CORBA Component Model - status and experiences

Juha Haataja, Egil Silfver, Markku Vähäaho and Lea Kutvonen

Helsinki December 31, 2001
UNIVERSITY OF HELSINKI
Department of Computer Science

Contents

Chapter 1

Introduction

The Pilarcos project is researching topics related to federation establishment between autonomous systems. During the first year of the project a reference architecture [5] was designed and a prototype was implemented [11]. The goal of the prototype was to demonstrate and study the feasibility of the Pilarcos architecture. The prototype was implemented on top of the OpenCCM platform, which was the first available implementation of the CORBA Component Model specification (CCM) [8].

The purpose of this document is to summarise the experiences gained during prototyping. The experiences are divided into three categories: experiences of CCM standard, experiences of the OpenCCM platform, and experiences of OpenCCM based development process.

Chapter 2 of this document first provides an overview of the CCM standard. The second part of the chapter takes a deeper look at the OpenCCM platform. Finally, some code generation and memory consumption metrics are reported and results of some basic benchmarks are discussed.

Chapter 3 concentrates on the OpenCCM based development process and provides a chronological summary of the steps needed when using the platform. The focus is on the usage of the OpenCCM platform itself; an accompanying document [11] describes the implemented prototype.

Chapter 4 concludes the document. It provides a summary of the CCM standardisation effort and the experiences gained.

After reading this document the reader should have understanding of the CCM standard, its strong points and weaknesses, as well as guidelines on how to build CORBA component applications on the OpenCCM platform.

Chapter 2

Status of CCM - standard and implementations

In order to specify a sophisticated component model for the CORBA platform, the Object Management Group (OMG) started a standardisation effort a few years ago. The result was the CORBA Component Model (CCM) specification published at the end of 1999. CCM is a major part of the larger CORBA 3 “umbrella”. The original CCM specification was huge, containing almost 2000 pages (with all parts included). The size and complexity of the standard made it very difficult to understand and implement, and a relatively large number of issues were raised during the implementation efforts. Now, at the end of the year 2001, it seems that all the major issues have finally been fixed, and the first version of the standard is finalised. Implementations seem to be maturing towards industrial level strength.

This chapter provides an overview of the CCM standard as it is now, provides a review of the experiences reported by the LIFL GOAL group which implemented the OpenCCM platform, lists the available implementations of CCM, explains the features of OpenCCM, and reports the results of performance tests conducted.

2.1 Review of the standard

The CCM component model is split into two levels: basic and extended. The basic model can be used to componentise conventional CORBA objects, and is equivalent in functionality to Enterprise Java Beans 1.1. The CCM and EJB models have a deep relationship, because the original Java Bean model and later the EJB were the main source of inspiration for the CCM. The extended model brings the CCM up one level by adding functionality to the basic component model and is usually the model referenced when talking about CCM.

The CCM standard consists of five separate models. Here we summarise the models shortly and provide some discussion on their strong points and weaknesses. The discussion is based on the experiences of the LIFL GOAL group (see [6]) which implemented the OpenCCM platform, as well as on the experiences gained during Pilarcos prototyping.

2.1.1 The abstract model

The abstract model defines the semantics of components. A component is fundamentally a unit of design and deployment capable of communicating with its environment through different kinds of communication ports.

Components and their ports are defined using new IDL3 extensions. Ports are component's interaction points with other components. There are separate ports for synchronous connections and for asynchronous, event based connections. The component ports allow for the definition of the component's provided interfaces (facets) as well as connection points for components used interfaces (receptacles). The asynchronous counterparts for facets and receptacles are called event sources and event sinks. Components are monolithic in the sense that the implementations of the facets cannot be distributed physically to different component servers, and static in the sense, that it is not possible to add ports to or remove ports from a component at runtime. The ports are fixed at design time.

Receptacles can be defined to store one or multiple simultaneous facet connections. The standard does not provide any usage scenarios for the receptacles supporting multiple connections, however. One possible usage scenario would be supporting fault tolerance [6]. Simplex receptacles provide one operation (`get_connection`) to get a handle to the facet connected to it. Multiplex receptacles also provide one operation (`get_connections`) which returns handles to all facets connected to the receptacle and their associated cookie values. The cookie values usually have little meaning to the component implementor as they are intended to be used by the deployment and configuration application. In any case it would be a good idea to add fine-grained `get` operations to multiplex receptacles which would allow getting one reference at a time instead of all of them in a bulk.

In addition to the basic communication ports, it is possible to define configurable attributes for the components. The life-cycle of a component can be divided into configuration phase, where attributes are configured, and usage phase where the services of the component are used. However the standard only suggests this distinction to be made, but does not actually mandate it.

Components only support single inheritance in IDL3 even though component is basically an extension of the interface meta-type and there are no apparent reasons for not supporting multiple inheritance [6].

Finally, component homes provide a generic way to instantiate and destroy components. Homes are also defined in IDL3 which makes it possible to add home specific operations to them. Home is a completely separate meta-type from component which means that components cannot be homes. Allowing a home to be a specialisation of a component would have made it possible to create hierarchical instantiation structures.

2.1.2 The programming model

The programming model defines a Component Implementation Framework (CIF). The focal point of CIF is the Component Implementation Definition Language (CIDL). CIDL is used for defining how components interact with their runtime environment (container). As CIDL definitions may include statements describing the persistent state of a component and OMG has already specified a language, called Persistent State Definition Language (PSDL), for that purpose the CIDL should be a superset of PSDL. However, the relationship of the languages or even the syntax and semantics of CIDL were not properly defined in the first release of the standard. As a result, the generation of component implementations is under-specified which hampers component portability across containers from different platform vendors.

2.1.3 The packaging model

The packaging model defines how components are packaged into units suitable for automatic deployment. The packaging specification is based on the Open Software Description (OSD) model and is realised as an XML-DTD. The packaging model enables the definition of component packages for single components and component assembly packages for several interconnected components. The OSD based XML-DTD is very flexible but also quite large and difficult to understand. There is no tool support yet for defining the descriptors and none of the existing CCM platforms actually support deployment based on those descriptors.

2.1.4 The deployment model

The deployment model defines how to automatically deploy and instantiate the component based applications in distributed servers. The deployment process defines how to install component implementations, create components with their homes, and connect the components together as described in the packaging descriptors.

The CCM standard leaves the deployment interfaces under-specified, however. It specifies ComponentInstallation, AssemblyFactory, and Assembly interfaces which enable installation of component packages and assemblies but shows only an example scenario of the instantiation of containers and component homes. The example scenario uses non-standard interfaces to provide a complete deployment scenario. Only partial implementations of the deployment architecture exist so far.

