Department of Computer Science
Series of Publications C
Report C-2003-NN

Platform experiences

Juha-Pekka Haataja, Janne Metso, Timo Suoranta, Markku Vähäaho,

Egil Silfver, Lea Kutvonen

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

Computing Reviews (1998) Classification: C.2.4
Helsinki 2003

Department of Computer Science
Series of Publications C
Report C-2003-NN

Platform experiences

Juha-Pekka Haataja, Janne Metso, Timo Suoranta, Markku Vähäaho,
Egil Silfver, Lea Kutvonen

University of Helsinki
Finland

Platform experiences

Juha-Pekka Haataja, Janne Metso, Timo Suoranta, Markku Vähäaho, Egil Silfver, Lea Kutvonen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Lea.Kutvonen@cs.Helsinki.FI

Technical report, Series of Publications C, Report C-2003-NN
Helsinki, January 2003, iii + 27 pages

Abstract

The Pilarcos project is researching topics related to federation establishment between autonomous systems. During the first phase of the project a reference architecture was designed and a simple prototype was implemented. During the second phase of the project, the prototype was extended by implementing more demanding application and interoperability scenarios. The goal of the advanced prototype was to demonstrate and study the feasibility of the Pilarcos architecture in a multi-platform environment.

During the prototyping efforts it was discovered that the CORBA Component Model (CCM) offers a powerful and flexible component programming model, but the CCM platforms (OpenCCM and MicoCCM) are quite immature. The platforms seem to be scalable, but they implement the CCM standard incompletely and have an annoying number of bugs. The development of CCM platforms has been quite stagnant lately and CCM's future in the middleware market is unclear.

The EJB programming model was discovered to be compact and easy to learn, but too simplified to easily support unconventional applications. For example, implementing middleware services as enterprise beans produces an unnecessary complex and non-scalable design. The JBoss EJB platform was discovered to be of very high standard, and can be recommended for production use.

Experimentation with CCM and EJB interoperability was performed and an application level adaptation model was adopted. Application level adapters are easy to implement, and their life-cycle management is simple. The down-sides of the approach are additional marshaling overhead and larger memory footprint.

Computing Reviews (1998) Categories and Subject Descriptors:
C.2.4 Computer-communication networks: Distributed Systems

General Terms:
Experimentation

Additional Key Words and Phrases:
Component models, CORBA component model, Enterprise Java Beans, OpenCCM, MicoCCM, JBoss, Interoperability

Contents

Chapter 1

Introduction

The Pilarcos project is researching topics related to federation establishment between autonomous systems. During the first phase of the project a reference architecture [4] was designed and a simple prototype was implemented [18]. During the second phase of the project the prototype was extended and more advanced application and interoperability scenarios were implemented [17]. The goal of the advanced prototype was to demonstrate and study the feasibility of the Pilarcos architecture in a multi-platform environment.

The prototype was implemented using three component platforms, two of them being implementations of the CORBA Component Model specification [9, 10, 3, 1] (OpenCCM, MicoCCM) and one being an implementation of the Enterprise Java Beans specification [15, 16] (JBoss). The interoperability between OpenCCM and MicoCCM as well as between OpenCCM and JBoss was experimented with.

The purpose of this document is to summarise the experiences gained when developing the advanced prototype. The experiences fall into three categories: experiences useful for the users of the platforms, experiences useful for developers of the platforms, and (to some extent) experiences useful for the standardisation bodies.

The document first introduces the CCM and EJB programming models along with some useful programming idioms for both. Second, an overview of the platforms themselves is provided. The overview contains information on the features, history, and background of the platforms along with descriptions of found bugs, patches to them, and subjective developer experiences. Third the interoperability model used to bridge communication between EJB and CCM platforms is further studied. Finally, short conclusions of the experiences gained is provided.

Chapter 2

Comparison of CCM and EJB models

Implementations of two separate component standards, CORBA Component Model (CCM) and Enterprise Java Beans (EJB) model, were used during Pilarcos prototyping efforts. The models have many similarities but also some fundamental differences. Each model has its strengths but also its weaknesses. The weaknesses can partly be conquered with proper design patterns and programming idioms.

2.1 Component programming

Both CCM and EJB models are component programming models. They see software as independently developed units of deployment and development. In CCM model these units are called CORBA components and in EJB model they are called Enterprise Beans.

Components live in a runtime environment called a container. Components interact with the runtime environment via a standard up-call and down-call APIs. Component container provides components with standardised access to important infrastructure services, like naming, transaction, security, and persistence services. Container also provides components with standard deployment (and configuration) architecture. Component containers themselves are instantiated in a component server. This implies that both CCM and EJB models are fundamentally server-side models. Their main purpose is to provide a standard and easy to use model for developing transactional, scalable and secure server software, using well defined design patterns and minimum possible effort. The component platforms do not prevent the usage of components to other kind of use cases. CCM, for example, is generic enough to be used for many kind of usage scenarios beyond the conventional ones.

This chapter provides a overview and comparison of CCM and EJB models based on our experiences. Some useful programming idioms are also presented for both platforms.

2.2 CCM programming model

CCM supports a very powerful programming model. CCM defines several so called ports, which include interfaces for synchronous and event based communication and attributes for component configuration. Components can be connected together using their ports to form synchronous or asynchronous multi-component assemblies. CCM defines a standard packaging and deployment model for these assemblies. Component life-cycle management is done using a standardised factory pattern. Component factories are automatically generated and they are called component homes.

The features which are totally unique to CCM programming model are the capability to define several independent interfaces for a component, and the capability to define the components dependencies from external interfaces (i.e. definition of components receptacles).

CCM runtime model is multi-threaded and thus quite generic and scalable. As a downside, the added concurrency support brings some extra difficulty to programming efforts. When multi-threading is used, the component programmer is responsible for keeping the component state stable and atomic even when there are several concurrent threads accessing it simultaneously. During our programming efforts the concurrent programming was found to be effective, scalable but error-prone (as expected).

It seems that CCM is trying to be as generic and powerful as possible but at the same time sacrificing some of the programming simplicity. Although it has its complexities, CCM programming still simplifies traditional CORBA programming considerably. When you master the CCM programming, the generic nature of the programming model allows implementation of many kinds of applications and even infrastructure services. As the future seems to bring more and more networked application environments, where the line between client and server is not clearly defined (peer-to-peer environments), the CCM kind of programming models will have advantages over the more conventional models (like EJB).

Following list summarises some features related to the CCM programming model

- CCM provides four component types: service, session, process, entity
  - Only (stateful) session components were used in the prototype
- CCM programming model is more powerful than EJB
  - CCM component may support various port types
  - CCM component may have several port instances of each port type
  - Component's external dependencies can be defined with used interfaces and event sinks
  - CCM can be used to program all kinds of server applications as well as infrastructure services
  - Nameable multiple receptacles would expand their usage scope. This would enable the programmer to select which receptacle to use at runtime. At the moment CCM provides only primitive multiple receptacles which are mostly good for primitive load balancing and fault tolerance (not, for example, storing interface references related to different application federations)
- CCM relies much on code generation
  - Code generation brings some benefits but many difficulties
  - Generated code is often un-optimised and resource consuming
  - Iterative software development requires continuous re-generation and merging of the new component skeletons to old component implementations (or giving up the usage of code generation capabilities)

2.3 CCM programming idioms

Although CCM programming model is very flexible there were some problems which we were unable to resolve with standard CCM features.

