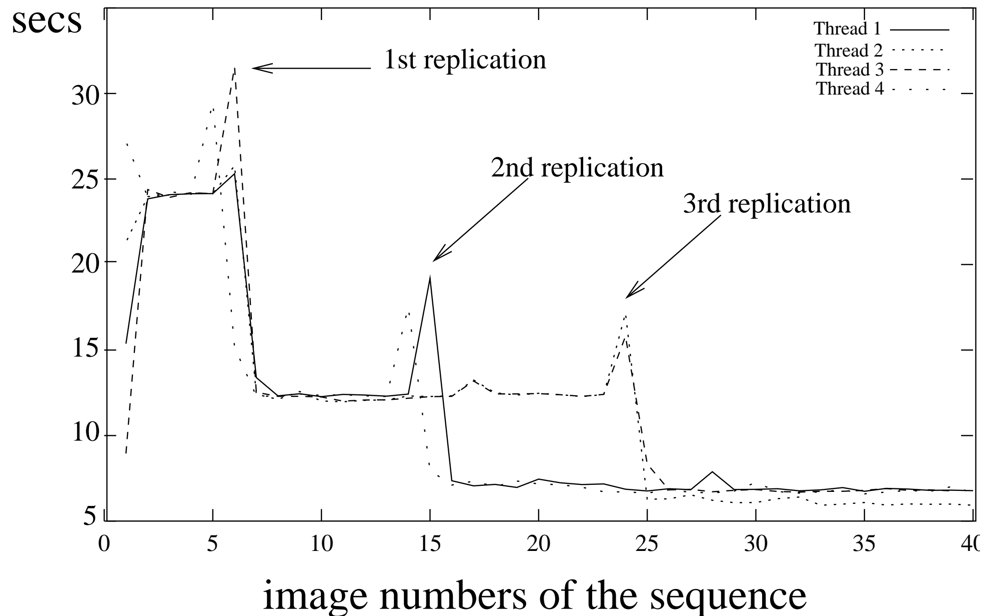


Figure 4.5: Computation times per image for each of the client's four threads.

The way the load balancer assigns the threads to the newly created replica at each step in the given example is depicted in figure 4.6.

4.3.3 Migration and Replication

Two single-threaded clients (Client 1, Client 2) have been started in single object mode in the same network as above. Diagram 4.7 shows the effect of a *stateless* replication ($\approx$ 9th image of sequence) conducted by the load balancer on the computation time per image. Initially a new replica is created on another machine, as expected.

After completion of this replication, load is created on the new machine in order to force a migration. This explains the sudden increase of the average computation time per image around the 12th image of the test sequence (Client 2). The load balancer reacts correctly to this increase in load and performs a *stateless* migration away from the heavily loaded machine.

Figure 4.8 shows that the persistency mechanism is ineffective for the realignment application if CORBA communication is used for transferring the state. Two test sequences, with a pair of clients each, have been run. The first (Client 1C, Client 2C) uses *stateless* migration/replication in a configuration without client cache. The second sequence (Client 1S, Client 2S) uses *stateful* migration/replication via CORBA. It becomes evident, that although the client once

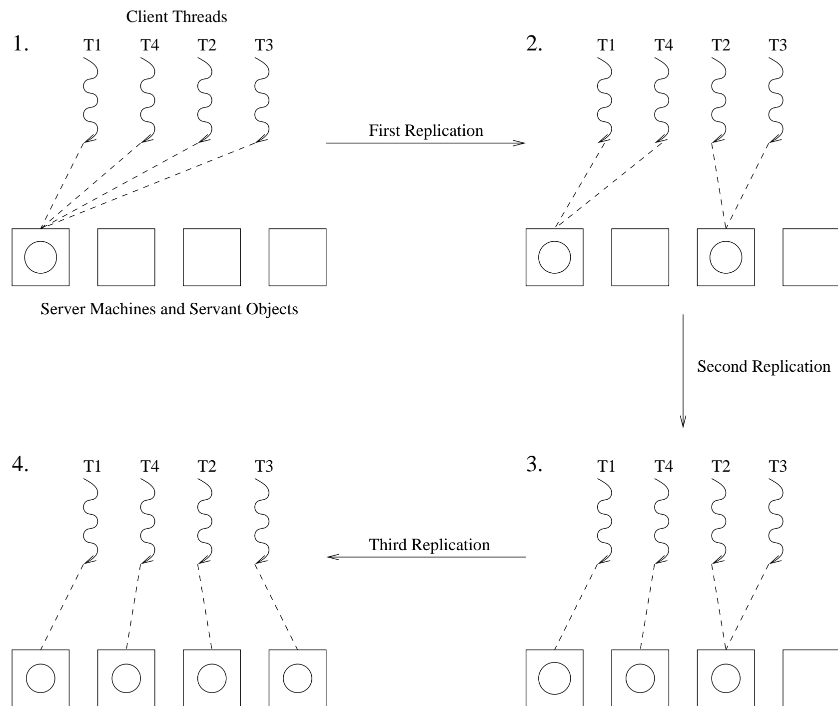


Figure 4.6: At each step a new replica is created and one or two client threads are attached to it.

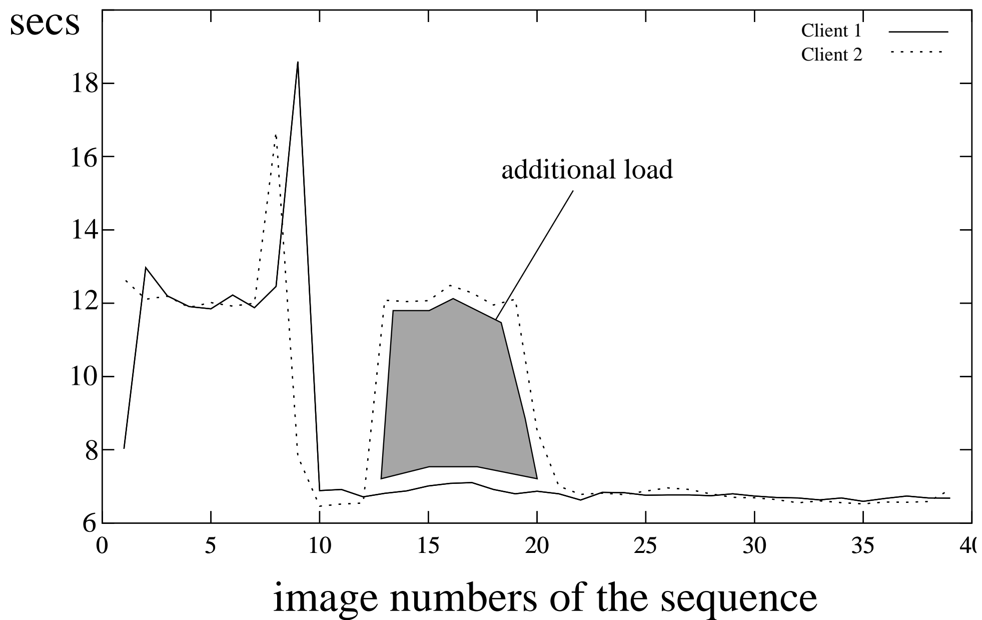


Figure 4.7: Computation times per image for each of the two clients.

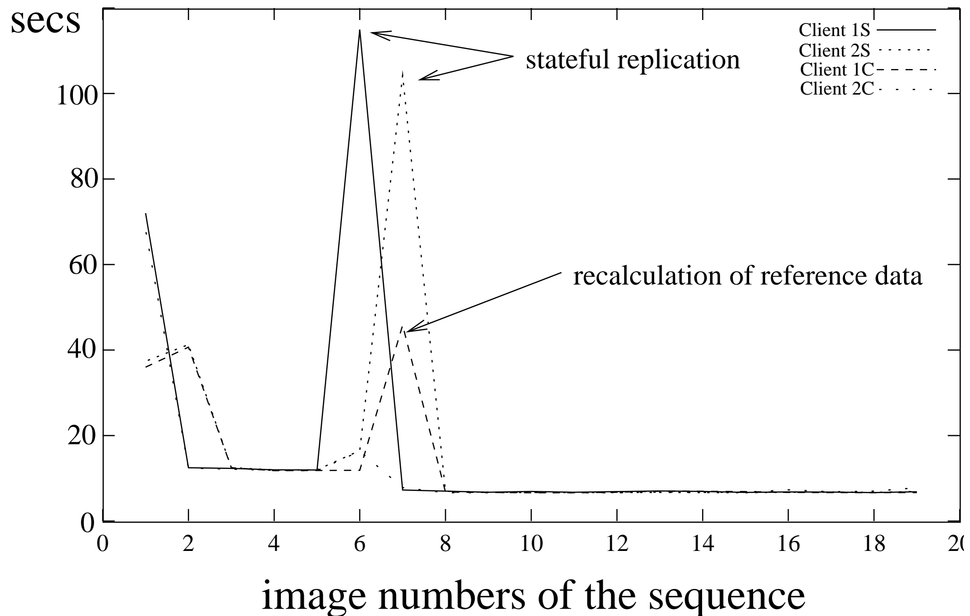


Figure 4.8: Computation times per image for a pair of clients with stateful and stateless replications/migrations.

again has to perform the heavy reference data computation, the *stateful* migration takes about 3 times longer to be performed.

Diagram 4.9 consists once again of the comparison of two series of tests, one with 2 *cache-less* clients (Client 1C, Client 2C using *stateless* migration and replication) and one with 2 *cached* clients (Client1S, Client 2S using *stateful* migration/replication via NFS). The initial computation times for Client 1C and Client 2C are high because both clients have to calculate the reference data set. When a replication for Client 1S and Client 2S is performed the state is transferred to the new replica and an additional call of `corbaGetReferenceData()` is avoided, since the cache is reestablished. The replication performed for the *cache-less* clients takes significantly longer since Client 2C has to recalculate the reference data, because the cache of the new replica is empty.