2.1.5 The execution model

The execution model describes the runtime environment for the components. The runtime environment is called the component container. The container provides an additional abstraction layer above the CORBA platform hiding non-functional aspects like handling of CORBA services and persistence management from the component implementor. It is intended that the generated component implementation skeleton interacts with the container through the container interface.

The number and properties of containers is fixed in the standard. The specified containers are service, session, process, and entity container. Each container has its roots in the EJB specification. According to available implementation experiences (e.g. [6]), the container interfaces are poorly specified which is one source for the portability problems mentioned in the programming model section.

The GOAL group is designing adaptive containers which would enable adding new services to already instantiated containers making it possible to dynamically deploy and configure them [6]. These kind of adaptive containers may very well be part of the future versions of CCM standard.

2.1.6 The main changes in the finalised version

The finalised version of the CCM specification contains 13 significant changes and 23 minor changes compared to the initial proposal. The following list summarises the main changes:

- Addition of CIF meta-model and related language mappings,
- removal of Component Container Architecture,

- removal of Packaging and Deployment meta-model,
- fixes in IDL3, Interface Repository interfaces, CIDL language, component model and its IDL mappings, container interfaces, EJB and CCM integration, packaging and deployment XML DTDs, component deployment interfaces, and in baseIDL and componentIDL meta-models.

The CIDL language which has been under wide criticism has kept its existing grammar but a meta-model has been specified for it. It is assumed that CIDL will continue development and new versions of it may contain significant changes.

There are no product level implementations of the EJB/CCM mapping but enough proof of concept exists so that the mapping is kept in the specification with some changes. The EJB/CCM mapping may also be a target for changes in the future.

2.1.7 Available implementations

By now, there exist several platforms which implement at least parts of the CCM specification. None of them support the whole specification yet, but together they implement all the features which are included in the finalised specification. The following list summarises the properties of the five most sophisticated implementations available. The information is based on what is available at the implementations' web-sites [1, 3, 7, 9, 10].

OpenCCM (LIFL)

- Author: LIFL GOAL group
- ORBs supported: ORBacus, OpenORB, Visibroker, JavaORB
- Implementation language: Java
- Special features: Open Source; First Java implementation published; Name will be changed in the future because of the Exolab's OpenCCM
- Where to find: `http://corbaweb.lifl.fr/OpenCCM/`

MicoCCM

- Author: FPX
- ORBs supported: Mico
- Implementation language: C++
- Special features: Open Source; First C++ implementation published
- Where to find: `http://www.fpx.de/MicoCCM/`

OpenCCM (Exolab)

- Author: Exolab
- ORBs supported: OpenORB
- Implementation language: Java

- Special features: Open Source, Exolab also the author of OpenEJB, OpenJMS, OpenORB; promises CCM/EJB integration based on OpenEJB and CIDL support
- Where to find: `http://openccm.exolab.org/`

K2-CCM

- Author: ICMG
- ORBs supported: information not available
- Implementation language: information not available
- Special features: Provides a CCM/EJB bridge and a CIDL compiler
- Where to find: `http://www.componentworld.nu/corp/ccm/ccm.overview.asp`

CIF-CCM

- Author: CIF project at Sourceforge
- ORBs supported: information not available
- Implementation language: C++
- Special features: Open Source (LGPL)
- Where to find: `http://sourceforge.net/projects/cif/`

The platforms are in their pre-alpha/alpha phases, and do not qualify as industrial level platforms. It is to be expected that new commercial implementations will start shipping during 2002-2003 now that the CCM specification is finalised.

2.2 The first implementation of CCM standard - OpenCCM

OpenCCM was the first published implementation of the CCM specification. OpenCCM is a Java-based Open Source platform implemented by the LIFL GOAL group. This section describes the history and goals of the OpenCCM platform as well as its main features as they were in the latest published version (v0.2). In addition some measurements and benchmarks are reported.

2.2.1 History and goals of OpenCCM

Originally the OpenCCM platform was meant to be a research platform. The idea was to implement parts of the CCM specification for research purposes. When the GOAL group later joined the CCM finalisation task force, these plans were re-evaluated and it was decided to go for an "industrial quality Open Source CCM platform". The latest published version of OpenCCM is version 0.2. The OpenCCM platform is implemented in Java and, at the moment, supports only Java based components. The GOAL group however promises to support also C++ based components in the future [6].

OpenCCM can be used on top of several ORBs. Version 0.2 supports ORBacus 4 for Java, Visibroker 4 for Java, OpenORB, and JavaORB. More ORBs are said to be supported in the future.

Next we will take a look the parts of the CCM standard the version 0.2 implements and how the platform will develop in the near future.

2.2.2 Features implemented in OpenCCM

The OpenCCM 0.2 implements only parts of the original CCM specification. Parts included in the platform so far are:

- Interface repository for IDL3,
- IDL3 compiler for loading IDL3 files into the repository, and generation of IDL2 files and Java-based component implementation skeletons from the repository,
- a monolithic component server / container implementation capable of hosting components. The monolithic server provides the standard ComponentInstallation interface for installing the `jar`-packages containing component implementations and a proprietary interface for instantiating component homes.

Several upgrades to the platform are under development [6]. The time schedule for the updates remains to be seen but it is promised that following functionality will be included quite soon:

- Packaging tool supporting OSD-based packaging,
- a model and a framework for defining component based applications called "CODeX - Composite Oriented Deployment and eXecution" model,
- a deployment framework that supports the standard CCM deployment model and extends the model to fit a larger set of requirements,
- adaptive containers which allow for the composition and deployment of containers into component servers in a similar way components themselves are composed and deployed.

The idea that monolithic containers are replaced with modular services that can be used to compose generic component containers sounds very interesting from the Pilarcos point of view, and might provide new possibilities to integrate part or all of the Application Binder [4, 5, 11] functionality to CORBA in a more sophisticated way. One might envision, for example, that Application Binder could be implemented a service which could be plugged into any container on demand.

2.2.3 Performance measurements

In order to get basic understanding of the OpenCCM runtime behaviour, some code generation and memory consumption metrics were gathered and some basic benchmarks were conducted.

Size of the platform

The OpenCCM 0.2 platform is packaged into a `jar`-package sized 1.1MB. Compared to ORBacus core classes (3.4MB) and ORBacus naming service (0.5MB) we can conclude that the size of the platform in its latest release is about one third of the size of the core platform it is run above and twice the size of a naming service. It should be kept in mind that the size of the platform will continue to grow in the future.