In the Pilarcos prototype we had several places where it would have been convenient for a CORBA component to be able to return a reference to one of its facets as a return value to an operation call to one of its other facets. This situation is illustrated in Figure 2.1.

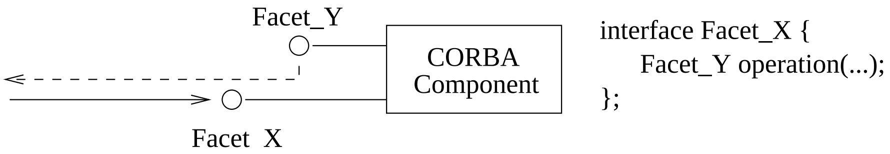

Figure 2.1: Illustration of a "facet returning a reference to another facet"-situation.

This scenario however does not seem to be possible. This stems from the fact that components cannot access their own facet-references. Component naturally has access to all the facet-implementing code and can thus call the facet-operations internally but cannot access their remotable facet references in order to return them as return parameters to an operation call.

The problem was solved with a so called "attribute idiom". In order give a component access to its facet references, the references must be attached to the component as IDL-attributes. For example in the scenario presented in Figure 2.1 an attribute of type `Facet_Y` must be added to the component IDL definition and the reference to `Facet_Y` must be configured to be the value of the attribute. The configuration must be done at component instantiation time by some external entity (e.g. a service factory).

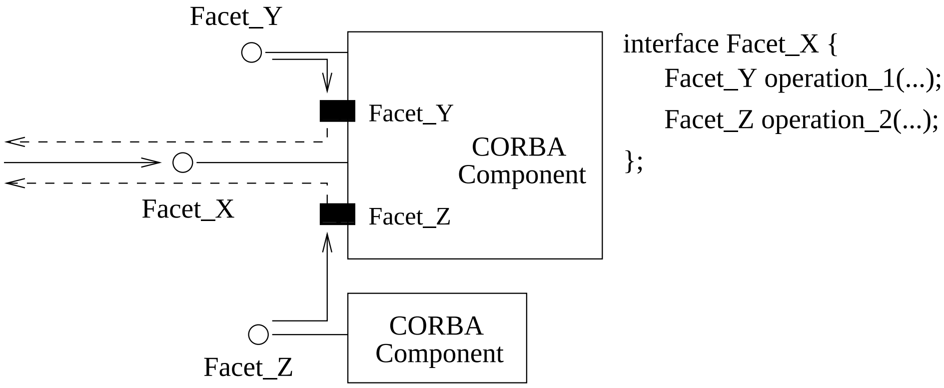

Figure 2.2: Illustration of using the "attribute idiom".

The situation is similar when a component wants to have access to facet references to other components. If these facet references are attached to component receptacles, the component can call their operations but it has no direct access to the facet reference itself and cannot, for example, return the facet reference as a return value of one of its own facets. Nor can the component give the facet as a parameter to a remote operation. The attribute idiom can be utilised in this situation also. Instead of connecting the facet of a remote component to a receptacle, the remote facet reference is attached to the component as an attribute value. Figure 2.2 illustrates two exemplary situations where attribute idiom can be used.

When using the attribute idiom it should be kept in mind that all the client-side interceptors in the CCM container will be disabled. For example OpenCCM-platform offers possibility to add so called `StubInterceptors` to a container. Every operation call made using a receptacle goes through a stub interceptor but a call made using the attribute connection goes directly to the underlying CORBA ORB.

2.4 EJB programming model

EJB supports a relatively simple and straightforward programming model. EJB bean is no more than a simple Java-class which provides a standard up-call API for container to use and which implements one or several remote Java-interfaces.

EJB does not support any other port mechanisms although programmer can define Java-attributes to the remote interfaces. They can be used for component configuration but provide no standardised division to configuration- versus usage-time epoch in the component life-cycle.

EJB defines a packaging and deployment model which is similar but more primitive than the CCM provided packaging and deployment model. The EJB model has, however, proven that it is a working model in the real-world scenarios and the advanced and complex model provided by CCM may not have as much added value as the standardisers would like to believe.

EJB supports a single-threaded execution model for components. This releases the burden of concurrent programming from the bean programmer. At the same time it reduces the amount of programmers choice for design patters. You could say that it is very easy to implement a typical server-application with EJB but if you want to implement something different or more complex you may have difficulties. For example, it is difficult to write scalable and memory effective beans for situations where you naturally should use multi-threaded programming. In these situations the bean programming loses it natural simplicity and you must learn and apply complex design patterns and even then you may not get the desired results or at least not the desired performance. However, there exists many books and articles describing working and commonly used EJB design patterns and idioms (see for example [5]).

EJB programming is naturally very magnetic since its very easy to compile and run your first Enterprise Bean, but after you have tried to do something complex like infrastructure services or multi-user adapters with the model, you definitely learn to appreciate the more powerful CCM programming model.

In the end, if your application scenarios are those typical server-application scenarios of the web-world then thats what the EJB is designed for. It does not provide a simple solution to all your problems but perhaps to the most common ones.

Following list summarises features of the EJB programming model

- EJB provides four types of beans: stateless session beans, stateful session beans, entity beans, and message driven beans
  - Stateless and stateful session beans were used in the project
  - Stateful session beans were easier to program but they offered practically no scalability

  - Stateless session beans were harder to program but they offered much better scalability. In our experience the stateful session beans seem to be more of a curiosity than a professional programming tool. Our advice is to use stateless session beans whenever no persistent state is needed and entity beans if state must be persisted
  - Stateful session beans can be modified to stateless session beans with proper programming idioms (see section below)

- EJB programming model offers only deployment time type-checking for bean interfaces
  - The bean relationship to its remote interfaces is described in its deployment descriptor
  - Whether bean actually implements all the interfaces it is supposed to, is only checked at deployment time which complicates programming
  - Software development tool support might provide proprietary type checking capabilities and ease up development

- EJB model makes doing "conventional" web-based server software quite easy
  - If used in non-intended way, complex design patterns are needed to gain performance and scalability
  - Suits well for certain classes of applications but does not suit well for some classes of applications or infrastructure service development

2.5 EJB programming idioms

In EJB domain a couple of programming idioms were used. EJB programming model differs from the CCM model in three fundamental ways. First, the EJB beans cannot provide separate facets but they must always combine all supported interfaces into one remote interface exposed to clients. Second, the EJB model does not provide any receptacles or similar mechanism to define the beans dependencies from external interfaces. Third the beans are single threaded so one bean instance cannot effectively provide service for multiple clients simultaneously. Instead a separate bean must be instantiated for each client.

To make things even more difficult the behaviour resulting from multiple clients trying to access a bean may vary among EJB servers. Some EJB servers may put the requests to a queue and service them one at a time. Some EJB servers just throw an exception to any client trying to access an already occupied bean.

No real idioms were developed for the first problem. The adopted solution was to define all beans “facets” and management related configuration interfaces as separate RMI-interfaces and then define a single interface which extends them all. This combined interface is then delegated to all parties wishing to use the bean. Each user can then cast the combined interface to the sub-interface they need to use.