This demonstrates the advantages in terms of computation time persistent objects offer. Furthermore, it shows that the load balancer is disturbed by the long migration/replication times, since it they are in the order of its load balancing interval. This explains the second migration performed around the twentieth image of the sequence (Client 2C) , although there was no additional load at that time on the host.

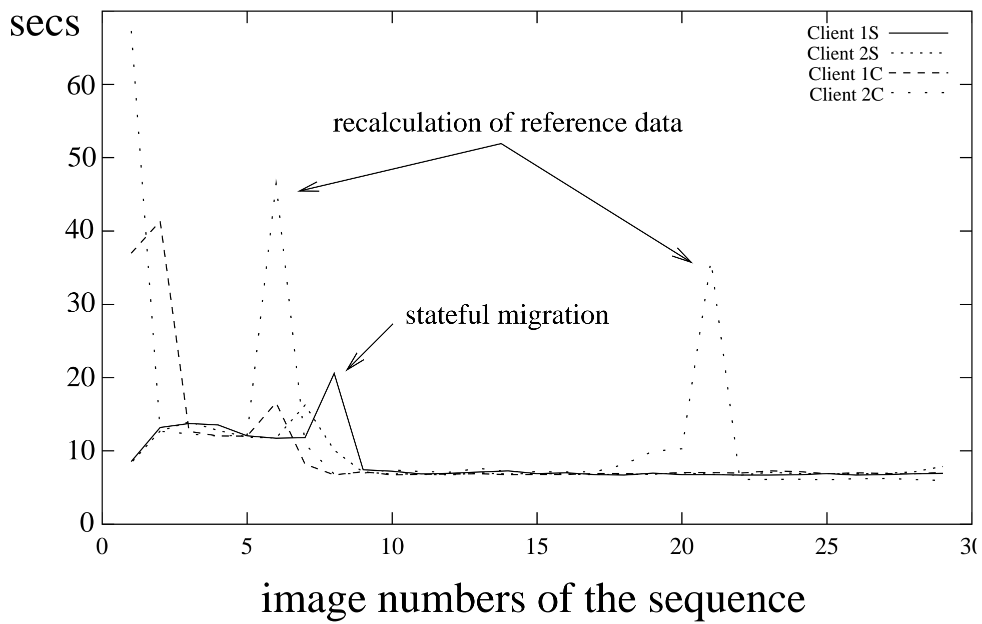


Figure 4.9: Computation times per image for a pair of clients using the persistency mechanism and a pair not using the persistency mechanism.

Chapter 5

MIMO and MiVis

This chapter gives an overview over the two new components added to the system, the MIddleware MOnitoring System (MIMO) and the closely related MiVis system, which is a CORBA visualization tool based on MIMO.

Initially a list of criteria for evaluating and classifying middleware monitoring tools, as well as a brief description of some existing systems is given.

5.1 Technical Background: Middleware Monitoring Systems

5.1.1 General Aspects

Some general aspects of middleware monitoring systems and tools are initially presented:

- *On-line and off-line tools:* Off-line tools collect and store data, e.g. in a data-base, during program execution and evaluate them *after* the test run. On-line tools collect and display the acquired data in real time during the test run, thus offering the possibility for manipulating the application.

- *Active and passive on-line tools:* Active on-line tools do not only visualize the application, but allow to steer it. Passive tools can not influence the application in any way.

- *Instrumentation techniques:* In order to visualize a program, it might be necessary to insert code into the application source. On the other hand instrumentation can be achieved by inserting wrapper functions into the middleware libraries, which remain transparent for the application (see 5.2.2).

- *Flexibility:* Middleware monitoring systems should be easily adaptable to a great variety of distributed object-oriented systems and applications.

- *Scalability:* A general model for describing distributed object-oriented systems should be available, such that large systems can conveniently be monitored.
- *Overhead:* A monitoring system, especially providing on-line services, should produce the least overhead possible, in order not to influence the execution of the application.

5.1.2 Existing Technologies

The approach of DePauw et al.
In this approach the visualization of the behavior of object-oriented systems has been explored. One of the main concerns of this work was scalability, i.e. the possibility to monitor big applications. The model consists of classes, objects, methods, and messages.

Various display types are proposed for visualizing the relations between those basic components, like for example the inter-class call cluster, which displays the message exchange between classes, the frequency of such inter-class calls and the send/receive ratio of a class. This display allows the detection of closely coupled classes. Several other interesting display types have been proposed (for details see [20]), which can be transferred and adapted to distributed object-oriented environments.

The different levels of granularity provided by this approach permit a fine-tuned program analysis.

The CORBA-Assistant
The CORBA-Assistant (see [21]) is a management tool for CORBA applications. It is based on the managed objects approach. A managed object provides additional services for delivering information about its state, events, method invocations etc. This kind of information is sent to a database via a CORBA event channel and is available for static evaluation. The CORBA-Assistant is an off-line tool and the stored data can be used by several visualization tools, to display performance graphs, invocation frequencies etc.

CORBA-Assistant requires extensive instrumentation of the application code and is furthermore limited to usage with the Orbix CORBA implementation, because it makes use of some Orbix-specific functions, which are not part of the CORBA standard.

5.2 Introduction to MIMO

MIMO is an on-line monitoring system for monitoring distributed applications during run time, mainly for debugging and performance analysis purposes (for

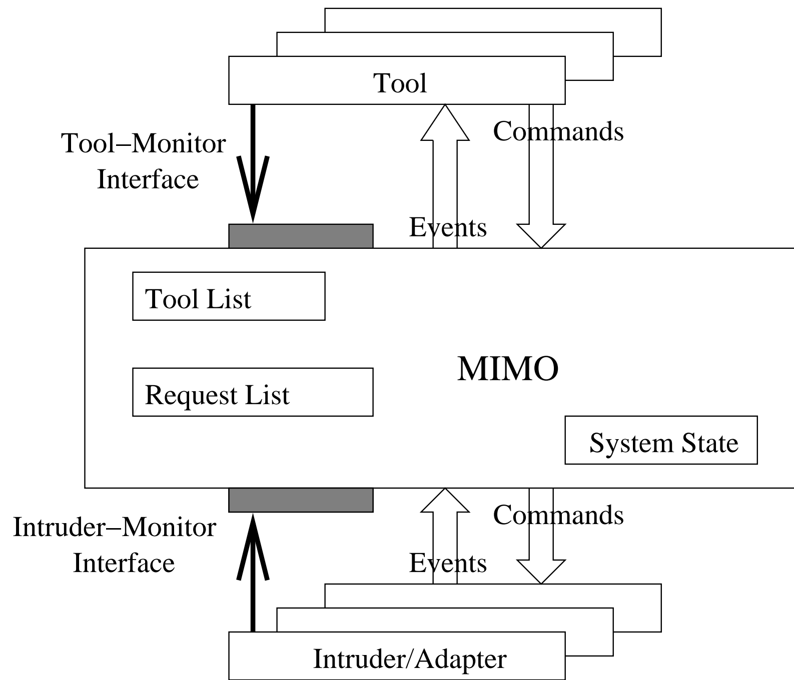


Figure 5.1: MIMO Architecture.

a more detailed description see [15] and [17]). MIMO provides a common on-line monitoring interface based on the CORBA event service (using ORBacus), in order to allow tools to gather and manipulate the required information from the monitored application. An important aspect of the MIMO architecture is the clear separation between tools, monitoring system, and the monitored application. A MIMO-based tool implementation is an independent component, i.e. MIMO only provides the infrastructure for gathering information (see figure 5.1). This approach is advantageous, since it permits the development of various tools based on the same interface.

Distributed applications are often based on the Client/Server paradigm. Thus, there is a need for dealing with heterogeneous platforms and multi-language systems. This type of heterogeneous environments is handled by using CORBA to define the interface and services provided by MIMO. Normally several tools need to observe an application, in order to analyze different aspects at various abstraction levels. Therefore, another important feature of MIMO is tool interoperability, since the application can be monitored by several MIMO tools at the same time without interferences (2nd degree system, see 6.1). Furthermore, MIMO provides an infrastructure for designing active tools, i.e. tools manipulating an application, like for example forcing the load balancer to execute a migration of an application object.

5.2.1 Multi-Layer-Monitoring (MLM)

The MIMO system is based on the Multi-Layer-Monitoring Model (see [16]), which describes the distributed object environment in a sufficiently detailed manner for a large class of on-line tools. The model consists of six abstraction layers, from which relevant data can be collected (see figure 5.2). Interactions between elements of the same or of different abstraction layers can be monitored. The following table explains the meaning of each abstraction layer and gives the general CORBA-mapping.

Application layer	→ The entire application
CORBA mapping	→ Application name
Interface layer	→ Interfaces exported by the components
CORBA mapping	→ IDL interfaces
Distr. object layer	→ Distributed objects, providing services
CORBA mapping	→ CORBA objects
Impl. layer	→ Implementation of the distributed objects
CORBA mapping	→ Implementation objects (e.g. in Java or C++)
Run-time layer	→ Object execution environment
CORBA mapping	→ PIDs or Thread IDs
Hardware layer	→ The underlying hardware
CORBA mapping	→ Host names

For each distributed application and specific middleware environment to be monitored, those abstract layers have to be mapped to concrete entity types. In the case of CORBA, application objects, which can be uniquely identified by their IOR, are the most important entity.

5.2.2 Instrumentation Techniques

There exist two basic concepts for instrumenting an application, which has to be monitored (see [12]): *intruders* and *adapters*. The main difference is that intruders are *transparently* integrated into the application, while the adapter model requires insertion of usually small pieces of code into the application program (see figure 5.3).