Code generation

The code generation metrics were gathered by first specifying a simple interface with one operation, a component implementing this interface and a home managing the component and secondly by generating the code with ORBacus and OpenCCM tools. The results are summarised below:

- CORBA based generation (with ORBacus tools):
    - IDL to Java generation produces 7 classes
    - Size of compiled classes: 11kB
    - Size of `jar`-packaged classes: 7kB
- CCM based generation (with OpenCCM 0.2 tools):
    - IDL to Java generation produces 38 classes
        - 7 classes for provided interface, total 11kB (same as in basic CORBA case)
        - 8 classes for component, total 36kB
        - 23 classes for component home, total 43kB
    - Generation of component implementation skeleton produces 1 class, total 1kB
    - Generation of home implementation skeleton produces 1 class, total 1kB
    - Total size of all compiled classes: 92kB
    - Total size of `jar`-packaged classes: 46kB

The results show that the amount of generated classes in OpenCCM case is massive compared to basic CORBA case. This is mostly due to the complex IDL3 to IDL2 mapping in CCM specification, especially for component home. The difference in size of the generated code is not of major concern since most of the OpenCCM generated classes are quite small.

Memory consumption

The memory consumption metrics were gathered by first instantiating the component server environment and then instantiating simple homes and components in it. In order to measure how much memory consuming functionality is added to the component executor with each additional provided interface, the memory consumption was measured with a component that provides one interface as well as with a component that provides five similar interfaces. The measurements were made using the `vmstat` and `free` utilities in the Linux system.

To provide a baseline for the measurements, the memory consumption of the Java Virtual Machine (JVM) was also measured. This is important as the memory footprint of the JVM is included in all the other measured footprints. It should be kept in mind that the JDK-version and used heap-size may greatly affect the memory consumption. As another baseline, the memory footprint of ORBacus naming service was also measured. The results are summarised below:

- Configuration:
    - Java version 1.2.2 Classic VM (build 1.2.2-L, green threads, nojit)

  - Default JVM, ORB, and OA start-up parameters and JVM heap size

- Measurements:
  - JVM and a class with only main() method 2.7MB RAM
  - ORBacus naming service: 4.1MB RAM (includes a JVM instance)
  - OpenCCM 0.2 component server: 4.9MB RAM (includes a JVM instance)
  - Instantiation of a component home: 900kB
  - Instantiation of component providing 1 interface: 400kB
  - Instantiation of component providing 5 interfaces: 600kB

It can be seen that the instantiation of the component platform takes around 5MB of memory. Roughly half of this footprint belongs to the JVM. The memory footprint of the OpenCCM platform seems to be somewhat larger than that of the ORBacus naming service but not considerably.

A simple component consumes a bit less than 0.5MB of memory and the footprint grows with about 50kB for each additional interface provided. Instantiation of the component home takes almost 1MB of memory. It should be noted that the component implementation includes only the generated parts of the component and embedding the actual business logic will make the footprint larger. The growth depends on the complexity of the application. In future releases the logic related to each facet may be instantiated without instantiating the whole component. This reduces the memory consumption of large components providing many interfaces.

```
interface Benchmarking {
    typedef sequence<octet> OctetSeq;

    /* for scalability benchmark */
    void ping();

    /* for marshalling benchmark */
    void marshal(in OctetSeq parameter); // 0kB, 0.5kB, 1kB, 2kB, 4kB, ...

    component BenchClient supports Management{
        uses Benchmarking bmarking;
    };
    home BenchClientHome manages BenchClient{
    };

    component BenchServer supports Management{
        provides Benchmarking bmarking;
    };
    home BenchServerHome manages BenchServer{
    };
};
```

Figure 2.1: The benchmarking interface and components.

Benchmark configuration

A series of simple benchmarks were conducted in order to explore the runtime behaviour of OpenCCM platform. These benchmarks do not provide much information themselves, and they should be used together with the wider benchmarks concerning the behaviour of the Pilarcos prototype as a whole [11]. The benchmarks reported here are simple client-server benchmarks which give some basic understanding of the benchmarking environment used during the prototype benchmarks.

The benchmarks are divided into two separate groups: the marshalling benchmarks, and the scalability benchmarks. The marshalling benchmarks perform operation calls between the client and the server components with growing amount of octet-data as operation parameter. The scalability benchmarks perform empty operation calls with full throughput and growing number of background clients. The idea of the scalability tests is to explore the invocation routing behaviour of the platform under heavy load.

The hosts used in the benchmarks had the following properties:

- Processor: 1GHz Pentium III
- Memory: 512MB RAM
- LAN: 100Mbit Ethernet
- Linux 2.2.19 OS and Java version 1.2.2 Classic VM (build 1.2.2-L, green threads, nojit)

The host and platform configuration was the following:

- One component server and CORBA component in each host (client or server component)
- Default ORBacus ORB threading model (blocking) and two different POA threading models (threaded, thread_pool=10). The fundamental difference between the threading models is that the threaded model synchronises the execution of the user code whereas the thread pool model allows several concurrent executing threads. The size of the pool defines the amount of pre-instantiated threads available for incoming requests. Pre-instantiation increases the performance of a server by preventing frequent thread creation and destruction.

All measures are gathered by making 1000 operation calls and calculating their mean value. This is done to get the average behaviour. Then this procedure is repeated 20 times and the mean value of the 20 previously calculated average values is calculated. This method makes sure that the measurement procedure is distributed in time. This has to be done in order to remove the effect of randomness in the operating system behaviour. Things measured in the benchmarks are:

- Marshalling benchmarks: Operation calls with growing amount of bulk data as a parameter (0 bytes, 500 bytes, 1kB, 2kB, ... , 128kB)
- Scalability benchmarks: Operation calls with empty operations at full throughput with growing number of background clients. The measured client and the background clients use the same interface. The things measured are the throughput and invocation time seen by each client, the throughput seen by the server, and the CPU load at the server node.

The interface and component definitions used in the benchmarks are shown in Figure 2.1.

Benchmark results

The marshalling benchmark shows a linear trend in the invocation times. The invocation with no parameters takes around 0.65-0.70ms and the invocation with 128kB octet-data as a parameter takes around 17ms. The linear growth is expected because the parameter size should only affect the marshalling/unmarshalling time.

The marshalling/unmarshalling is performed by static stub/skeleton code which consumes CPU time an amount linearly dependent on the size of the marshalled parameter. There is no real difference in invocation times between different POA threading policies. The roundtrip time compared to parameter size can be seen in Figure 2.2.