Because beans lack standard receptacles, a programming idiom was defined for connecting them together. First an approach where beans find each other from a local naming service was considered. This would have required the configuration of naming service locations and name information to the beans and would have added complexity to the bean

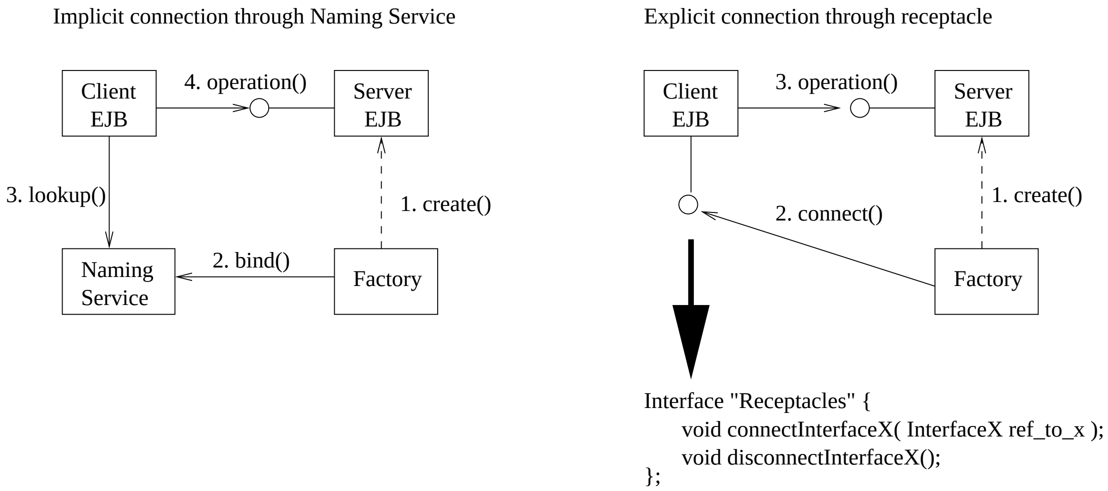

Figure 2.3: Comparing connection establishment through naming service and through receptacles.

programming. This kind of approach also makes the mutual dependencies of the beans implicit and hidden within the bean code.

Since CCM-based receptacles were seen as a good feature, custom receptacle-operations were added to the beans remote interfaces. Figure 2.3 shows how the receptacle approach differs from the naming service approach.

The problem of beans lacking multi-threading capabilities was discovered to easily become a bottleneck. At first, we tried to use stateful session beans to implement the applications and adapters. Using this approach we had to instantiate a separate instance for each client. As instances were created on-demand at runtime, the instance creation process was quickly found to be badly scalable both in memory consumption and in performance. Additional management problem was caused by the fact that stateful beans had to be removed when client did not need them anymore. We decided to move from stateful session beans to stateless session beans which are more scalable and less resource consuming.

The used EJB platform (JBoss) instantiates bean pools for stateless session beans and when someone tries to instantiate a bean, an instance is taken from the pool instead of instantiating a completely new bean. The pool size and behaviour can be configured quite flexibly but we decided to use the default configuration.

Using stateless session beans eased up the performance problems but they produced a new problem. Every time a client makes a call to a stateless bean a new instance is allocated for it. When using stateful session beans it is easy to keep track of the session state since every call from client to server goes to the same bean instance. With stateless session beans, programming idioms need to be used when implementing stateful sessions.

When using stateful beans the session state was kept in session beans instance variables. After we moved from stateful to stateless session beans we had to give up some of the object orientation and change all instance variables to be global class variables (`static` variables in Java). The class variables are common to all instances of a certain bean type and thus can be used to store the state common to all stateless beans.

Changing the instance variables to be class variables was not enough since separate clients must be able to have separate sessions with separate states. Instead of using a individual global state variables we used a global `HashMap` that maps client session id's to client specific session data. All stateless beans access this global `HashMap`. This way it is possible to store the states of each client separately from each other and access their state without interfering the states of the other clients.

Chapter 3

OpenCCM and MicoCCM

The CORBA Component Model platforms used in the Pilarcos project were MicoCCM and OpenCCM. OpenCCM was extensively used when developing Pilarcos applications and infrastructure services. MicoCCM was utilised when developing the Pilarcos trader. The interoperability of the platforms was also tested. This chapter provides a thorough view of the platforms as well as our experiences with them.

3.1 Platform introduction

In this section, the main features and status of the MicoCCM and OpenCCM platforms are introduced, and reasons for choosing them as Pilarcos prototype platforms are discussed.

3.1.1 MicoCCM

MicoCCM [7] is an open source C++ implementation of the CORBA Component Model, licensed under the GNU Lesser General Public License (LGPL). It is based on the well-known Mico ORB, and has been ported to a wide variety of platforms, including Linux and several Unixes as well as Windows.

The aim of the MicoCCM project is to provide a reference implementation of the CCM. Work on MicoCCM began in October 2000 in a separate project. The version used in the Pilarcos prototype is a snapshot from April 15, 2002 (`20020415`). Since then, MicoCCM has been integrated into Mico, which is available for download from the Mico web site [6]. In addition to frequent releases, anonymous CVS access is also available.

MicoCCM is complete with respect to the planned functionality: full support for extended service and session components. This includes

- an IDL3 compiler;
- all types of ports, including multiplex receptacles and event sources and sinks;
- service and session containers;
- monolithic and locator-based component implementations;
- stand-alone and shared-library components; and
- tools to install, host and manage component implementations.

All MicoCCM features are in accordance with the OMG specifications.

A separate Assembly and Deployment Toolkit is available for MicoCCM. The toolkit includes a graphical assembly tool, which reads components from CORBA Software Packages. The designer can graphically connect components to form a CORBA Assembly, and package the assembly into a CORBA Assembly Archive. Assembly archives can be automatically deployed on target hosts using the deployment tool from the toolkit. OMG's XML-based file formats are used for the deployment architecture. The toolkit was not used in the Pilarcos prototype, because it is MicoCCM specific, and we needed to use a combination of MicoCCM, OpenCCM and EJB components.

MicoCCM was chosen for the Pilarcos prototype because it was the most mature CCM implementation available in spring 2002. For the performance-critical Pilarcos trader, it was also important that MicoCCM supports components written in C++.

3.1.2 OpenCCM

OpenCCM [11] is also an open source CCM implementation licensed under the LGPL, but written completely in Java and supporting only components written in Java. It needs a separate CORBA 2.4 compliant ORB, either Borland Enterprise Server 5 for Java, ORBacus 4.1 for Java or Community OpenORB 1.3.0. OpenCCM is nowadays developed under ObjectWeb consortium [8]. We chose OpenORB [12], because it was the only open source implementation.

OpenCCM development began at LIFL in the year 2000. The currently available version is OpenCCM 0.5, released in December 2002. In spring 2002, however, the latest publicly available version was 0.2, which was over a year old. At an explicit request, we received an unofficial 0.3.2 version, which was used for the Pilarcos prototype.

Feature-wise, OpenCCM 0.5 is very similar to MicoCCM, with the following main differences:

- OpenCCM has a basic, non-standard container framework that supports plug-ins. (A new container framework is planned.)
- OpenCCM does not provide a complete Component Implementation Framework (CIF), but has support for monolithic and segmented implementation strategies.
- OpenCCM does not have a graphical assembly tool.
- MicoCCM supports only components written in C++; OpenCCM supports only components written in Java.