Intruders:
When the application can not be rebuilt or the instrumentation has to be transparent the construction of an intruder is the appropriate method. For example the CORBA library can be instrumented by inserting *wrapper functions* into the library code. The original functions have to be renamed and called from within the *wrapper functions*, which handle the communication with MIMO. This is

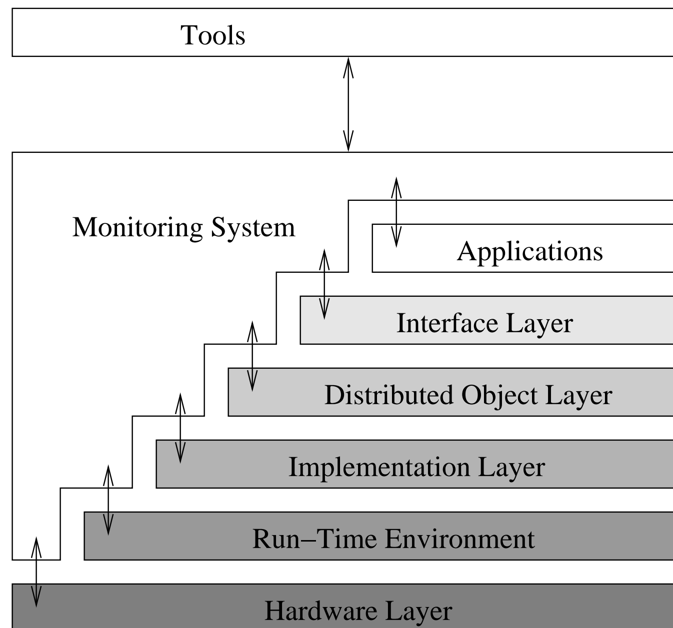


Figure 5.2: Multi-Layer-Monitoring Model.

performed by applying symbol replacement within the CORBA library. The CORBA methods of interest have to be identified and selected carefully, such as e.g. `ORB.init()` for attaching to MIMO. For this thesis however the adapter approach was used, which is described in more depth. For a detailed description of the intruder concept see [15].

Adapters:
When using the adapter model, and in order to reduce the code that has to be integrated into the original application to a minimum, it is useful to group the code for attaching/detaching to/from MIMO, as well as for sending MIMO events via the CORBA event channel in a separate adapter class. For the CORBA middleware there already exists a Java adapter class providing these services. The following three standard MIMO events are available as adapter methods.

1. Object creation (`new`)
2. Object interaction (`interaction`)
3. Object deletion (`del`)

For instrumenting an application, only a few adapter method calls have to be inserted into the application and the rest (establishing connection with MIMO, generating, packing/unpacking events etc.) is handled by the adapter class. Those

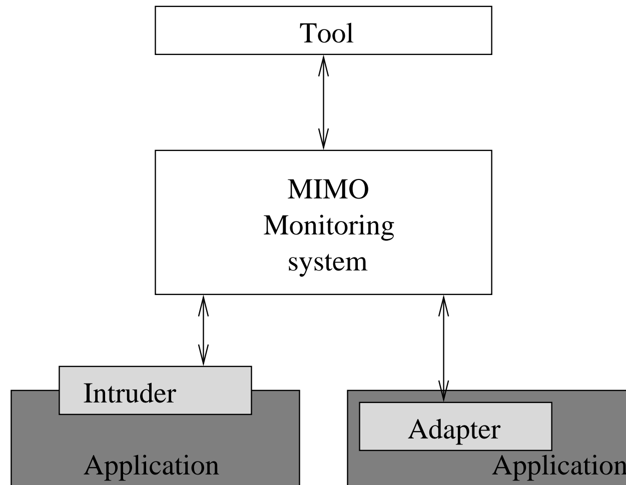


Figure 5.3: Adapter and intruder instrumentation technique for a MIMO-based system.

three basic methods should be sufficient for instrumenting a normal CORBA application. Additional methods can be added if necessary, for handling generic events, which are described in the following section.

5.2.3 Generic Events

Generic events are user-defined MIMO events, that can be specified if the standard events (e.g. `new`, `del`, `interaction`) are insufficient for the instrumentation of an application, as in the case of the load balancer. The additional user-defined CORBA events have to be specified in the MIMO `IntruderEvent.idl` file. When an adapter class is used, supplementary methods have to be added to handle the generic events.

Events are packed into a general event type called `genericEvent`. If a tool requires the information provided by these special events, it has to start an additional MIMO request, using `start_request()` with the respective service name as parameter. The generic events are gathered and channeled through MIMO and then provided to the tools interested.

Example:
The definition of a generic event for replications and the implementation of the corresponding adapter method are presented.

1. Step: Define the generic event data structure in the `IntruderEvent.idl` file.

```
struct repEvent {
    string sourceOld; //old IOR  of object to be replicated
    string sourceNew; //new IOR of object to be replicated
    string destNew;   //IOR of the replica
    string destHost;  //Name of the replicas host
  };
```

2. Step: Add a corresponding Java adapter class method (fragmentary, only important passages).

```
public void replication(String _sourceOld, String _sourceNew,
                        String _dest, String destHost)
{
   ... //initialize an IntruderEvent object: ievt
   ievt.etype = "replication"; //specify the service-name

   ...//initialize an entity list object: el1

   genericEvent ge = new genericEvent();
   //initialize a generic event object: ge
   ge.elist = el1; //set its entity list to el1

   repEvent mivt = new repEvent(sourceOld, sourceNew,
                                dest, destHost);
   //initialize a replication event object: mivt

   ge.genDesc = orb.create_any();
   repEventHelper.insert(ge.genDesc, mivt);
   //insert the replication event into the generic event

   ievt.description = orb.create_any();
   genericEventHelper.insert(ievt.description,ge);
   //insert the generic event into the intruder event

   Any aevt = orb.create_any();
   IntruderEventHelper.insert(aevt,ievt);
   //insert the intruder event data into a CORBA any

   pushEvent(aevt); //push the any event to MIMO
}
```

5.2.4 Active Tools

As already mentioned, MIMO provides the possibility to design active tools, i.e. tools, that not only monitor, but also steer the application (see figure 5.1). Commands are also issued as CORBA events, only that the communication flow is now being reversed from the tool (via MIMO) to the application.

This mechanism was firstly used in this thesis, so that some general considerations for the design of adapter-based steered applications and active tools will be made at this point. The intruder approach is inconvenient within this context, since an active tool imposes the introduction of an additional interface within the application that receives tool instructions. Since code *has* to be added to the application program the adapter approach should be used for building active tools.

Application and Adapter Requirements:
The adapter has to poll the incoming event queue regularly for incoming commands and store them in a command list. An additional thread has to be added to the application, which regularly checks the command list maintained by the adapter and executes the tool commands.

The effect of additional computations within the application has to be carefully analyzed, in order to avoid interferences like for example synchronization problems.

Tool Requirements:
The tool sends the command to the respective entity by using

```
MIMO.ToolMonitor.send_command(String requestName,
                              Entity[] elist,
                              Any params);
```

where `requestName` specifies the command name, `elist` the MLM components it is sent to, and `params` the command data. A CORBA event type for a tool command has to be specified in the `MonitorCommand.idl` file. The implementation of a migration command is presented in 6.2.6.

5.3 Introduction to MiVis

MiVis is a visualization framework based on MIMO for monitoring CORBA applications. A detailed description can be found in [10]. The display consists of a main window, which provides an entity selection frame for adding/removing monitored MLM-entities to/from the various display types. It offers three basic display types, implemented as Java Beans (see [11]):

1. The `TextDisplay` Java Bean, is a simple text window which tracks every event received by MIMO and every entity added or deleted by the user in the main window.

2. The `ScrollDisplay` Java Bean, which displays communications between entities (usually CORBA objects) in a graph. The time is displayed on the x-axis and the monitored entities are displayed on the y-axis as horizontal lines parallel to the x-axis. Interactions are depicted as vertical arrows between entities.

3. The `CallFrequency` Java Bean displays the call frequency of selected entities and distinguishes between caller and callee (different colors). Entities are once again displayed on the y-axis and the call frequency is represented as horizontal bar parallel to the x-axis.

Additionally, each of the three displays has an options window for selecting display-specific properties like colors, delay times etc. One of the key features of MiVis is, that it can easily be extended by adding further Java Beans to the Bean path (i.e. new display types), without modifying the MiVis core, i.e. it provides an infrastructure for developing new CORBA visualization tools. The Java Beans have to fulfill certain requirements specified in [10].

At start-up the tool starts an asynchronous request for interactions. When an entity-type button is pressed by the user in the selection frame of the main window, a synchronous request is issued to MIMO, in order to detect if new entities of the specific type have been created since the last request. When a display Java Bean has been opened by clicking on the respective icon, entities to be monitored can be added or removed by clicking on the add or delete buttons. The MiVis architecture consists of four components (see figure 5.4):

1. The *gatherer*, the lowest layer of MiVis, collects events received from MIMO, unpacks them, and propagates them to the processors as Java events.

2. The *processors* filter the data and propagate it to the interested Java Beans.

3. The *Java Beans* consume the data obtained by the processor and display it.

4. The main window forms part of the *MiVis GUI*, in which the Java Beans are started and the options menu is displayed. The selection frame issues requests directly to MIMO for detecting newly added entities.

Thus, MiVis and MIMO provide an appropriate infrastructure for the design of an *active on-line visualization tool* for monitoring the load balancer and the realignment application. The specification and implementation of this new tool will be presented in the following chapter.