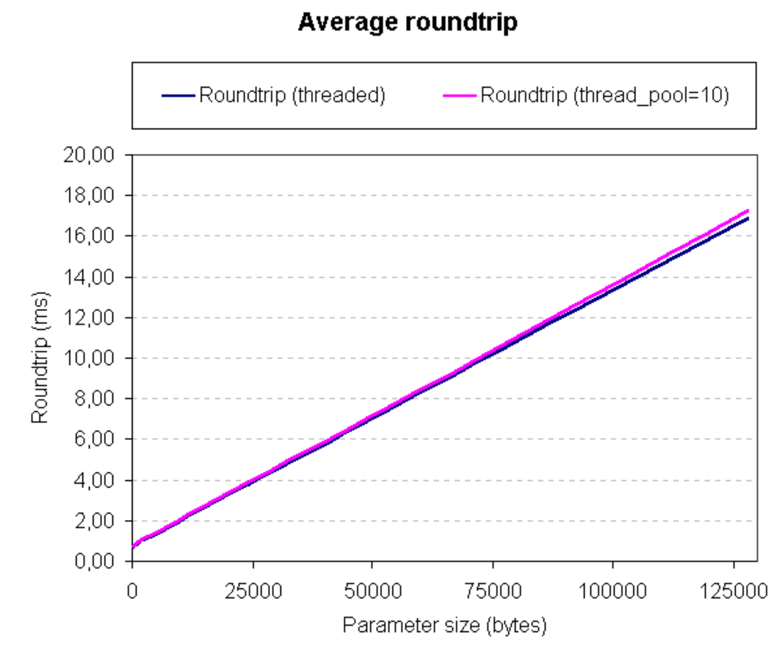

Figure 2.2: Invocation time versus size of marshalled parameters.

The scalability benchmarks show some differences between the different threading policies of the POA. Using a pool of pre-instantiated threads results in a greater overhead but a faster response to suddenly growing loads. A fixed-size pool also results in a more deterministic behaviour both at the client and at the server side. Figure 2.3 shows the server throughput with both policies. It can be seen that the server throughput is higher without the pool (around 1600 invocations per second) than with the pool (around 1400 invocations per second) in case of single concurrent client.

Figure 2.3 also shows how the server throughput grows faster when a thread pool is used and stabilises in a level around 2400-2500 invocations per second after the CPU is fully utilised (see Figure 2.4). The throughput of the `threaded` version reaches a maximum of around 2700 invocations per second but becomes very un-deterministic and starts to decrease after the CPU is congested. With 10 simultaneous clients the throughput of both threading models is equal.

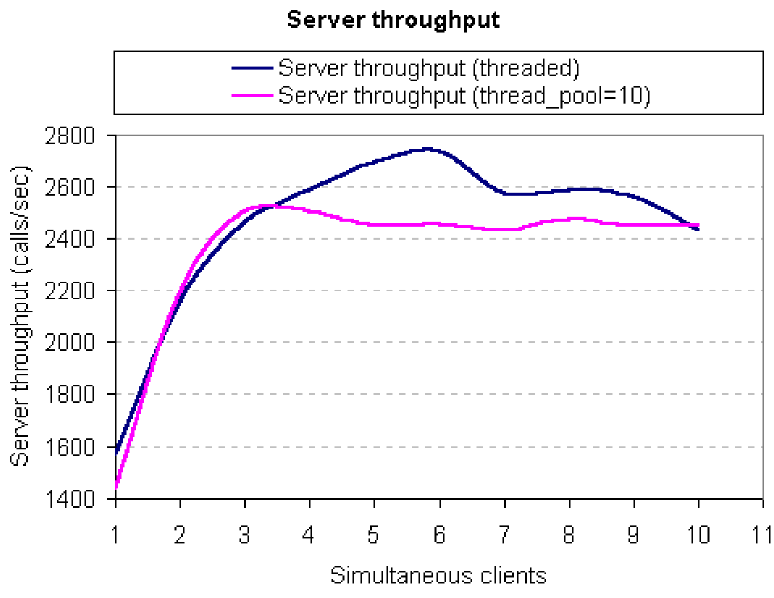

Figure 2.3: Total server throughput versus simultaneous clients.

As a conclusion it can be said that the marshalling/unmarshalling of operation parameters usually consumes most of the time during an operation call. It can also be said that in the environment we used, the network latency had nearly zero-effect on the response time. In a slower or congested network the situation would be different

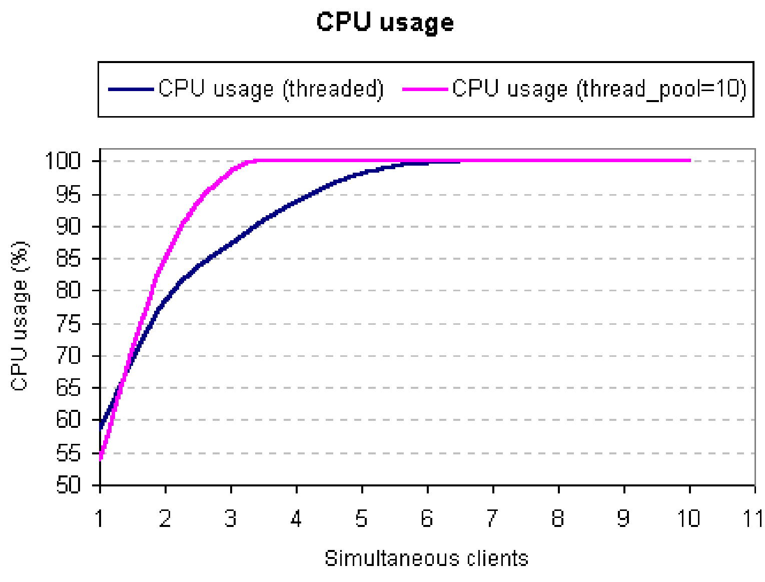

Figure 2.4: CPU-load versus simultaneous clients.

The scalability tests show that the default threading model of the ORBacus POA (`threaded`) produces less overhead than the pooled model (`thread_pool=10`). If variation in observed behavior should be minimised and/or the invocations arrive in short bursts, the thread pool provides a clear advantage.

The size of the pool should be fitted according to the amount of expected simultaneous clients. Facts to be taken into account when choosing thread pool size are the average number of concurrent clients during a typical burst, and the maximum expected number of concurrent clients during a typical burst.

Chapter 3

Software development with OpenCCM - development process and experiences

Previous chapter provided an overview of the CCM standard and its first implementation, the OpenCCM platform. This chapter describes the development process used when implementing CCM based applications with OpenCCM and summarises the experiences gained.

3.1 Pilarcos prototype - the context for OpenCCM evaluation

The Pilarcos prototype is the first realisation of the Pilarcos infrastructure services. The prototype is implemented in the context of a Tourist Information Service business case. The business case defines a context where clients can request tourist information services through a portal service and pay those services with a payment method of their choice. The prototype implementation contains first versions of Pilarcos infrastructure services implemented as CORBA components as well as simple implementations of the service applications involved. The complete definition of the business case and the prototype design can be found in an accompanying document [11].