Except for the above items, current OpenCCM versions follow the OMG specifications.

Although OpenCCM is less mature and less robust than MicoCCM, OpenCCM was used for most of the Pilarcos prototype components because it was the only open source CCM implementation for Java. In addition, the first Pilarcos prototype [18] used OpenCCM version 0.2, and we were familiar with it.

3.2 Documentation and getting started

In this this section we take a closer look to available MicoCCM and OpenCCM documentation and provide a short guide on how you can get started with software development.

3.2.1 MicoCCM

Documentation for MicoCCM is sparse. A short tutorial and a manual for the MicoCCM tools are available on the MicoCCM web site. For the Mico ORB, more comprehensive documentation is available, but even that does not cover all of the functionality and is slightly outdated in places.

Installing and compiling MicoCCM on Linux was relatively easy, although not all configuration options worked as advertised. Difficulties began to arise when we tried to write and compile our own CCM application. Because of the scarcity of documentation, we had to rely heavily on examples supplied with MicoCCM for information. The complete CCM compilation process is not simple, and has several phases:

1. Generating stubs and marshaling code from the IDL files using the `idl` compiler.

2. Generating component-specific skeletons from the IDL files using the `mico-ccm` generator.

3. Compiling the stubs, the component-specific skeletons and the actual implementation classes with the C++ compiler.

4. Linking the resulting object files together.

Unfortunately, the makefiles accompanying the examples were very tightly tied to the MicoCCM distribution. Thus, a makefile that performed the above process had to be written practically from scratch.

The example programs were also convoluted, partly because they did not use the assembly and deployment toolkit. Also, MicoCCM does not provide a generator for implementation classes. They must be written from scratch according to specific rules. For these reasons, it took several days to create a functional makefile, a session component without any functionality and a custom deployment application for it.

3.2.2 OpenCCM

We used the unofficial OpenCCM version 0.3.2, which had practically no documentation. Compilation and installation was done largely by inspecting the build files and by trial and error. Although the situation has improved — OpenCCM 0.5 now has an installation guide — the guide is still incomplete, and OpenCCM will not install properly according to the instructions.

After installation, the OpenCCM user's guide covers enough to make it possible to compile and run the examples accompanying the distribution. The user's guide is not complete enough for learning to write OpenCCM applications; the examples must be studied instead. The new Getting Started with OpenCCM Tutorial is helpful in this, and aids in understanding the development process.

Examples and build files are tightly tied to the OpenCCM distribution, and are rather difficult to use as basis for custom projects. However, the Apache Ant build system used by OpenCCM is easier to use for this purpose than the GNU Make we used with MicoCCM.

OpenCCM's `ir3_jimpl` generator used to generate monolithic component implementation classes was faulty in OpenCCM 0.3.2, and produced incomplete code. The `create` method body in the generated homes was empty. Also, `get_<facet>` methods were completely missing from the generated implementations. After the initial confusion caused by these faults was cleared up, writing and compiling a simple session component was straightforward.

Although OpenCCM 0.5 does have a deployment tool, and the examples now use it, OpenCCM 0.3.2 did not have one. Creating a deployment application from scratch was a tedious task, but was possible by taking advantage of the examples and our previous experience with OpenCCM 0.2.

3.3 Development experiences

The Pilarcos II prototype is quite large and complicated, and may be among the most complicated CCM applications created to date. The IDL definitions are 1550 lines long, and consist of 16 components, 29 interfaces and 30 structures in 12 IDL files. Code lines total over 23000, of which more than 11000 are non-comment, non-blank lines. Developing such a large application clearly highlights the strong and weak points of the middleware platforms. These experiences are discussed in this section.

3.3.1 MicoCCM

The Pilarcos trader and type repository components were written in C++ for the MicoCCM platform. The trader and type repository IDL definitions were in two separate IDL files, which were compiled separately, along with an IDL file containing definitions common to the entire Pilarcos prototype.

For each of the three IDL files, two pairs of header and source code files are generated by the MicoCCM tools. One pair contained the Mico stubs, the other contained CCM component definitions and skeletons. For a single session component with a standard home, MicoCCM generates three skeleton and ten stub classes. In our case, the generated files were about 22700 lines long in total. Compiled to object files without debug information and with full optimisations, they were about 2300 kB in size in all.

The actual Pilarcos trader and type repository implementations were less than half the size of the generated classes, in both lines of code and size of object files. Combined with the fact that after single change to an IDL file all of the related CCM files must be regenerated and recompiled, means that modifying and recompiling IDL definitions is a very time-consuming process. In addition, since modifications to components' interfaces also necessitate modifications to the actual implementation classes, in large projects the IDL definitions should be stabilised as early as possible to save work and compile time. This has always been the case with CORBA software, but the CCM with its plethora of generated classes emphasises this point even more.

After the IDL had been stabilised, MicoCCM performed reliably. The problems in CORBA development with MicoCCM are related to the OMG IDL to C++ mapping in general, not to CCM per se. The C++ mapping is outdated and unnecessarily complicated [14]. Development and debugging is slow, because both compiler error messages and CORBA runtime exception messages are either hard to interpret or simply do not give enough indication about the actual cause of errors. The use of CORBA's `Any` type is a frequent point of failure when developing complex data structures, as memory leaks and crashes can easily occur. In a few cases, even after extensive investigations it was not clear whether the problem was in application code or in Mico.

An issue arose when implementing components that provide several interfaces. MicoCCM relies on a non-standardised technique for implementing monolithic executors that support multiple facets. A local mix-in interface must be defined in the IDL that inherits from the equivalent executor interfaces of the facets. The actual implementation class then implements this mix-in interface. An example is shown in Figure 3.1. This technique is allowed by the CCM standard, but is not adequately documented in the MicoCCM manuals.

```
local interface TraderMixin : Pilarcos::Trading::CCM_TraderComponent,
                              Pilarcos::Trading::CCM_OfferRegistration,
                              Pilarcos::Trading::CCM_Lookup,
                              Components::SessionComponent {
};
```

Figure 3.1: Example of MicoCCM mix-in interface for multiple facets.

Originally, we tried to make the Pilarcos trader and type repository (CCM) service components. However, we did not succeed, either because of a fault in the MicoCCM code generator or a mistake caused by the inadequate documentation. Switching to session components fixed the problem.

A small buglet was also found in the Mico IDL compiler. To be able to use sequences in IDL operations, the IDL compiler required that a `typedef` for the sequence is used. In other scenarios, the MicoCCM tools handled our nested modules and multiple IDL files without problem.

The MicoCCM component server daemon is multi-threaded and by default assumes that components are thread-safe. This was not documented at the time we started using MicoCCM. After the necessary thread-safety corrections were made to our code, the MicoCCM runtime performed well even in a distributed and heavily loaded configuration.

In our opinion, despite the glitches, the ground is laid for MicoCCM to become a mature and generally usable CCM platform for C++ if its development continues actively.

3.3.2 Mico trader

Although not part of the CCM, the Mico trader is a CORBA service included with Mico, and is thus discussed here. Initially, the Pilarcos trader used Mico's implementation of CORBA Trading Object Service to maintain service offers. Later on the Mico trader turned out to be inadequate because of features which were not implemented at all or which did not work as advertised.