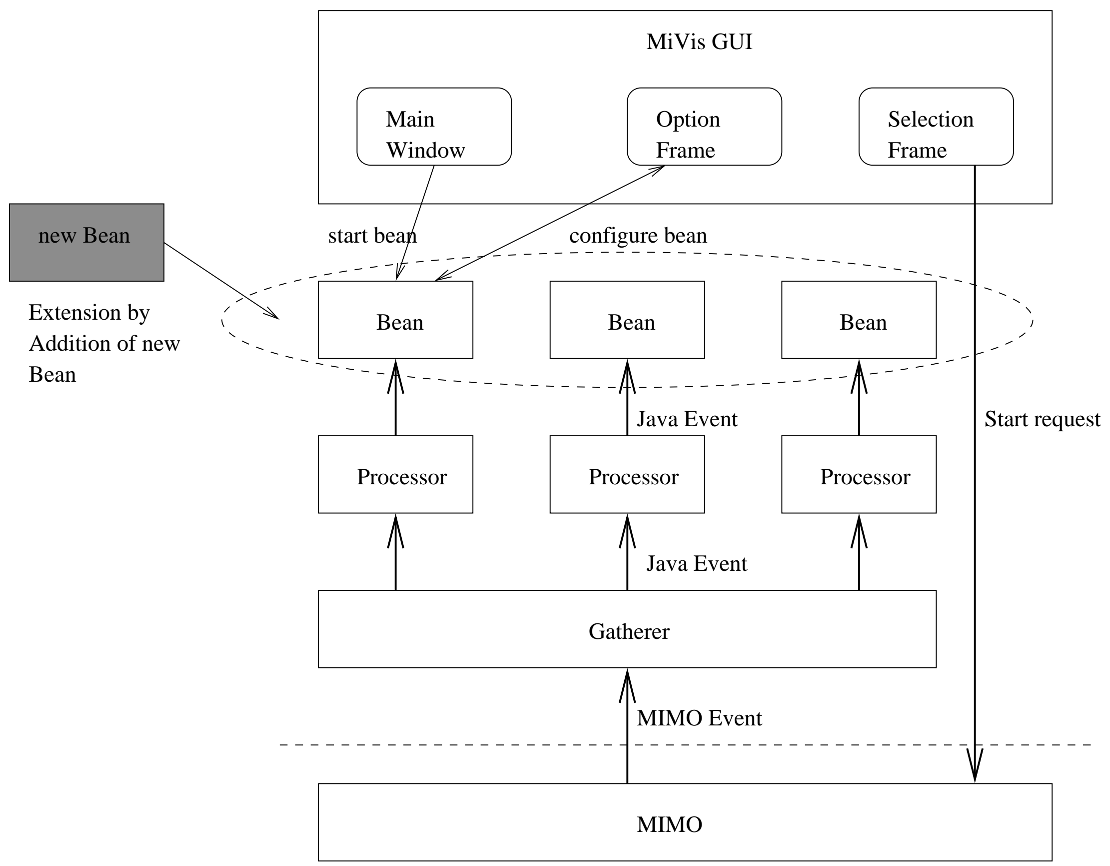


Figure 5.4: MiVis Architecture, new Java Beans can be inserted into the existing infrastructure.

Chapter 6

Interoperability of MiVis and the Load Balancer

This chapter presents the specification, design, and implementation of the new MiVis Java Bean, which monitors the load balancer *and* the application and furthermore allows to issue migration commands. The new JacORB MIMO adapter, the new IDL interface for the load balancer with generic events and the instrumentation of the load-balanced system are also described. The tool development methodology proposed in [14] was used for the design of the new tool.

6.1 Specification of the Degree of Interoperability and of the Tool Functionality

A classification for the degree of interoperability of tools could be as follows (for a more detailed analysis see [18]):

1. The tools run on the same system and do not perturb each others work.
2. The tools are used for the same application without perturbing each others work.
3. The tools are used for the same application without perturbing each others work and in addition one tool provides information to the other (supplier/consumer model).
4. The tools are used for the same application without perturbing each others work and exchange information (highest degree of interoperability).

The goal is to design a 4th degree system for the load balancer and MiVis, by implementing a new MiVis Java Bean. This new Java Bean should visualize the following information:

- The nodes, the load balancer considers for load distribution and the servant objects located on them.
- Nodes on which the clients are located and the respective client objects.
- Host load value for each host controlled by the load balancer.
- Object load value for each servant object.
- Application object interactions, i.e. method invocations of `corbaCompute()` and `corbaGetReferenceData()`, in a differentiated manner.
- Count and display the interactions between client and servant objects.
- Actions performed by the load balancer (migrations/replications).

A menu for setting properties, such as the time an interaction is displayed and if migrations or replications leave a trace on the screen, should also be available.

Furthermore, a tool command and an auxiliary generic event have to be added for testing and evaluating the active tool infrastructure.

- Provide a drag and drop functionality for issuing migration commands to the load balancer.
- The load balancer has to return a migration status event, indicating if the migration has been conducted successfully or delivering an appropriate error message.

6.2 Instrumenting the Load Balancer and the Application

The first step was to define the MLM mapping, an appropriate generic event interface, and extend the adapter class accordingly, using the methodology described in [14]. Thereafter the load balancer and the realignment application had to be instrumented, using the new adapter class methods, in order to obtain the required data.

Problems had to be resolved concerning the identification of client and servant objects, since the load balancer hides and changes dynamically servant object IORs. Furthermore, interoperability problems concerning JacORB/ORBacus had to be resolved. Finally, the load balancer had to be modified, in order to be able to execute external migration commands.

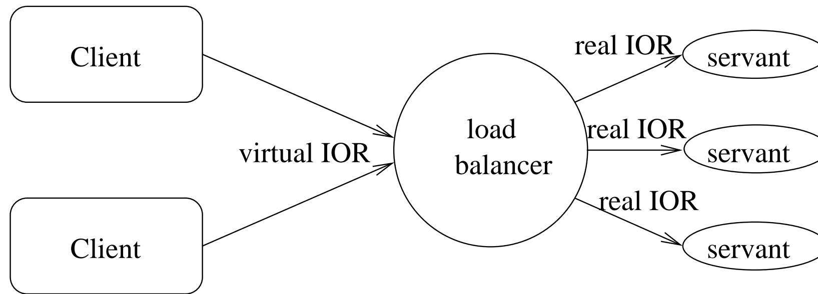


Figure 6.1: Client objects access one single IOR, the requests are then being forwarded and distributed by the load balancer to the actual real IORs of the replicas.

A new command option and a new property (see section 4.2) have been added to the client and the load balancer respectively, for enabling and disabling monitoring.

The option `[-MIMO]`, enabling monitoring, has been added to the client command options and the property `WithMIMO` can be set to `true` or `false` in the applications `Realign.imr` file for the server.

The new property of the load balancer: `MIMO` can be set to `on` or `off` in the `imr.properties` file.

6.2.1 Initial Approach and Associated Problems

The first idea for approaching the problem was to instrument initially *only* the realignment application and visualize it with the existing MiVis Java Beans. It soon became evident, that this approach was a dead end due to the way the load balancer changes and hides servant IORs, which are used for unique object identification. Therefore, an integral approach, instrumenting the realignment application and the load balancer, was necessary to handle the particularities of the system. The client objects have access to a *virtual* IOR only and the distribution to the *real* IORs of the replicas is performed by the load balancer (see figure 6.1).

Therefore, there exists no possibility for the client object to obtain the *real* IOR of the servant object it is actually requesting the compute service from. Thus, `corbaCompute()` and `corbaGetReferenceData()` could only be instrumented on the servant-side of the code. Once again, to fulfill the specification requirements, the servant has to know which client it is providing the service to, which is usually not the case. Clients normally are anonymous. For the specific application however, the clients are not anonymous. The instrumentation problem can generally be solved by adding the client's IOR as supplementary parameter to the CORBA methods which have to be monitored.

A basic problem was that the servant object itself does not know its actual real IOR, since it is hidden by the load balancer and it is not aware that it changes dynamically during program execution, due to the invalidation [1] mechanism. The following solution is proposed for handling this fundamental problem:

Obtaining the real IOR at servant creation time:
The initial real IOR of a servant object is available in the modified `JacORB.POA` class, in the `protected create_object()` and `protected recreate_object()` methods. A new empty method `setRealIOR(org.omg.CORBA.Object obj)` has been added to the `org.omg.PortableServer.Servant` class, which *must* be overwritten and implemented by the application's servant class, extending `Servant`, in order to set its initial real IOR. This method is invoked before the two `JacORB.POA` methods mentioned above return the CORBA object created e.g. :

```
protected org.omg.CORBA.Object create_object(...)
{
   ...

   servant.setRealIOR(replica);

   return replica;
}
```

The implementation of `setRealIOR(org.omg.CORBA.Object obj)` in the servant class of the application consists of just setting an attribute, for example `org.omg.CORBA.Object realIOR`, to `obj`.

Tracking dynamic IOR changes (invalidations):
Real object IORs are reassigned by the load balancer at each replication. Furthermore, they are once more reassigned shortly *after* each migration and replication by the invalidation mechanism. The idea consists of maintaining the *initial* real IOR, which was assigned at object creation as *father* address and maintain the *actual* real IOR, the *son* object, in a suitable data structure. The load balancer creates a generic invalidation event when IORs change. The new IOR addresses assigned after a replication event become additional parameters of the generic replication event. The details of the data structure for maintaining the *father-son* relation will be discussed in the presentation of the new MiVis Java Bean (see 6.3.2), since it is handled by the Bean.

[1] Invalidation is a load balancer action which reassigns IORs to servant objects after replications and migrations.