The prototype development environment was the following:

- Operating system: Linux 2.2.19
- Programming language: Java
- Language environment: JDK 1.2.2 (also JDK 1.4 beta 2 was tested)
- CORBA ORB: ORBacus for Java 4.0.5
- CCM platform: OpenCCM 0.2

The prototype development group consisted of three people. Each of the three participated in design and implementation phase. The group had considerable amount of previous experience in programming with Java in centralised environments but less experience in distributed Java programming.

The group had no experience in CORBA Component Model based programming and only some experience in conventional CORBA programming. The group, however, had a fair understanding of the CORBA and CORBA Component Model specifications. In addition, the group had professional level understanding of networked environments in general.

The development effort took four months in real world time. Each of the three developers used around one third of their working hours in the development effort. The exact amount of work consumed to produce the implementation is difficult to calculate, because the research work cannot be separated from the development work.

The following features of ORBacus and OpenCCM were used in the prototype:

- ORBacus 4.0.5
  - ORB
  - Naming service
  - Trading service
  - IDL to Java generator (`jidl`)
- OpenCCM 0.2
  - Interface repository for IDL3
  - IDL3 to IDL2 generator (`ir3_idl2`)
  - Component executor generator (`ir3_java`)
  - Component implementation skeleton generator (`ir3_jimpl`)
  - OpenCCM runtime component server
  - OpenCCM components and homes
  - Synchronous connections between components (facets/receptacles)

The size of the software produced during the prototyping effort can be summarised as follows:

- Around 300 lines of IDL definitions (not counting comments or empty lines),
- around 3500 lines of Java-code (not counting comments or empty lines),
- 32 user written Java classes including the implementations of 10 CORBA components.

3.2 OpenCCM based development process

This section explains the process of OpenCCM based development. It shows, in chronological order, the necessary steps needed to develop software with OpenCCM. From each step the necessary actions to take, the problems which may arise, and the solutions to the problems are explained. A concrete example of each step is also provided in order to better illustrate the meaning of each action. Figure 3.1 shows the different steps in the development process. The steps are explained in the following subsections.

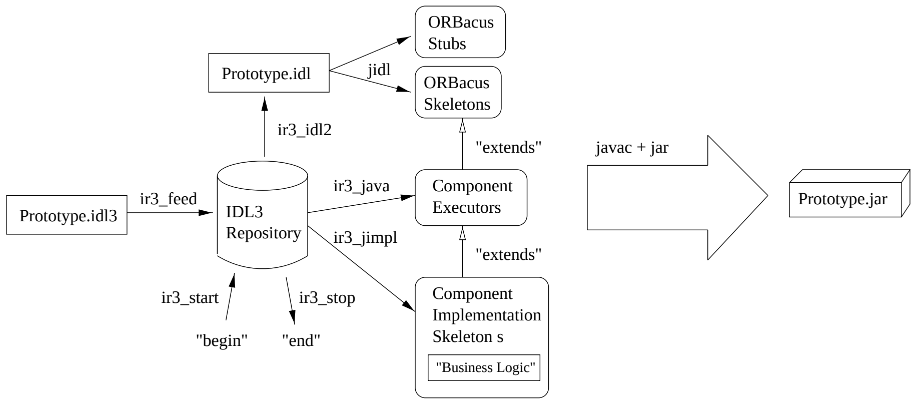

Figure 3.1: Overview of OpenCCM based development process.

3.2.1 Getting started

The first step in the development effort is to find the platform implementations and related documentation. In our development environment the Linux operating system and Java platform (JDK 1.2.2) were pre-installed in the machines so what was left was to get ORBacus 4.0.5 for Java ORB, ORBacus naming service, ORBacus trading service, and OpenCCM 0.2 platform.

The ORBacus platform is available both in source code and in binary format. Since we had no plans to touch the ORB code or the naming/trading service code, we just downloaded the ORBacus components in binary format. The ORBacus platform and related documentation are available at the IONA web-site [2].

The OpenCCM platform was downloaded in source code format. The OpenCCM source code can be found from the OpenCCM web-site [9] along with separate installation and development guides. The guides are actually only web pages providing short explanation on how to install and use the platform. The source code package also provides some exemplary source code. In addition to the OpenCCM guides, the CCM specification [8] is useful in understanding some of the more complex issues. There also exists a very nice document describing the GOAL group experiences with CCM and detailed examples on how to use OpenCCM in development [6].

3.2.2 Installation

The installation phase includes installation of the ORBacus platform and installation of the OpenCCM platform. In our case this means installing the ORBacus 4.0.5 binaries and configuring, compiling and installing the OpenCCM 0.2 source code.

The ORBacus installation only requires copying the used platform libraries to some directory. In our case the needed libraries were `OB.jar`, `OBNaming.jar`, `OBTrading.jar`, and `OBUtil.jar` containing the core classes, naming service, trading service, and necessary utilities. In addition to these the ORBacus command-line utilities, like the `jidl` compiler, are needed.

OpenCCM installation is somewhat complicated. Un-zipping the platform release results in a complex directory structure. The configuration of the platform is done by editing three files related to the used operating system, the used Java environment, and the used ORB product. Since we were using Linux and ORBacus the files needed to be configured were `make` rules for Unix systems, `make` rules for JDK in Unix systems, and `make` rules for ORBacus in Unix systems.

When the configurations are properly set, the platform can be compiled with the `make` utility. The compilation procedure results in several new directories plus the `OpenCCM.jar` package containing the compiled platform code.

After compilation, the platform can be installed from command-line by typing `make install`. This installs the platform to a directory specified during the configuration phase. The installation results in three new directories (`bin`, `lib`, `idl`) which contain all the tools, libraries, and IDL interfaces of the platform. The `bin` directory also contains `envi.OpenCCM.sh` script which can be used to set values for the environment variables.

3.2.3 Definition of interfaces and components with IDL3

The actual development effort starts with the definition of interfaces and components. First phase is the definition of the IDL interfaces. This phase is identical to that of conventional CORBA development. Here is where the first bug in the OpenCCM platform was discovered: it is not possible to use several IDL modules. Nested IDL modules result in parallel Java packages in the generated code. The usage of multiplex receptacles also leads to several errors in the generated code.

Because the time frame and work hours available in our development effort were limited, it was decided that instead of fixing the platform code, we put everything in one IDL module called `Prototype`. The module and its descriptions were placed in a file called `Prototype.idl3`.

An example of the definition process is provided in the context of Tourist Information Service (see [11] for the service description). First the service interface provided by the Tourist Information Service needs to be defined. The interface includes three operations called `requestService`, `useService`, and `abort` which are used for requesting the service related billing information, for using the service, and for aborting the service usage. The interface is shown in Figure 3.2.

```
interface TouristInfo {
  void requestService(out string billId, in string servContId);
  OctetSeq useService(in string what, in string servContId);
  void abort(in string servContId);
};
```

Figure 3.2: The IDL interface of the Tourist Information Service.