The constraint language used to qualify service offers for lookup query in the CORBA trader had several problems. Neither `and` nor `or` operators of the constraint language worked properly when the constraint expression had several operators. It appears that the implementation has a fixed maximum length for the constraint expression such that longer expressions simply do not work correctly. It also appears that the implementation only supports a small subset of CORBA types, for example, it has no support for double floating point data type.

The CORBA trader has a trader policy specifying how many service offers are returned at most by default. For the Mico trader this policy appears to have a fixed value of 100 offers without any means of changing it.

We were able to fix `and` and `or` operators and work around the missing double data type. While the other bugs could have been fixed too, we suspected that the Mico trader implementation could have even more bugs and missing features. Because of this we decided to use the ORBacus trader implementation instead. It turned out to work well as we did not encounter any problems with it.

3.3.3 OpenCCM

Most of the Pilarcos prototype — 14 of 16 CCM components — was written in Java 1.4 for the OpenCCM 0.3.2 platform. A multitude of bugs and unimplemented features were found in the compilation chain as well as the runtime system.

From the IDL definitions of the entire Pilarcos prototype, the OpenCCM compilation chain generates 1135 stubs and skeleton classes. For a single session component with a standard home, six skeleton and 52 stub classes are generated. Most generated classes are four kilobytes in size when compiled to Java bytecode. In total, the generated classes for a session component and its home are about 480 kB. The compiled Java classes of the entire Pilarcos prototype amount to 7.6 MB uncompressed, of which 7.1 MB is OpenCCM generated classes.

Since an IDL change results in the re-compilation of all IDL definitions in our configuration, IDL changes are expensive in development time. It is also difficult to keep code in sync with a frequently changing IDL definitions, and some IDL modifications may necessitate changes all over the application code. This is a major point against the CCM development model as compared to the EJB model. If the IDL is stable, CCM programming in Java is fairly simple and enjoyable. This mostly due to the clean and modern OMG IDL to Java mapping.

It appears that OpenCCM has not been tested on large applications at all. The `ir3_idl2` and `ir3_java` IDL compilation tools take only one IDL object as parameter for code generation. This makes using IDL files that contain multiple modules next to impossible. As a result, in the Pilarcos prototype IDL files had to be partitioned according to IDL modules. Also, the OpenCCM 0.3.2 interface repository crashed when the IDL file fed into it using `ir3_feed` contained nested include directives. This bug was easy to circumvent by avoiding nested inclusions.

The IDL3 to IDL2 compiler, `ir3_idl2`, messed up the order of declarations and definitions when modules were opened multiple times in the original IDL3 file. For this reason, we had to put all definitions within an IDL file into a single module. This breaks modularity in some places.

The `ir3_idl2` compiler generated IDL2 that does not allow for forward declaration of provided or used interfaces. This, combined with the nested inclusion bug, made partitioning IDL definitions sensibly quite difficult. These problems were circumvented by a complicated Apache Ant build file that concatenates all separate IDL definition files into one big file before feeding it to the OpenCCM IDL compiler.

OpenCCM does not fully implement all the standard features it claims to implement. Several part of the container implementation are only partially implemented. For example component instance removal operations are only partially implemented which results in memory leaks.

The OpenCCM 0.3.2 runtime allows concurrent execution of application components when using CORBA ORBs capable of multi-threading, but is not itself thread-safe. Concurrent operation calls randomly cause an exception to be thrown on the server side before the execution even reaches application code. We circumvented this problem by modifying the relevant generated files by hand. However, adding general thread-safety to OpenCCM 0.3.2 without losing the multi-threading capability would have required structural changes to the OpenCCM runtime and code generation tools.

3.3.4 OpenORB

As the underlying CORBA ORB for OpenCCM, we initially used OpenORB version 1.2.1 and later switched to version 1.3.0 beta 1. The switch was made because of a wide character code set bug in the earlier version, which prevented interoperability with Mico. With the newer OpenORB version, no interoperability problems arose.

When OpenORB 1.3.0 beta 1 was used in a distributed configuration with more than seven clients per server, severe probably concurrency related problems were encountered. For some reason all threads in execution were deadlocked inside OpenORB. Therefore the maximum number of concurrent clients we were able to use was limited to seven.

OpenORB implements direct collocation optimisation which enables direct Java-calls between objects if they are collocated in the same virtual machine. This leads to several changes in the operation call semantics [19] which must be taken into account in application designs.

In our prototyping effort two semantic changes were encountered. When using CORBA `oneway` calls with collocated objects the oneway semantics are not supported and call becomes a normal synchronised Java-call. Another semantical change happens when IDL-defined data structures are passed between collocated objects. In a distributed environment ORBs pass data structures (for example, IDL `structs`) by value, but in case of collocated Java objects the data structures are passed by reference. This may cause subtle side effects, since modifications to the data structure by the callee become visible to the caller.

Implicit collocation optimisations that cause semantic changes are potentially dangerous. Programmers of CORBA applications normally do not know how the application will be distributed. To avoid these problems, collocation should either be explicitly visible or collocation-safe stubs should be used in all cases.

Chapter 4

JBoss

In addition to the introduced CORBA component model platforms (OpenCCM, MicoCCM) an Enterprise Java Beans platform was used. JBoss (formerly known as EJBoss) was selected as the EJB platform to be used. One of the reasons for the selection was the well-established position of JBoss in the J2EE application server market. In addition the JBoss was considered to be the most advanced of the open source J2EE platforms. All JBoss development was performed on Computer Science Department's Linux system.

4.1 Platform history and status

JBoss development started in March 1999. At first the JBoss developers wanted to build a simple EJB container but quickly the ambitions went higher and today JBoss is an award-winning (JavaWorld Editors' choice 2002 Award for best Java application server) full-fledged J2EE application server with rich set of additional features.

JBoss is now at 3.0 series (latest release is 3.0.4 and 4.0 implementation is underway) and is developed by an active developer community of around 100 members. JBoss is downloaded almost 200.000 times a month from the JBoss website [2]. JBoss is developed under GNU Lesser General Public License (LGPL) and is available in source code and binary format without fee.

4.2 Documentation and other resources

In the Pilarcos project we used JBoss version 3.0.0. At the time we installed the platform there was no up-to-date documentation freely available. Commercial documentation could have been bought from the JBoss website but we decided to try how well we could do without the commercial services.

The freely available documentation was in version 2.4.4 and was mostly outdated. It was quickly discovered that in order to be able to use any of the advanced features of JBoss, the commercial documentation would have to be bought. We only needed to use a small subset of all JBoss features, so we managed to survive without buying the official documents.

In addition to the official documentation, JBoss has two mailing lists. One of the lists is targeted for JBoss users (i.e. application developers) and the other is targeted for JBoss platform-developers. The mailing lists seemed to be very active and anyone joining them should be prepared to receive even tens of mails every day. We didn't have any major problems using JBoss nor did we discover any bugs. For this reason the mailing lists were of little use to us. In product environments with more complex usage scenarios and tight deadlines they could provide great help to acute situations.

According to JBoss website, commercial services as support, consulting, and training are available for JBoss users. This commercialised service provisioning system enables JBoss platform-developers to get extra income for working with JBoss and motivates the development community to stay active. For JBoss users it offers value-added services not that often found in other open source platforms.