6.2.2 Multi Layer Monitoring Mapping

The MLM mapping for the client, the server, and the load balancer was straightforward:

Mapping for the client:

Application layer	→ Realignment
Interface layer	→ `corbaGetComputeData()` interface
Distributed object-oriented layer	→ CORBA IOR
Implementation layer	→ Java `Realign` class
Run-time environment layer	→ PID
Hardware layer	→ Host name

Mapping for the server:

Application layer	→ Realignment
Interface layer	→ `corbaCompute()` interface
Distributed object-oriented layer	→ CORBA IOR (initial real IOR)
Implementation layer	→ Java `Compute` class
Run-time environment layer	→ PID
Hardware layer	→ Host name

Mapping for the load balancer:

Application layer	→ Load Balancer
Interface layer	→ Load Balancer IDL interfaces
Distributed object-oriented layer	→ CORBA IOR (virtual IOR)
Implementation layer	→ Load balancer Java classes
Run-time environment layer	→ PID
Hardware layer	→ Host name

6.2.3 Visualization Interface Definition

The data interface consists of a definition of generic events, for extracting the required visualization data from the load balancer and the realignment application, according to the specification.

Data Interface:
Specified in `IntruderEvent.idl`. Object IORs are converted and passed as strings.

Definition of generic events for replications (stateful and stateless). The new initial IOR of the replica, the old and new IOR of the original object, and the host of the replica are passed.

```
struct repEvent {          //stateless replication
  string sourceOld;        //actual real IOR of object
  string sourceNew;        //newly assigned  real IOR of object
  string destNew;          //initial real IOR of replica
  string destHost;         //host of the replica
};

struct repEventStateful { //stateful replication
  string sourceOld;        //actual real IOR of object
  string sourceNew;        //newly assigned real IOR of object
  string destNew;          //initial real IOR of replica
  string destHost;         //host of the replica
  long stateSize;          //size of the data
                           //transferred (not used)
};
```

Definition of generic events for migrations (stateful and stateless). The new initial IOR of the migrated object, the IOR of the object on the original host, and the host of the migrated object are passed.

```
struct migEvent {          //stateless migration
  string source;           //actual real IOR of object
  string dest;             //new initial real IOR of object
  string destHost;         //name of new host
};

struct migEventStateful { //stateful migration
  string source;           //actual real IOR of object
  string dest;             //new initial real IOR of object
  string destHost;         //name of new host
  long stateSize;          //size of the data
                           //transferred (not used)
};
```

Definition of a generic event for client object creations. The IOR of the client object and the host name are passed.

```
struct newClientObject{    //client object creation
  string name;             //IOR
  string hostName;         //host name
};
```

Definition of a generic event for servant object creations performed by the generic factory. The IOR of the servant object and the host name are passed.

```
struct newServantObject{   //servant object creation
                           //by generic factory
  string name;             //initial real IOR
  string hostName;         //host name
};
```

Definition of a string sequence and a generic event for transferring the host names of the nodes controlled by the load balancer.

```
typedef sequence<string> SeqString; //list of strings

struct nodeList{           //hosts available to
                           //the load balancer
  SeqString List;          //list of host names
};
```

Definition of generic events for the host load and servant object load values. The load value and the host name or actual real IOR of the servant object respectively are passed.

```
struct hostLoad{           //load of a host controlled
                           //by the load balancer
  double load;             //load value
  string hostName;         //host name
};

struct objectLoad{         //load of a servant object
  double load;             //load value
  string objName;          //actual real IOR
};
```

Definition of a generic events for invalidations. The old and new actual IORs are passed as parameters.

```
struct invalidationEvent{  //invalidation, reassignment
                           //of servant object IORs
  string oldName;          //old real IOR
  string newName;          //new real IOR
};
```

6.2.4 Command Interface Definition

The command interface definition consists of the translation of the tool command specification into IDL code.

Command Interface:
Specified in `MonitorCommand.idl`.

```
struct MigrationCommandEvent { //migration command
  string destHost;             //destination host
  string Object;               //actual real IOR
};
```

Specified in `IntruderEvent.idl`.

```
struct migrationStatus{ //migration status indicating
                        //if migration was succesful
  long status;          //integer value for status
};
```

The migration status values have been defined as follows:

```
0:   object not found
1:   migration OK
2:   host not found
```

6.2.5 Extension of the Manual Adapter

In order to instrument the application and the load balancer, adequate methods for issuing the generic events, defined in section 6.2.3 and section 6.2.4, had to be implemented. Therefore, the existing Java adapter class had to be extended, by one adapter method for each event defined. The adapter methods were implemented as in the example of section 5.2.3. Furthermore, an adapter method `getCommand()`, that enables the application to check for incoming events, had to be provided. It returns an `MigrationInfo` object containing the command data of a tool migration command.

A fact that led to some problems was, that the Java adapter class had to be implemented using the JacORB since the load balancer and the application are based on it. Thus, this JacORB adapter had to communicate with MIMO, which uses ORBacus, via the CORBA event channel. Initially the adapter could attach itself to MIMO, but sending events did not work at all. After time-consuming investigation it was found, that an `if`-clause within the automatically generated ORBacus event helper classes had to be removed.

This example demonstrates that there are still some deficiencies concerning interoperability of different CORBA implementations.

6.2.6 Instrumentation and Command Implementation

The instrumentation of the application and the load balancer for provision of the specified visualization data, using the adapter methods, was straight forward. The points at which instrumentation code had to be inserted into the *realignment* code could easily be identified, taking into account the results and considerations of the initial approach. Only adapter methods handling client object creation and application object interactions were inserted.

The remaining events are generated by the instrumentation of the load balancer. The instrumentation points and classes have been determined in cooperation with the developer of the automatic load balancer. Since multiple Java classes were affected by the instrumentation and thus required access to the adapter, an additional Java class was introduced holding a globally available adapter object, which is instantiated at start-up. The load balancer creates the remaining visualization interface events like e.g. servant object creation, replication, migration, object load etc.

Load balancer instrumentation example (stateless migration):
`AdapterConstant.Adapter` is the globally available adapter object.

```
if(AdapterConstant.WithMIMO) //check if monitoring is enabled
  {
     String NewHost =
       corbaView.servers.find(adapterElementTarget.orbControl).host;
     //get the name of the new host, load balancer internal method

     AdapterConstant.Adapter.migration(replicaElement.replica,
                                       replicaNew, NewHost);
     //send a stateless migration event to MIMO by
     //using the respective adapter function
  }
```

The adaptation of the load balancer, for receiving tool commands, was more complicated, because this could not be implemented by insertion of simple adapter method calls.

Firstly, as already mentioned in section 5.2.4, a special thread has to be added to the load balancer, that regularly checks if a tool command has been issued, by calling the adapter's `getCommand()` method. An additional class `MimoThread`, extending `Thread`, has been added for this purpose. The detached `MimoThread` is started before the load balancer enters the evaluation loop.

Secondly, for executing external migration commands a supplementary method `userMigration()` had to be offered to the `MimoThread` by the load balancer. A migration outside the load balancer's evaluation cycle led to some problems,

mainly concerning synchronization issues, which could however be resolved in a joint effort with the developer of the load balancer.

This experience shows that building active tools is a much more complex and error-prone process, since it induces important modifications of the application program, if an infrastructure for receiving external commands is not available.

6.3 The new MiVis Java Bean

The design of the new MiVis Java Bean consisted of two major phases, the specification of the display and its implementation.

6.3.1 Display Design

The following arrangement for displaying the obtained data is proposed (see also screen-shots at the end of this section):

All information is displayed in one window. The upper two thirds of the window display the servant hosts as rectangles. They are arranged in a grid (minimum size 2 x 2). On top of the rectangle, the host-name is displayed and on the right hand side of it, the load is represented as bar and as numerical value.

Client hosts are depicted as rounded rectangles and listed in the lower third of the window. Again the host name is displayed on top.

Client and servant objects are represented by ovals, situated within the respective host rectangles. The numerical value of the servant object load is printed to the screen as numerical value in the servant's oval. A call to `corbaCompute()` is indicated by a blue straight arrow between the oval's centers. The method `corbaGetReferenceData()` is visualized by an offset parallel turquoise arrow, for not overwriting the compute arrow.

Migrations and Replications are depicted by red and yellow straight arrows respectively, between the original and the migrated or replicated object.

The time in *ms* for which those arrows are depicted can be set within the options menu, for adapting it to different applications. Furthermore, if `Trace` is set to `true` in this menu, replication and migration arrows, as well as source objects leave a trace on the screen and are not erased.

Finally, an object migration command can be issued by clicking on a servant object and dragging it to the desired host. No other migration command can be kicked off, until the receipt of the return status. Until then a rectangular text window is opened in the center, informing the user about that fact. It also prints out a message about the migration status when received.