The next phase is to define the interfaces needed by the Tourist Information Service. The service needs one external interface called Billing in order to support on-line payment via an external payment service provider. The Billing interface provides operations for initialising the billing procedure, for aborting the billing procedure, and for checking if the customer has paid his bill. Figure 3.3 shows the interface used for billing.

Now the needed interfaces are defined and it is time to define the CORBA component implementing the Tourist Information Service. The component is defined to provide the TouristInfo interface and use the Billing interface. The resulting component definition is shown in Figure 3.4.

```
interface Billing {
  string getBillId(in Bill bill, in string account,
                   in string servContId);
  void invalidateBill(in string billId, in string servContId);
  boolean isBillPaid(in string billId, in string servContId);
};
```

Figure 3.3: The Billing interface used by the Tourist Information Service.

At this point we have the definition of the service interfaces used and provided by the TouristInfoServer component as well as the definition of the component itself. Because our component is supposed to work in the Pilarcos environment interacting with the Pilarcos services, we need to define some additional interfaces for policy and federation management. The interfaces needed are the PolicyConfiguration interface and the FederationManagement interface.

```
component TouristInfoServer {
   uses Billing billing;
   provides TouristInfo touristInfo;
};
```

Figure 3.4: The TouristInfoServer component with basic definitions.

It is intended that in the future the policy and federation management interfaces become an integrated part of every component so that it would not be necessary to define them separately for every component.

In order to support component configuration and benchmarking we also need to add few attributes to the component. One attribute is needed to support configuration of component debugging level and two others for workload simulation and benchmarking purposes.

In addition, we need a simple home which can be completely generated and needs only to have the ability to create and destroy components. Unfortunately the generated homes in OpenCCM 0.2 do not (yet) implement the operations for destroying components so we can only have a home which automates the component creation.

Putting all the pieces together we have the complete IDL3 definition for the Tourist Information Service including the interfaces, component, and home definitions. Figure 3.5 shows the final design of the TouristInfoServer component (some irrelevant details omitted). This example shows nicely the power and elegance of the new IDL3 concepts of component and component home.

At this point it should be reminded that the definition process was presented as a rather straightforward procedure. This actually gives a false impression. In reality the process of (1) IDL3 definition, (2) IDL2 generation, (3) code generation, (4) component implementation, (5) component and application testing goes on in an iterative manner until the IDL definitions stabilise. The fact that the IDL definitions change during the development cycle introduces difficulties related to code generation and implementation phases. These issues are discussed more in corresponding subsections.

```
component TouristInfoServer {
  uses Billing billing;
  provides TouristInfo touristInfo;

  uses PolicyConfiguration policyConf;
  provides FederationManagement federationManagement;

  attribute long workMillis;
  attribute long sleepMillis;
  attribute long debugLevel;
};

home TouristInfoServerHome manages TouristInfoServer {
};
```

Figure 3.5: The IDL3 definition for the TouristInfoServer-component.

3.2.4 IDL3 to IDL2 generation

In order to proceed to the code generation phase, the IDL3 definitions have to be converted to IDL2 definitions. The CCM specification provides a standard mapping to do this. The mapping is realised in OpenCCM by an IDL3 to IDL2 generator which utilises the interface repository in the process.

The tools needed to generate the IDL2 definitions are

1. `ir3_start` to startup the interface repository,
2. `ir3_feed` to feed the IDL3 files into the interface repository,
3. `ir3_idl2` to generate the IDL2 files from the contents of the interface repository, and
4. `ir3_stop` to terminate the interface repository.

The usage of these tools to generate the IDL2 files for the Pilarcos prototype is illustrated in Figure 3.6.

```
ir3_start
# Feeding the OMG IDL3 Repository with Prototype.idl3
ir3_feed Prototype.idl3
# Generating idls for Prototype module
ir3_idl2 -o Prototype.idl -i Components.idl -i CosNaming.idl
         -i CosTrading.idl -i CosTradingRepos.idl ::Prototype
ir3_stop
```

Figure 3.6: Usage of OpenCCM interface repository to generate IDL2 descriptions.

The generation procedure produces the `Prototype.idl` file which contains the IDL2 versions of the prototype definitions.

3.2.5 Code generation

The code generation phase has three distinct steps: (1) generation of component and home executors, (2) generation of component and home implementation skeletons, and (3) generation of marshalling stubs and skeletons.

The generation of component and home executor skeletons is achieved by utilising again the interface repository. In addition to the tools utilised in the IDL3 to IDL2 section a new tool called `ir3_java` needs to be used. Figure 3.7 shows the flow of activities.

```
ir3_start
# Feeding the OMG IDL3 Repository with Prototype.idl3
ir3_feed Prototype.idl3
# Generating OpenCCM skeletons Prototype module
ir3_java ::Prototype
ir3_stop
```

Figure 3.7: The generation of component and home executors from IDL3.

The results from the executor skeleton generation phase are the non-functional parts for components and component homes. These parts act as glue between the user implemented part of the component and the component container. In the future versions of the platform the generated executor can be customised by including proper CIDL statements into the generation process.

In addition to the component and home executor generation the OpenCCM also provides tools for generating the skeleton classes for the user implemented parts. These so called implementation skeleton classes extend the generated component and home executors and include the signatures for all the business operations. What developer has to do is to add the business logic into the operations. The implementation skeleton generation is achieved by using the `ir3_jimpl` tool. Figure 3.8 summarises the tool usage process.

```
# Generating Java implementation skeletons
ir3_start
ir3_feed Prototype.idl3
ir3_jimpl ::Prototype
ir3_stop
```

Figure 3.8: The generation of the component and home implementation skeletons.

A clip from the implementation skeleton of the TouristInfoServer component (Figure 3.9) shows the generated signature for the `requestService` operation.

The stubs are generated from the IDL2 file `Prototype.idl` with ORBacus IDL to Java generator (`jidl`). The stub generation does not involve the interface repository (see Figure 3.10).

During the development effort, one bug was discovered related to the generation phase. The generated implementation skeletons (both component and home implementations skeletons) try to import code from a non-existing package. The extra line (see Figure 3.11) can be found just below the package definition at the beginning of each class. This bug

```
public class TouristInfoServerImpl  // component implementation skeleton
       extends TouristInfoServerCCM // component executor
       implements fi.helsinki.cs.pilarcos.Prototype.TouristInfoOperations,
       fi.helsinki.cs.pilarcos.Prototype.FederationManagementOperations
{

    /* Example operation signature from the implementation skeleton */

    public void
    requestService(org.omg.CORBA.StringHolder billId, String servContId)
    {
        //
        // TODO : put your own code here !!!
        //
    }

}
```

Figure 3.9: Part of the implementation skeleton generated for TouristInfoServer component.

```
# Generating Prototype CORBA 2 stubs.
jidl --auto-package --tie --output-dir $(gendir) -I $(idldir) Prototype.idl
```

Figure 3.10: An example of the ORBacus IDL to Java generator usage.

is probably a legacy of the OpenCCM 0.1 and a straightforward solution to this problem is to remove the extra line from each implementation skeleton. In order to automate the removal process a simple shell script was written.