In general the JBoss website has a professional flavour and, in addition to the above mentioned items, offers other interesting services. For example many EJB-developers might be interested in the migration tool (and guide) for migrating BEA Weblogic-based EJB applications to JBoss environment.

4.3 Usage experiences

Starting to use JBoss is very easy. When using the binary distribution (as we did) basically what you have to do is to unpack the JBoss to some local directory and set the environment variable `JBOSS_HOME` to point into that directory.

JBoss offers three different example configurations which can be used. If you have the commercial documentation available it should be easy to add your own custom configurations. Without the commercial documentation it is a tedious task, however, due to the large amount of data in the JBoss distribution. JBoss looks more like an operating system than an application server. For example the JBoss start-up time with default configuration is over 30 seconds in a computer with 1.3 GHz Pentium processor and 512 megabytes of RAM.

JBoss offers an innovative and very easy-to-use deployment system for plugging in platform services and user applications. Users can configure JBoss to use one or more deployment directories (or URLs) from where the server looks for deployable services at start-up time.

In addition to start-up deployment, JBoss offers a possibility for using hot-deployment. When hot-deployment is active JBoss continuously scans the deployment directories for new services. If a assembly of beans is added to some of the directories (or URLs) JBoss automatically deploys it and makes it available in the network. If the assembly is removed from the directory JBoss automatically un-deploys the service. This feature was found very useful and easy to use.

JBoss implementation seemed very robust and well tested. Not a single bug could be found during the prototyping efforts. Only two negative issue were discovered. One of them was the large start-up time and memory footprint of the default configuration. By customising the configuration, both of them could be minimised. This, however, it is not a completely trivial task and requires the commercial documentation and some developer effort.

The second problem was the in-ability to easily build a multi-developer environment where JBoss platform is installed into one shared directory but all developers have their own code base of the applications they develop. OpenCCM make this kind of configuration easy by supporting a primitive "configuration repository". This makes it possible to install OpenCCM in one shared directory but have private configuration and deployment directories for each developer. With JBoss the easiest working solution found was to install JBoss separately in each developer machine.

Chapter 5

EJB/CCM Interoperability

One of the work subjects in the Pilarcos prototype was to experiment with the interoperability of EJB and CCM platforms. The interoperability scenarios were implemented in the context of the `Tourist Info Service`-case (see [17] for case description).

The interoperability model had following requirements placed upon it

- It should support utilising pre-registered interface descriptions in a global type repository
- It should support descriptions of public interfaces in several different technology formats
  - In the prototype, IDL was the only format used in specification of inter-domain interfaces
  - In intra-domain adaptation also RMI remote interfaces were used
- It should support the implementation of client and server applications either as CCM or EJB applications but all domains must be able to use/support the interfaces described in the global type repository (despite the application platform they use)

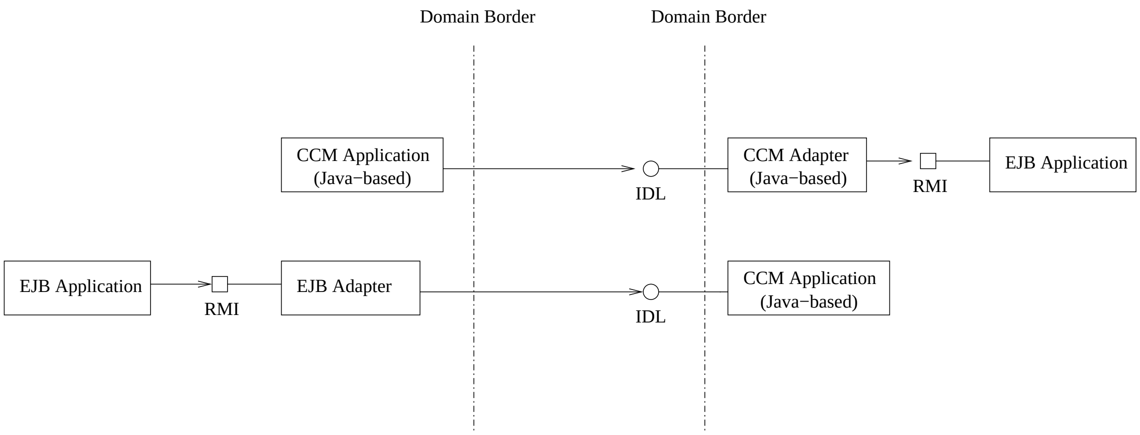

Figure 5.1: Prototyped interoperability scenarios.

From these requirements we can derive the interoperability scenarios of the prototype. The scenarios are shown in Figure 5.1. The figure shows the two implemented inter-domain interoperability scenarios

- OpenCCM application to JBoss application (CCM to EJB interoperability)
- JBoss application to OpenCCM application (EJB to CCM interoperability)

The interoperability requirements could be supported with interface-level conversions which remove the need to do component model adaptation. The rest of the chapter discusses the interoperability model used to implement the interoperability scenarios.

5.1 Interoperability model

It was decided that the interoperability solutions were to be implemented on the application level rather than embedded into the infrastructure. This decision was made from the following reasons

- Implementing the interoperability adapters at application level simplifies their implementation
  - Adapters fundamentally become special purpose applications and anyone capable of implementing applications is capable of implementing adapters
  - Adapters can use underlying middleware in a standard way without need to implement their own communication layer or integrate with the middleware with possibly complex and proprietary means
- Implementing the infrastructure services needed to manage the adapter life-cycle requires less effort if adapters are at application level
  - Same principles can be used to implement both adapter and application life-cycle management
  - When adapters are not only application level instances but also components, features available for component life-cycle management can be utilised in adapter life-cycle management
  - Pilarcos infrastructure aims at dynamic life-cycle management and implementing this with infrastructure level adapters is very difficult (partly because of limited platform support available)

5.1.1 Application level adaptation model

The main driving forces of the application level adaptation model were the simplicity of adapter and life-cycle service implementation and support for dynamic nature of the Pilarcos architecture.

The down-sides of the application level adaptation model are resource consumption and un-optimal performance. The resource consumption results from the fact that application level adapters consume as much resources, for example memory, as do normal applications. How big the resource consumption is depends on the application programming model used. With some light-weight model the consumption can be quite restricted but when, for example, component models are used it cannot be that optimised.

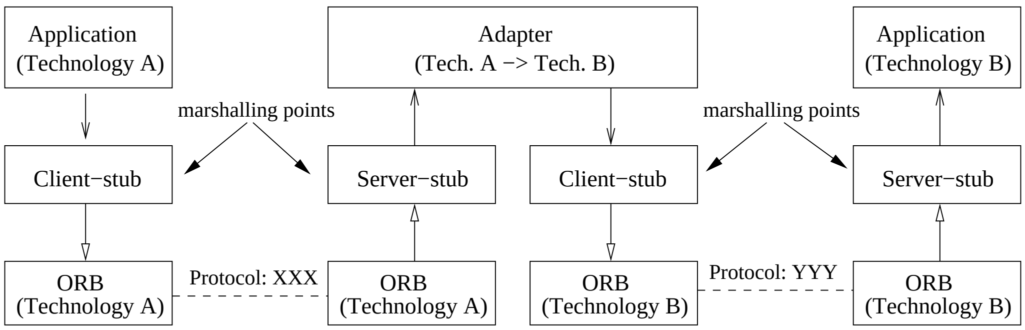

Figure 5.2: Marshalling overhead caused by application level adaptation.