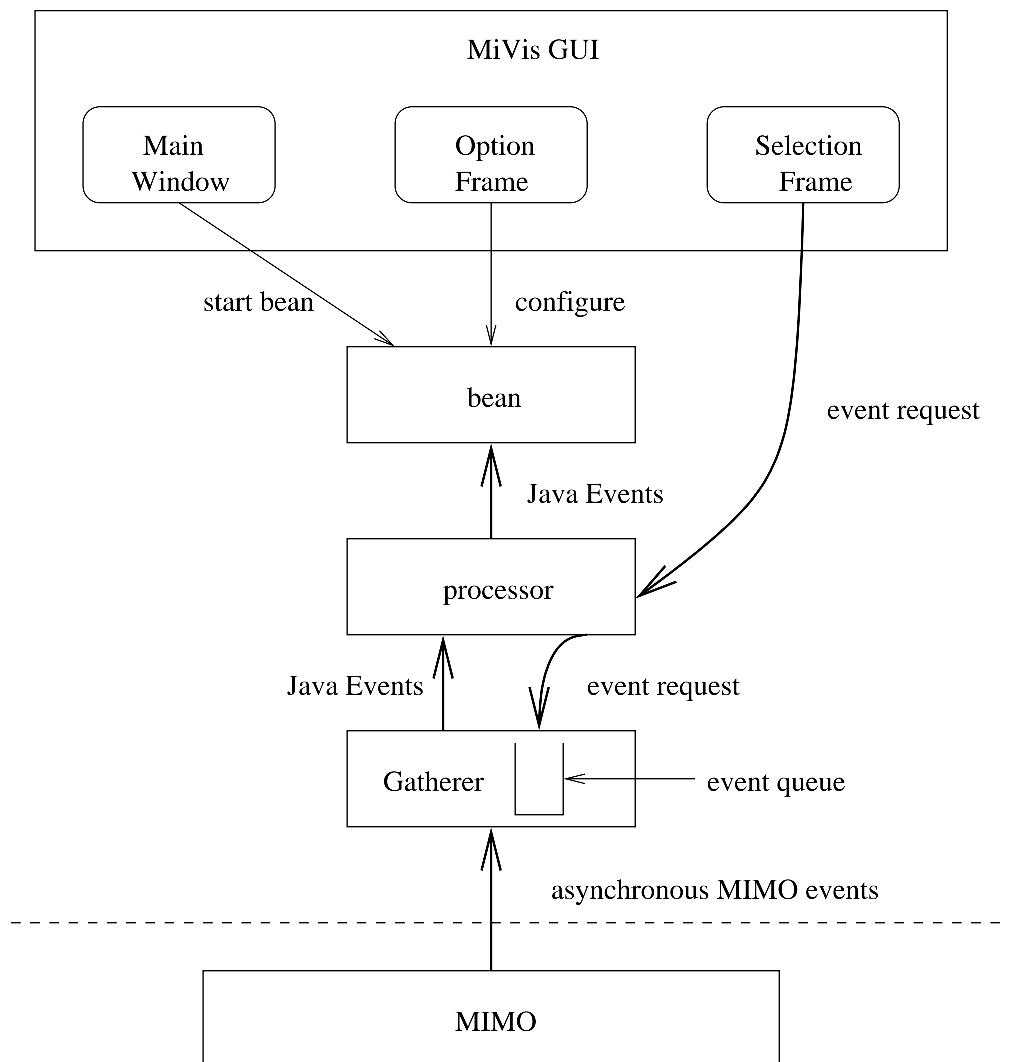


Figure 6.2: Proposal for a new and more flexible MiVis design.

6.3.2 Java Bean Implementation and Improved MiVis Design Proposal

MiVis Insufficiencies:
The implementation of the Java Bean was more complex than expected, since the MiVis system showed not to be as flexible as required. The particularity of the specific visualization tool is, that all events have to be received in asynchronous mode. This has not been foreseen in MiVis and therefore extensive changes to the MiVis core program, i.e. the MiVis GUI, the processors, and the gatherer, unfortunately became necessary.

The acquisition and selection of new entities for monitoring in MiVis is user-initiated. In contrast all events and entities created have to be propagated directly to the new Java Bean. to it. The insufficiency in the MiVis design philosophy is, that e.g. checking for object creations circumvents the gatherer-processor structure and directly communicates with MIMO (see section 5.3 and figure 5.4). A more appropriate approach would be to collect all events asynchronously within the gatherer and then design different processor types, that either propagate those events directly or deliver them when requested by the MiVis GUI selection frame (see figure 6.2).

The redisgn of MiVis was however not the subject of this thesis. The gatherer and processor were modified in such a way, to deliver all events asynchronously to the MiVis GUI. The selection frame thus becomes obsolete for the new Java Bean.

Command Implementation:
For reasons of simplicity and in order not to perturb the load balancer too much in its work, the choice was made to allow issuing only one migration command at a time. This means, that the mouse has to be blocked until the migration has been executed by the load balancer. In addition to this, the introduction of the return status event was necessary, because the desired object might already have been migrated by the load balancer. Furthermore, it was considered helpful for debugging purposes.

When the object to be moved and the new hosts have been selected with the mouse, the graphic objects have to be mapped to their internal representations (IOR and host name) and then passed as parameters to the function `userMigration()`, which sends the command to the load balancer via MIMO and blocks the mouse. Figure 6.3 depicts the control flow and components such an command execution traverses.

Handling changing IORs:
As already mentioned, a data structure for handling the dynamically changing IOR addresses was also implemented in the Java Bean. It would however be more appropriate to handle this in the adapter class for a clearer separation between the graphical interface and the specific aspects of the load balancer. This and

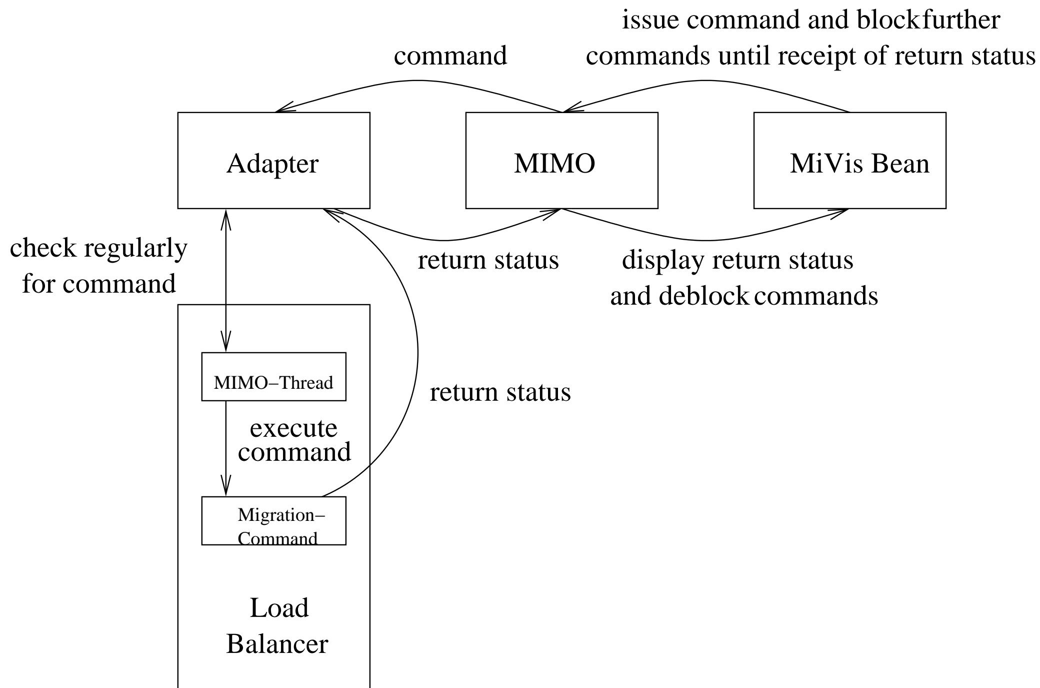


Figure 6.3: Structure of a migration command execution.

the improvement of MiVis, according to the proposal, could be subject of future work.

The idea, as stated earlier in section 6.2.1, is to maintain a *father-son* relation between the initial real IOR, which was assigned at object creation time and the actual real IOR, which is changed by replications, and shortly after replications or migrations by the invalidation mechanism. When one of those two events occurs, the old actual IOR is looked up in a list of *father-son* relations and replaced by the new one. In this step the initial real IOR (*father*) becomes also available, which is needed to identify the graphical objects affected by the IOR change, since all objects are referred to by their initial real IOR. If it is the first dynamically changed IOR the *father* IOR is assigned its first *son*, which previously was marked empty.

On the other hand when a migration command is issued the actual real IOR has to be looked up, for the *father* IOR selected.

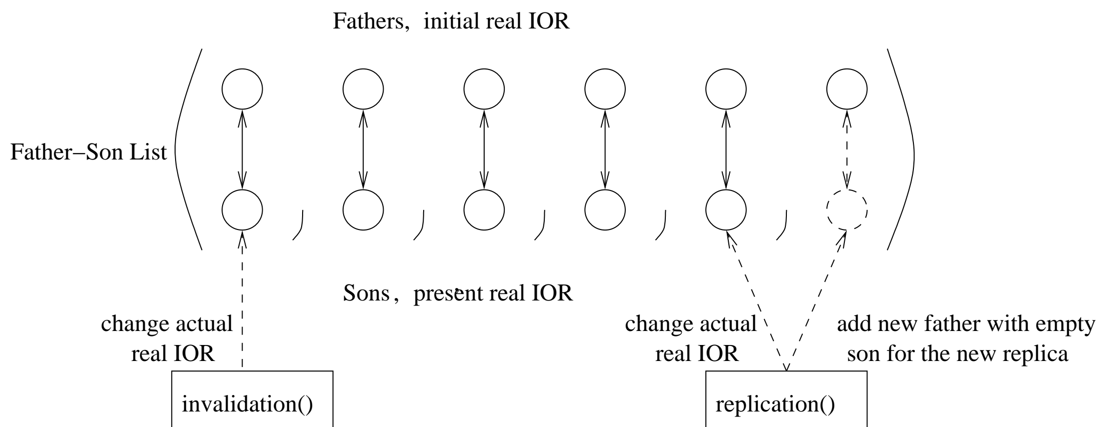


Figure 6.4: Dynamic IOR changes are handled by the above data-structure.

6.4 Instrumentation of Load-Balanced Applications

At this point the necessary steps and modifications for instrumenting a load-balanced application will be described.

Client:

- Define and instantiate a new adapter object.
- Register the new client object IOR and its host, by using the adapter method `newClientObject()`.

Servant:

- Define and instantiate a new adapter object.
- Define an attribute for the initial real IOR of the servant object, for example `org.omg.CORBA.Object realIOR;`.
- Implement the `setRealIOR(org.omg.CORBA.Object obj)` method, by assigning `realIOR = obj;`.
- Instrument all CORBA method calls or implementations, using the adapter method `interaction()`.