The generation paradigm was found to be easy to use and effective. Some problems arise, however, if the IDL descriptions change frequently. Every time the IDL changes, not only the stubs and executors need to be re-generated but also the implementation skeletons. This means that the implemented business logic must be moved from the old implementation skeletons to the new ones. This problem is a fundamental problem in the generation paradigm.

3.2.6 Component and deployment application implementation

Implementing components using the generated implementation skeleton is, in theory, a straightforward procedure. The components are implemented by just filling in the business logic to the generated implementation skeleton. In reality the process involves complicating

```
import fr.lifl.goal.OpenCCM.runtime.*;
```

Figure 3.11: The import bug in the beginning of each generated implementation skeleton.

aspects like concurrency of execution.

The concurrency problem arises when there are multiple threads simultaneously executing the business logic of the component. In this case the component programmer must make the component implementation thread-safe by synchronizing the access to the concurrently used data structures, and in some cases synchronize the usage of the business operations themselves. This complicates the programming effort and requires some understanding of the problems related to concurrency and synchronization. Understanding the business logic is not enough.

In addition to making the component code thread-safe, component thread-safety should be made explicitly visible in the component documentation and packaging descriptor. Because OpenCCM 0.2 does not support XML-based packaging this could not be done in the Pilarcos prototype.

The concurrency issue becomes a real problem when people start using off-the-shelf CORBA components made by third parties in critical software. If components which are not programmed thread-safe are deployed in multithreaded environments the result may be a catastrophy.

After the component is implemented it is usually a good idea to add the `configuration_complete` operation in order to support the division of the components life-cycle into configurational and operational phases as shown in Figure 3.12.

```
public class TouristInfoServerImpl  // component implementation skeleton
       extends TouristInfoServerCCM // component executor
       implements fi.helsinki.cs.pilarcos.Prototype.TouristInfoOperations,
       fi.helsinki.cs.pilarcos.Prototype.FederationManagementOperations
{

     /* The added configuration_complete() operation */

     public void configuration_complete ()
         throws org.omg.Components.InvalidConfiguration
        {
           // Check if the configuration is valid,
           // and ``lock'' the attributes which should not
           // change state after this operation is called
        }
}
```

Figure 3.12: Adding the `configuration_complete` operation to the implementation skeleton.

If a component uses services from other components, the client behaviour can be added by bootstrapping the interface references from the receptacles using the `get_connection` operation in case of simplex receptacles and `get_connections` operation in case of multiplex receptacles. Figure 3.13 shows how the Billing interface is bootstrapped and used in the `requestService` operation of the Tourist Info Server component.

In case the platform would provide a deployment tool, implementing the components would be enough. Unfortunately OpenCCM 0.2 does not offer a deployment tool, and a proprietary development application must be implemented in order to use the components.

Since the platform is in its early stages and is evolving constantly there was no point in implementing our own generic deployment tool. It was decided to implement a simple

```
public class TouristInfoServerImpl  // component implementation skeleton
       extends TouristInfoServerCCM // component executor
       implements fi.helsinki.cs.pilarcos.Prototype.TouristInfoOperations,
       fi.helsinki.cs.pilarcos.Prototype.FederationManagementOperations
{

    /* Implementation of the requestService-method (some details
        e.g. exception handling omitted) */

    public void
    requestService(org.omg.CORBA.StringHolder billId, String servContId)
    {
           Bill bill = new Bill("TouristInfoService", 6.0, "Test Bill");
           String account = "10101010";

           // Bootstrapping the Billing interface from the receptacle
           Billing billing = get_connection_billing();

           synchronized (billIdTable) {
                // Using the Billing interface
                billId.value = billing.getBillId(bill, account, servContId);
                billIdTable.put(servContId, billId.value);
           }
    }

}
```

Figure 3.13: Example of bootstrapping and usage of an interface connected to a receptacle.

deployment application which can deploy, instantiate, and connect the prototype related components. The demonstrative examples which come along the OpenCCM distribution gave a good starting point for implementing it. Figure 3.14 shows a small deployment application which is able to deploy and instantiate the TouristInfoServer component, configure it, as well as connect the Billing interface to one of its receptacles.

3.2.7 Compilation and packaging

The compilation of both, the generated files and the user implemented files is done in a three-phase compilation chain. First the generated ORBacus stubs ans skeletons are compiled, then the generated executors are compiled and third the generated implementation skeletons filled with user written business logic are compiled. In our case, the compilation was done using the `javac` compiler included in the JDK 1.2.2 release.

As the OpenCCM platform does not support OSD-based packaging yet, the packaging was performed by simply putting all the compiled Java-classes in one `Prototype.jar` file. This package was used as the basic unit when deploying components in the component servers. The packaging was performed using the `jar`-tool included in the JDK environment. In a more realistic scenario the Java-classes related to each component should be separately packaged and zipped together with the corresponding XML-descriptors.

```
// Obtain the component server reference from the naming service
fr.lifl.goal.OpenCCM.ComponentServer.Server server = obtainComponentServer(componentServerName);

// Get the ComponentInstallation interface from the component server
org.omg.Components.Deployment.ComponentInstallation install = server.provide_install();

// Install the Prototype.jar containing the prototype classes
install.install(implId, archiveURLString);

// Get a home factory from the component server
fr.lifl.goal.OpenCCM.ComponentServer.CCMHomeFactory factory = server.provide_ccm_home_factory();

// Create a home for the TouristInfoServer
org.omg.Components.CCMHome home = factory.create(
                "fi.helsinki.cs.pilarcos.Prototype.TouristInfoServerHomeImpl",
                "touristInfoServerHome");

// Narrow the home to the exact home type
touristInfoServerHome = TouristInfoServerHomeHelper.narrow(home);

// Instantiate the TouristInfoServer component
touristInfoServer = touristInfoServerHome.create();

// Configure the attributes of the component
touristInfoServer.debugLevel(this.debugLevel());
touristInfoServer.workMillis(this.workMillis());
touristInfoServer.sleepMillis(this.sleepMillis());

// Get the reference of the Billing facet from the Application Binder
Billing billing_facet = touristInfoAB.provide_billing();

// Connect the Billing facet to the component receptacle
touristInfoServer.connectBilling(billing_facet);

// End the configuration phase
touristInfoServer.configuration_complete();
```

Figure 3.14: Code example from a deployment application.