The un-optimal runtime performance is mostly caused by the overhead related to additional marshaling. Figure 5.2 shows the additional marshaling caused by application level adaptation. How much this affects the overall performance depends on the application scenario and the platforms used. If the amount of work performed by applications is relatively small but the amount of information moved between applications is relatively large, then the marshaling overhead can be considerable. However, in these cases the applications tend to be so trivial that it is often easier to implement a completely new application than to implement an adapter for an old one. The marshaling overhead is also reduced if the applications and adapters are deployed in the same server process and the used platform implements collocation optimisations (as CORBA ORBs often do).

5.1.2 Other considered adaptation models

The stub-level adaptation model was considered as a competing approach. The stub-level adaptation model would have been more resource and performance effective than application level model. In the stub-level adaptation model the adapters are not applications but they are instead implemented as a thin layer between the underlying ORB and the actual applications. This adaptation layer could be implemented as advanced client- or server-side stubs or as other lower-level adaptation objects separated from the stubs and the ORB.

Stub-level adaptation would result in smaller adapter footprint and reduced amount of marshaling. Figure 5.3 illustrates an example situation of stub-level adaptation. As can be seen, the amount of marshaling points is reduced.

There were several reasons why the stub-level adaptation model was not considered a good approach in this project. One of the main reasons was that the available platform support for this approach was minimal at the decision making time. RMI-IIOP [13] was the only available approach that was even close to the intended model and used platforms did not even fully support RMI-IIOP. RMI-IIOP relies on adapter generation via static, standardised mappings. With this kind of approach only a minimal subset of the adaptation model proposed by Pilarcos architecture could be supported. In addition, the management possibilities for RMI-IIOP based adapters did not offer enough flexibility and support for needed dynamics. Implementing the intended adapter and application life-cycle modes would have been also been a tedious task.

Final and maybe the most serious drawback of stub-level adaptation model is the inability to fully support two-way interoperability. In the stub-level adaptation approach

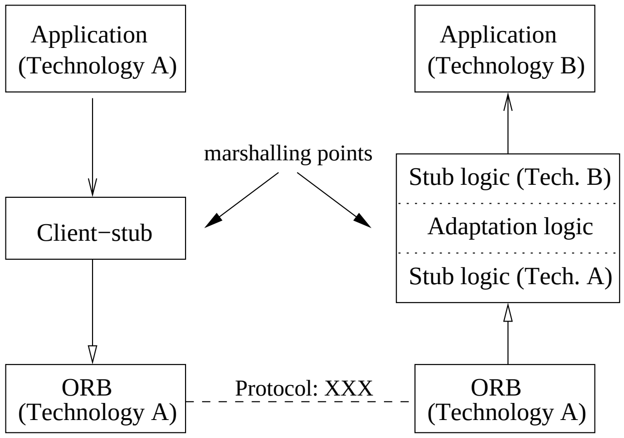

Figure 5.3: Example scenario of stub-level adaptation.

the messaging protocol used between domains cannot be adapted. Since a messaging level protocol is always bound to a specific distributed object model (IIOP to CORBA and JRMP to RMI) and to a specific type system (IIOP to IDL and JRMP to Java) only a subset of one technology could be adapted to another technology. In RMI-IIOP case this is manifested by the fact that RMI-developers wishing to use IIOP must restrict themselves to use only a subset of Java-types (those mappable to IDL). In Pilarcos approach no such restrictions were allowed but a more generic adaptation system was needed.

5.2 Implementation considerations

When implementing the application level adaptation scheme two issues needed to be decided. First the adapter implementation technology needed to be selected and second the bootstrapping mechanism for connecting client to the server needed to be defined (the latter depends on the former).

5.2.1 Adapters in single and multi-language environments

If an adapter makes adaptations from technology X to technology Y it fundamentally becomes server in the technology domain X and client to the technology domain Y. Since we had to implement both CCM to EJB and EJB to CCM adaptation scenarios we needed to have both, CORBA based adapters (CCM to EJB scenario) and Java RMI based adapters (EJB to CCM scenario). Since we were using component platforms it was most convenient to implement the CORBA based adapters as CCM components rather than stand-alone CORBA servers and RMI adapters as EJBs.

We were able to implement all the adapters as components since all our platforms were Java-based. If we would have built interoperability from MicoCCM based CCM applications to EJB applications, the CORBA-based adapters would have had to be small stand-alone Java-based CORBA servers running in different process than the 'C++'-based MicoCCM components.

If you are using application level adapters and have differences in both the programming language and the component model you have to first adapt the differences in the

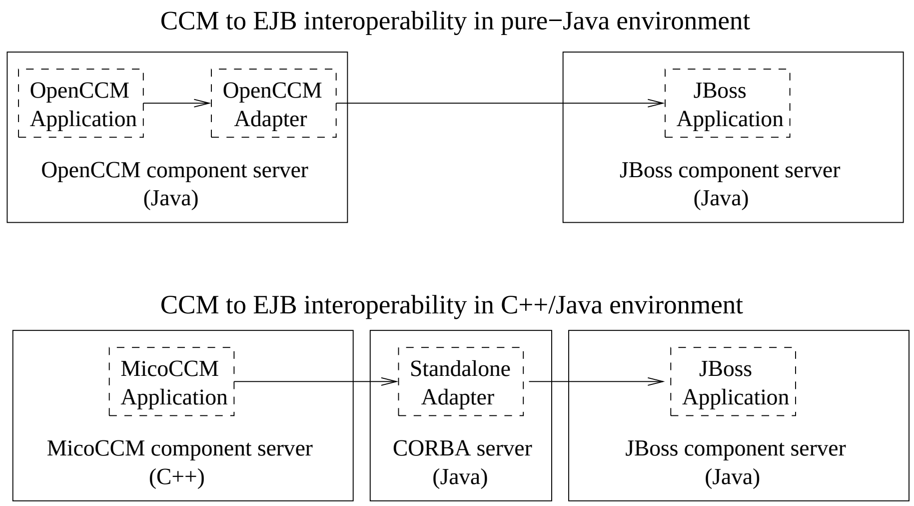

Figure 5.4: CCM to EJB adaptation in pure-Java and hybrid C++/Java environment.

programming language environment and then the differences between the middleware environments. Figure 5.4 shows a pure-Java and a hybrid C++/Java adaptation scenarios.

5.2.2 Bootstrapping in multi-technology environment

The bootstrapping problem is related to the fact that different middleware technologies have different object reference formats and different ways to bootstrap them for the client.

A very often used pattern for bootstrapping is to bootstrap the object references from a naming service. Naming service bootstrapping has two benefits. First an object reference stored in a naming service can easily be presented as two strings. One string (an URL for example) for identifying the location of the naming service and one string for identifying the object reference stored within the naming service. This way the referencing information can be easily stored and carried over domain boundaries even when technology boundaries are crossed. The second benefit of the naming service usage is the fact that the object reference becomes in-direct thus loosening the binding between client and server. The object instance bound to the naming service can be changed at any given time making, for example, software upgrades easier.

The drawback of the naming service usage is that the added value of in-direction comes with a performance penalty. Client has to make an additional request to the naming service in order to receive the concrete object reference.