Application IDL *interface:*

- Add the client IOR as supplementary parameter to all CORBA methods that need to be instrumented.

MiVis Java Bean:

- If desired, implement additional arrow types for displaying CORBA method invocations. The methods `drawArrow()` (straight arrow between objects) and `drawMyArrow()` (offset arrow between objects) are already available (see section 6.3.1 and screen-shot 6.5).
- Modify the `if`-clause in method `handleEvents()`, to include the CORBA method names of the application.
- Modify the `if`-clauses in method `drawInteraction()`, to display the appropriate arrow type for the CORBA method name.

This guideline permits a straight forward instrumentation of the the client and servant components. Attention has to be drawn to the fact, that all interactions of the application have to be instrumented on the servant-side of the system and that the IDL interface of the application may need to be slightly modified. Furthermore, the minor modifications in the Java Bean allow a flexible, fast and differentiated visualization of a load-balanced application. The arrow types provided should be sufficient for most applications.

6.5 Evaluation of Tool Functionality and Interoperability

The work conducted concerning interoperability of the load balancer, MIMO, and MiVis showed, that the highest level of interoperability for the system could be achieved, without influencing the correct execution of the application. Furthermore, the concept of MIMO generic events and MIMO commands could be proved. Although problems with the MiVis architecture were encountered, a Java Bean complying totally with the tool specification could be designed. Since the evaluation of a graphical display is always a matter of subjective judgment, it has to be stated, that it was positively received by the people involved, i.e. the developer of the load balancer and the developer of MIMO.

The display is presented by a series of screen-shots, demonstrating all key features of the visualization tool.

Screen-shot 6.5 presents the basic layout in a simple configuration. A client with two threads has just been started (bottom left) and issues the *first* `compute()` request. Thus, the servant object issues a request for the reference data, which is displayed as offset parallel turquoise arrow. The 4 nodes controlled by the load balancer are arranged in a grid. N/A indicates, that load values for servant objects and hosts are not available yet, since they are acquired periodically.

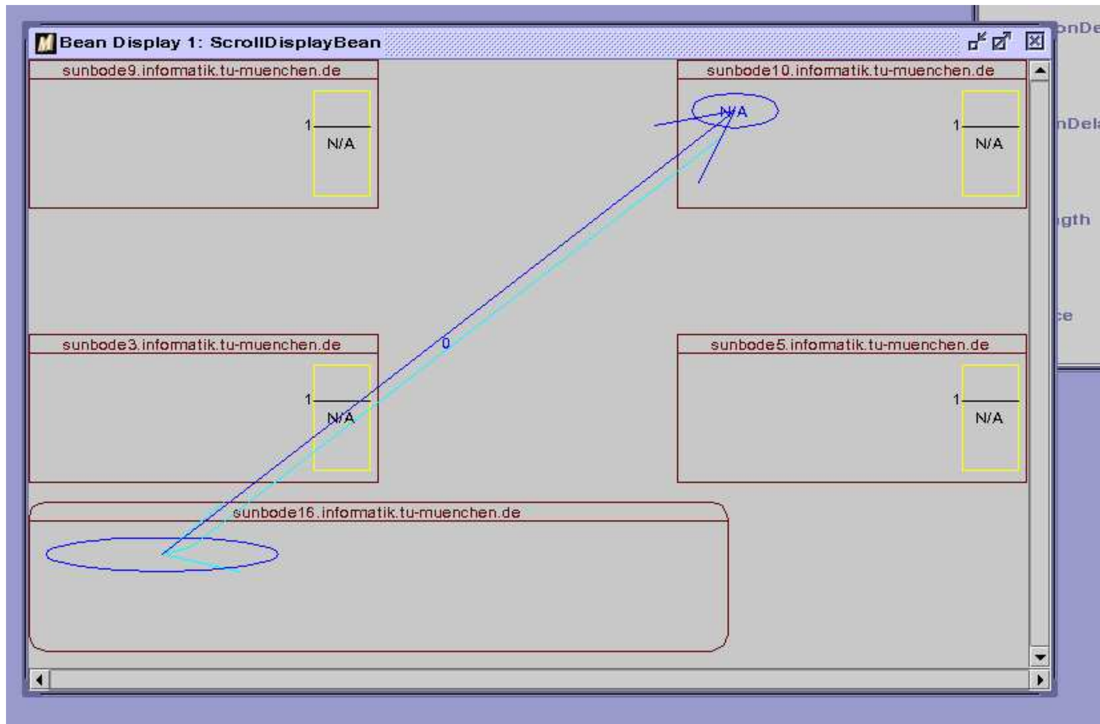


Figure 6.5: Screen-shot: New Display Layout

Screen-shot 6.6 was taken during the same test run as screen-shot 6.5. It presents the visualization of a replication conducted by the load balancer (yellow arrow). The requests of the two client threads are being distributed among the two replicas, i.e. `compute()` requests can be issued in parallel now. Load values for hosts and the initial servant object have become available in the meantime.

Screen-shot 6.7 presents the execution of a user-initiated migration. The migration command has already been issued by the tool and executed by the load balancer (red migration arrow). The return status event generated by the load balancer has not arrived yet. Thus, the migration text window is still visible and informs the user, that it is waiting for the return status event and will block the mouse until then.

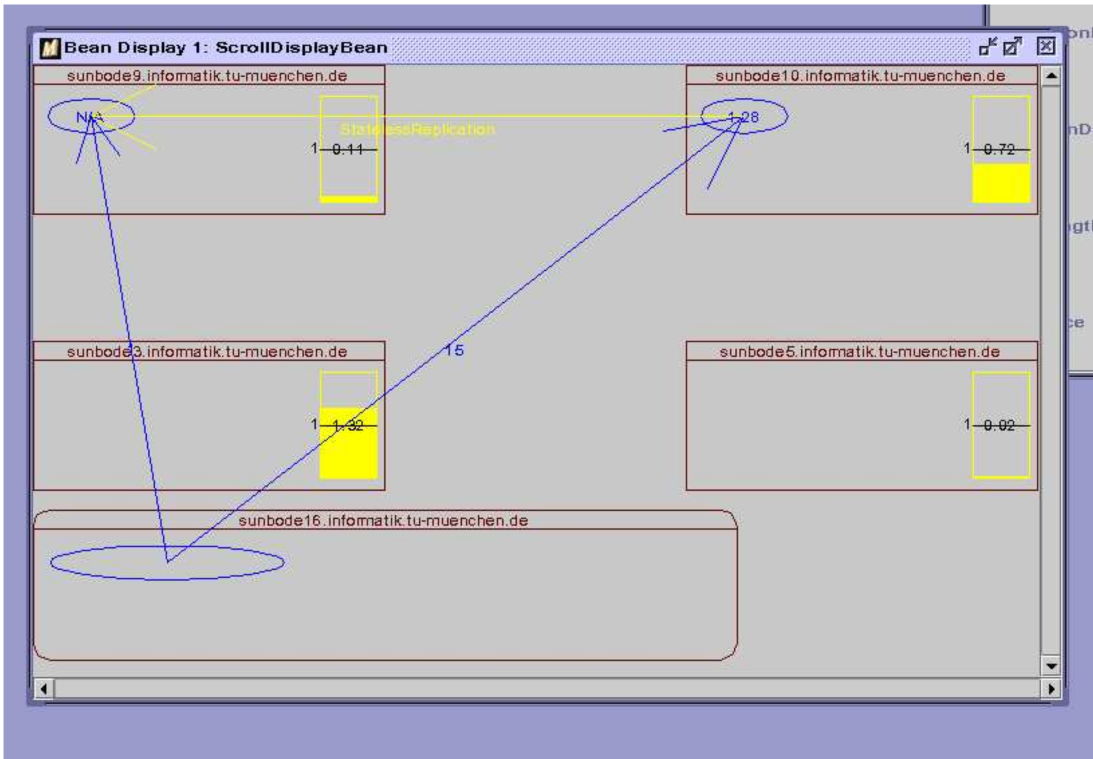


Figure 6.6: Screen-shot: Representation of Replications

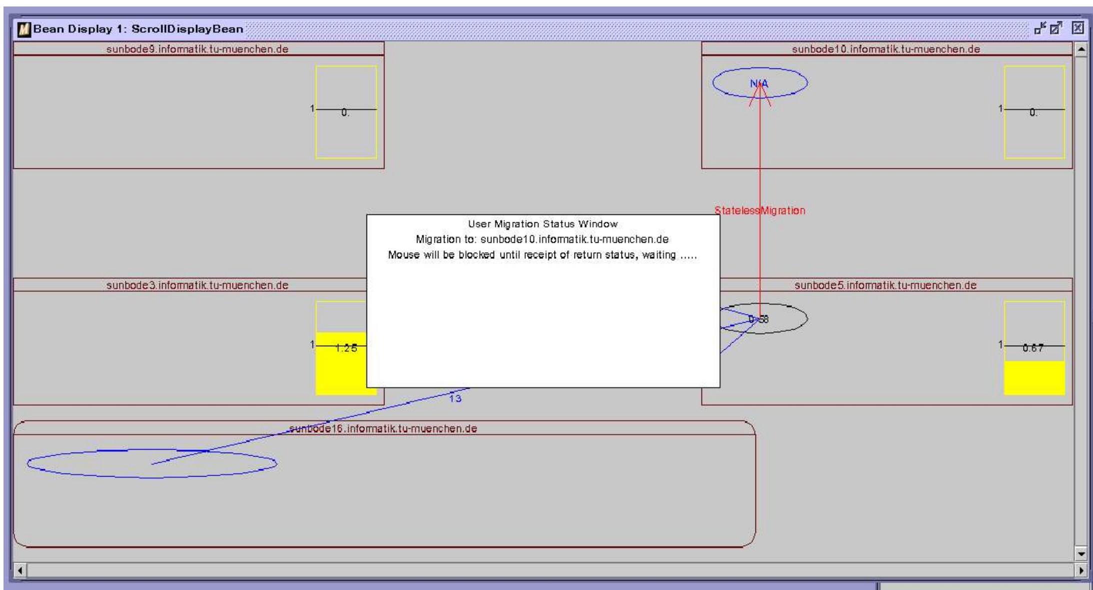


Figure 6.7: Screen-shot: Execution of a Migration Command

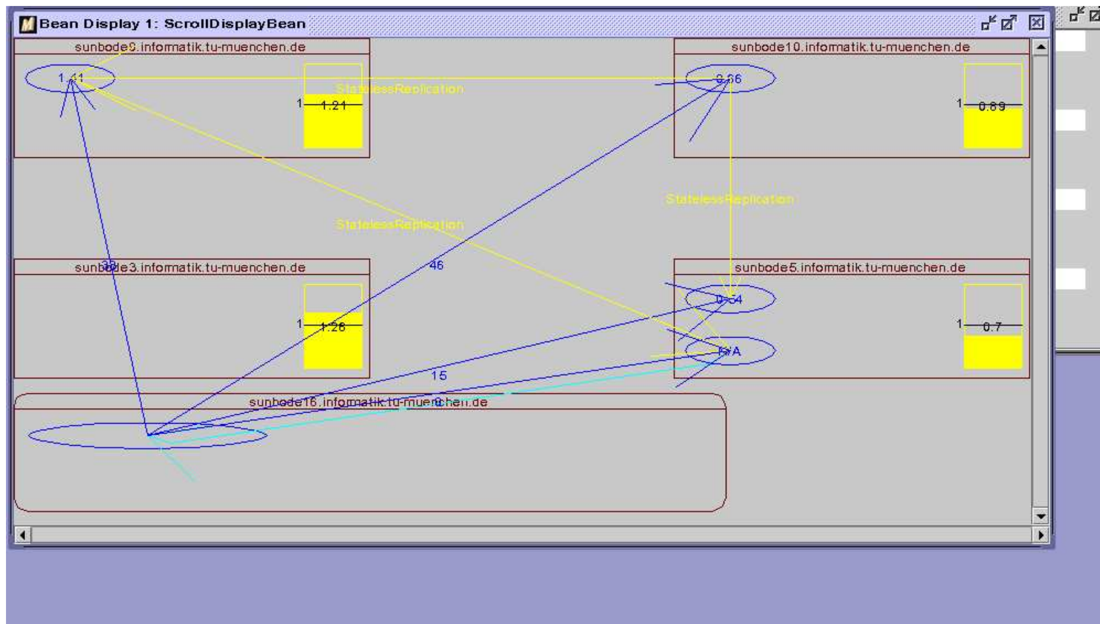


Figure 6.8: Screen-shot: Complex Replications

Screen-shot 6.8 presents a complex situation for a client with 4 threads similar to the test run presented in 4.3.2. The third replication has just been performed by the load balancer. The 4th replica on machine `sunbode5` receives its first `compute()` request (blue arrow) and issues a request for the reference data (turquoise arrow). The other replicas handle requests number 15, 33 and 46 respectively (see request counters in the middle of the blue arrows). The trace option was chosen, in order to display the replication tree rooted at `sunbode10` (yellow arrows), where the initial object was placed. Node `sunbode3` did not host a replica, since it was heavily loaded by another application during the test run..

Chapter 7

Conclusion

During this thesis two tools for distributed middleware environments have been tested and evaluated with a medical real world application.

The C++ realignment application has been transformed into a Java program using JNI and showed that the utilization of this interface does not lead to major performance penalties.

It then was transformed into a Java/CORBA program and adapted to the requirements imposed by the load balancer. Furthermore, these program modifications provided adequate mechanisms for an efficient parallelization of the application (multi-threading) and various program configurations. The results of this stage showed, that the load balancer performs as expected when applied to a large application. In addition to this, the test-phase helped discover some bugs in the load balancer.

The final phase consisted of getting two tools (load balancer, MiVis), the monitoring system (MIMO), and the realignment application to work together, by monitoring and steering the load balancement of the application, achieving the highest possible degree of interoperability. Within this context, MIMO generic events and MIMO commands were firstly used for the design of a new MiVis Java Bean, visualizing the load balancement process. The new Java Bean with its migration service was the first active tool based on MIMO and the graphical display was broadly accepted. Concepts for implementing generic events and tool commands have been presented, as well as a proposal for a revised MiVis architecture. A guideline for adapting other load-balanced applications to the new monitoring tool has been provided, together with a suitable Java adapter class. During the final phase, a new version of the load balancer, providing a method for the execution of external migrations could be tested and improved, in a joint effort with its developer.

Future work could cover the analysis of application scenarios for the load-balanced realignment application, e.g. a realignment computing center using a large cluster of nodes or a service by the university, provided during night-time,

when most nodes are running idle (interesting for the analysis of the load balancer's scalability). This would also include absolute performance tuning, like for example the implementation of a more sophisticated multi-threading concept, similar to the one proposed, or a more efficient JNI interface.

Another aspect, future work could cover, is the improvement of the MiVis architecture following the proposal given in section 6.3.2, figure 6.2 and the integration of the dynamic IOR handler into the adapter component. Additional tool-commands for shutting down servers (move all objects away and remove host from load balancer) or performing replications could also be of interest. Extensions of the display e.g for displaying client threads can be thought of.

Abbreviations

CORBA	Common Object Request Broker Architecture
FIFO	First In First Out
fMRI	functional Magnetic Resonance Imaging
IDL	Interface definition Language
IOR	Interoperable Object Reference
JNI	Java Native Interface
LRR	Lehrstuhl für Rechnertechnik und Rechnerorganisation
NFS	Network File System
MIMO	MIddleware MOnitoring System
MiVis	Middleware Visualization System
MLM	Multi-Layer-Monitoring
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
PET	Positron Emission Tomography
PID	Process Identification Number
POA	Portable Object Adapter
PVM	Parallel Virtual Machine
SNMP	Simple Network Management Protocol
SPM	Statistical Parametric Mapping

Bibliography

[1] Beth Sterns: Java Native Interface.
http://java.sun.com/docs/books/tutorial/native1.1/index.html/

[2] Jack Andrews: Interfacing Java with Native Code, Performance Limits.
http://www.str.com.au/jnibench/index.html/

[3] Marcel May: Diplomarbeit: Vergleich von PVM und CORBA bei der verteilten Berechnung medizinischer Bilddaten, LRR-TUM April 2000.

[4] Karl Friston: Spm, The Wellcome Department of Cognitive Neurology, University College London.
http://www.fil.ion.ac.uk/spm/

[5] The Math Works Inc.: Matlab.
http://www.mathworks.com/

[6] Robert Orfali, Dan Harkey: Client/Server Programming with Java and CORBA (second edition), John Wiley & Sons, INC. 1998.

[7] Andreas Sayegh: CORBA Standard, Spezifikation, Entwicklung, O'Reilly 1997.

[8] JacORB.
http://jacorb.inf.fu-berlin.de/

[9] ORBacus[tm] for C++ and Java.
http://www.ooc.com/ob/

[10] Michael Rudorfer: Diplomarbeit: Visualisierung des dynamischen Verhaltens verteilter objektorientierter Anwendungen, LRR-TUM August 1999.

[11] Java Beans.
http://java.sun.com/products/javabeans/

[12] Günther Rackl: Monitoring Globus Components with MIMO, TUM-10006 SFB-Bericht Nr. 342/06/00 A March 2000.

[13] Markus Lindermeier: Load Management for Distributed Object-Oriented Environments. In International Symposium on Distributed Objects and Applications (DOA'00), pages 59-68, Antwerp, Belgium, 2000. IEEE Computer Society.

[14] Günther Rackl and Thomas Ludwig: A Methodology for Efficiently Developing On-Line Tools for Heterogeneous Middleware. In Ralph H. Sprague Jr., editor, Proceedings of the 34th Annual Hawaii International Conference on System Sciences – HICSS-34. IEEE Computer Society, January 2001.

[15] Günther Rackl, Markus Lindermeier, Michael Rudorfer and Bernd Süss: MIMO – An Infrastructure for Monitoring and Managing Distributed Middleware Environments. In Joseph Sventek and Geoffrey Coulson, editors, Middleware 2000 — IFIP/ACM International Conference on Distributed Systems Platforms, volume 1795 of Lecture Notes in Computer Science, pages 71-87. Springer, April 2000.

[16] Günther Rackl: Multi-Layer Monitoring In Distributed Object-Environments. In Lea Kutvonen, Hartmut König, and Martti Tienari, editors, Distributed Applications and Interoperable Systems II — IFIP TC 6 WG 6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), pages 265-270, Helsinki, June 1999. Kluwer Academic Publishers.

[17] Günther Rackl: Monitoring and Managing Heterogeneous Middleware, Dissertation, TU München, 2001. To appear.

[18] Jörg Trinitis: Interoperability of Distributed Checkpointing and Debugging Tools, Dissertation, TU München, 1999.

[19] W.Stallings: Snmp, Snmpv2, Snmpv3, and Rmon 1 and 2, Addison Wesley 1998.

[20] W. DePauw, Richard Helm, Doug Kimmelmann, John Vlissides: Visualizing the Behavior of Object-Oriented systems, Proceedings of the ACM OOPSALA 1993 conference, Washington D.C., October 1993.

[21] Fraunhofer Institut für Informations- und Datenverarbeitung: The CORBA-Assistant - Monitoring of CORBA-based Applications, White Paper, Release 1.2, June 1997.