It was assumed that the performance penalty caused by additional naming service lookup would be small enough to be tolerable. The actual lookup procedure was indeed quite fast as expected. What came as a surprise was that connecting to a naming service and receiving a root naming context took excessive amounts of time. Sometimes receiving the root context took as much time as the whole Pilarcos federation establishment procedure itself (see [17] for more information on federation establishment). This was true for both used naming service APIs JNDI (Java Naming Directory Interface) and COSNaming (CORBA Naming).

In order to optimise the naming service usage we had to implement a couple of additional features. We implemented a light-weight naming service (called federation-aware naming service [17]) which allows bootstrapping of 'native' object references very effectively. Native object references in our prototype are CORBA object references (IORs) since our CORBA based Pilarcos infrastructure can handle them in their native format. In the prototype all the inter-domain references were CORBA references since IDL was used as the global interface specification language. For this reason we achieved great performance improvements with the optimisations.

Performance improvements are gained because the references of the federation-aware naming services in each domain are carried in every federation contract. These references can be seen as "root contexts" and are thus available to all domains without any additional connection establishment delay.

If the inter-domain references are not CORBA references but, for example, RMI references then the above optimisation bring no performance gains. To improve the performance of 'non-native' cases a caching approach was adopted. Since the initial naming context retrieval is the performance bottleneck here, then caching the initial contexts retrieved from other domains offers a partial solution.

With the caching optimisation, the first contact to a foreign domain is un-optimised, but the following ones are fast. In cases where domains interact regularly, the performance improves considerably. For example in business cases like the one used in Pilarcos (see [17]), where there exists general purpose services (i.e. payment services), the optimisation is useful.

In general, many business-to-business scenarios contain situations where same trading partners interact quite often (i.e. tourist service making hotel reservations to same hotel services over and over again). In all these scenarios the 'non-native' bootstrapping case can be optimised to be almost as fast as the native one.

Chapter 6

Conclusions

This document reports the experiences gained when implementing the Pilarcos prototype [17]. Prototype was developed using three component platforms, two CCM platforms (OpenCCM, MicoCCM) and one EJB platform (JBoss). Interoperability of OpenCCM and MicoCCM as well as interoperability of OpenCCM and JBoss were experimented with. This chapter collects the experiences reported in this document and combines them into short conclusions.

When using CCM and EJB to implement Pilarcos infrastructure services and applications it was quickly realised that CCM provides a more powerful and scalable programming model. CCM has many useful programming features, such as a rich collection of different component ports, as well as powerful runtime features like multi-threading. EJB programming model, instead, offers a simpler programming model which is easier to learn and which works well in many typical service provisioning scenarios. The lack of description capabilities of components external dependencies and limited run-time model cause that EJB usage outside its direct target use cases easily becomes un-flexible and non-scalable.

EJB platforms (e.g. JBoss) are very mature and rapidly developing. They have been successfully used in many real-world scenarios and have an established and growing user and developer base. CCM platforms, on the contrary, are non-mature and their momentum has decreased rapidly from the time CCM standard was first published (1999-2000). CCM platform developer base still exists and is active but it has not grown as expected and instead of gaining more success over time the CCM position in the market has been mostly stagnant.

EJB and CCM interoperability was experimented with OpenCCM and JBoss platforms. Application level adaptation was adopted as the preferred adaptation approach. Advantages of application level adaptation were the simplicity of adapter implementation, simplicity of adapter life-cycle management, and conceptual similarity with application development (application developers have no problems understanding the adaptation mechanisms). The advantages of the application level adaptation model made it easier to answer to the requirements for dynamic behaviour, placed by Pilarcos architecture [4].

The drawbacks of the application level adaptation approach are sometimes un-optimal performance (mainly because of marshaling overhead) and un-optimal resource consumption. Since the performance overhead was not considered significant in our business scenario and Pilarcos prototype was not targeted to resource critical devices, the drawbacks were considered acceptable.

In conclusion it can be said that as a programming model, CCM is conceptually superior to EJB. EJB, however has proven itself in many real-world scenarios and, despite many of its short-comings, can be considered a decent model. EJB platforms are of very high quality and have gained a very significant share of application server markets. Whether CCM will ever be a success in large remains to be seen. CCM platforms are developing slowly but CCM's many advanced features and status as an Object Management Group (OMG) specification give hope of a better future.

As CORBA can fundamentally be seen as an integration platform targeted mainly for tightly-coupled distributed applications, CCM tries to establish a universal software development model for multi-language programming environments. This task is by far more difficult. A more fruitful approach might be to promote integrating the most useful CCM features to EJB model and make an effort to push CCM to be a de-facto component programming model in C++ world. This would result, not as two competing component models, but with two similar, CORBA compliant, and easily inter-operable component models with separate target groups. Time will tell whether CCM will ever be truly popular among C++ and/or Java developers.

Bibliography

[1] Haataja, J., Silfver, E., Vähäaho, M., and Kutvonen, L. CORBA Component Model – status and experiences. Tech. rep., Dec. 2001. C-2001-65.

[2] JBoss web site. `http://www.jboss.org`.

[3] Kutvonen, L., Haataja, J.-P., Silfver, E., and Vähäaho, M. Evaluation and exploitation of CORBA and CCM technologies. Tech. rep., Mar. 2001. C-2001-11.

[4] Kutvonen, L., Haataja, J.-P., Silfver, E., and Vähäaho, M. Pilarcos architecture. Tech. rep., Mar. 2001. C-2001-10.

[5] Marinescu, F. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* 2002. PDF version at http://www.jdon.com/books/ejbdesignpatterns.pdf.

[6] MicoCCM web site. `http://www.fpx.de/MicoCCM/`.

[7] Mico web site. `http://www.mico.org`.

[8] ObjectWeb consortium web site. `http://www.objectweb.org`.

[9] Object Management Group. *CORBA Component Model - Volume 1.* Framingham, MA, USA, 1999. OMG Document orbos/99-07-01.

[10] Object Management Group. *CORBA Component Model v3.0.* Framingham, MA, USA, 2002. OMG Document formal/02-06-65.

[11] OpenCCM web site (LIFL). `http://www.lifl.fr/OpenCCM`.

[12] OpenORB web site. `http://www.openorb.org`.

[13] Java RMI over IIOP Technology Home Page. `http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop`.

[14] Schmidt, D. C., and Vinoski, S. Standard C++ and the OMG C++ mapping. *C/C++ Users Journal* (Jan. 2001). Also `http://www.cuj.com/experts/1901/vinoski.htm`.

[15] SUN Microsystems, Inc. *Enterprise JavaBeans Specification v1.1*, 1999.

[16] SUN Microsystems, Inc. *Enterprise JavaBeans Specification v2.0*, 2001.

[17] Vähäaho, M., Haataja, J.-P., Metso, J., Suoranta, T., Silfver, E., and Kutvonen, L. Pilarcos prototype II. Tech. rep., Department of Computer Science, University of Helsinki, Jan. 2003. C-2003-NN.

[18] Vähäaho, M., Silfver, E., Haataja, J., Kutvonen, L., and Alanko, T. Pilarcos demonstration prototype – design and performance. Tech. rep., Dec. 2001. C-2001-64.

[19] Vinoski, S. Collocation optimizations for CORBA. *C++ Report 11*, 9 (Oct. 1999). Also `http://www.iona.com/hyplan/vinoski/col18.pdf`.

Helsinki 2003