3.2.8 Problems related to federation management

Because the Pilarcos infrastructure services were implemented as CORBA components, their implementation procedure was similar to the application service implementations. There are, however, some problems related to management of Pilarcos federations which should be pointed out (see [11] for a definition of Pilarcos related concepts).

The Pilarcos infrastructure services lift the abstraction level seen by component programmer making the business logic unaware of lower level connections, even of the CORBA object references used and provided. It is in the responsibility of the infrastructure services to know to which application federation each incoming and outgoing operation call is related to and handle them accordingly.

Figure 3.15 shows an example case where a service application is connected to three clients (on the left side) and three sub-services (on the right side). There exists an association between each client and a corresponding sub-service, for example between the

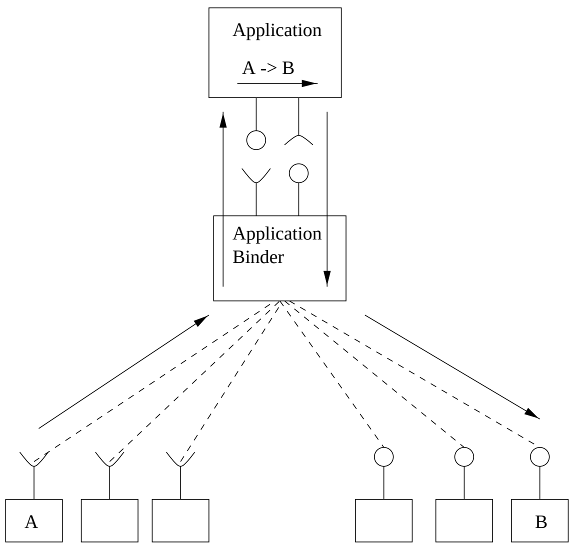

Figure 3.15: A service federation between client A, application service, and sub-service B.

client marked with A and the sub-service marked with B. The association is established before the service usage time during the federation negotiations. The figure also shows an infrastucture service called Application Binder (see [11]) which is responsible for managing the federations and their connections.

If we look at the situation from the Application Binder point of view, it first receives an invocation from client A, identifies the sender as A and applies corresponding federation policies to the request. It then forwards the invocation to the application service. The problem here is that the application service must know from which client the forwarded invocation came from and to which federation it is associated to. This information cannot be implicitly provided to the application service without complicating the business logic implementation, for example in a CORBA service context list. For this reason it was decided to add an explicit attribute (`ServiceContractId`) to each application service interface which identifies the related federation.

The usage of an explicit attribute also makes it easier to implement the application service and Application Binder in a multi-threaded environment where there may be several simultaneous client threads executing the business logic at the same time and usage of the implicit execution context becomes complicated.

After receiving the invocation, the application service executes the business logic which, in this case, requires usage of another application service (B). Since the business logic is completely unaware of the underlying connections, the Application Binder must intercept the invocation and identify that it is going to service B. In order for it to do this, the application service must provide it the `ServiceContractId` attribute it received in the up-call. In the selected CORBA component based service implementation, this requires that the `ServiceContractId` is added also to the interface the Application Binder provides to the application service.

It is worth noticing that the multiplex receptacles are not suitable for implementing the connection between the Application Binder and the sub-services (e.g. B). This is because the receptacles do not provide fine-grained query operations, but only `get_connections` operation which returns all the connected facets with every query. Thus the Application

Binder keeps the object references it uses hidden in internal data structures.

Because all the actors are CORBA components the described problems become visible in all component configurations where multi-party associations exist.

3.3 Developer experiences

The IDL3 concepts of component and component home were found to be useful when designing CORBA based applications. Components provide a nice way to group interfaces into implementation units and homes standardise component life-cycle management. The ability to describe components' dependencies on external interfaces (by defining receptacles and event sinks) is the enabling technique for designing large and automatically deployable component based applications.

The fact that there exists no methodology or design process for using IDL3 hampers its usability to some degree. There also exists no long-term experience on the effects of poorly designed components in rapidly changing systems. It can still be said that at least in the prototyping environment the IDL3 was seen as a clear improvement compared to conventional IDL.

The possibility to generate all necessary glue and skeleton code was also seen as an improvement. It certainly enables new developers to start producing software much faster than with conventional CORBA environment. On the other hand as the amount of generated code grows larger and the generation-chains get longer it becomes harder for the developer to understand what is actually happening. The temptation of not bothering to understand what happens "behind the scenes" grows which leaves the system more vulnerable to possible side-effects.

Frequently changing IDL also causes problems because the changes in IDL result in re-generation and re-implementation of the components. It is quite frustrating to have to copy the implemented parts of the components to newly generated component implementation skeletons repeatedly. With proper tool support the problems related to generation might be avoided. In general the CCM programming model places more importance on software development tools than does the conventional CORBA programming model.

The implementation of components in single threaded environments is rather straigforward. Adding business logic to the generated implementation skeletons is enough. In multi-threaded environments the business logic must be made thread-safe which complicates the programming task. Components can be made thread-safe by synchronising access to important data-structures and, in some cases, synchronising operations themselves. The thread-safety properties of the component should be explicitly stated in the component documentation and in the component packaging descriptor.

Since the OpenCCM 0.2 includes no XML-based packaging support the packaging model could not be evaluated. It is clear however that without sophisticated tool support the packaging process is not very user friendly.

OpenCCM does not implement all the standard deployment interfaces nor the standard interfaces are even enough for a full-scale deployment. For this reason the evaluation of the deployment model usability is OpenCCM specific. What can be said about it is that the concept of automated deployment with standard interfaces seems very promising even with the OpenCCM 0.2 monotlithic component servers. The idea of the standard deployment interfaces and component as a unit of deployment is perhaps the largest individual advance compared to conventional CORBA development.

The fact that component ports are fundamentally designed for client-server configurations was problematic when implementing federation management services as components. Since there is no built-in support for multi-party associations the information had to be explicitly included to the IDL interfaces.

As it is now the OpenCCM 0.2 is not suitable for commercial level usage. However, with the major IDL and code generation bugs fixed it suits well to research and educational purposes as well as experimental prototyping.

Chapter 4

Concluding Summary

CORBA Component Model specification is one of the key advances to be included in the CORBA 3.0 standard. The specification brings to CORBA perhaps the most advanced component model in the market.

Summary of CCM related issues

The CCM specification includes core component model (i.e. the abstract model), component programming model, component packaging model, component deployment model, and component execution model. The key features of each model can be summarised as follows:

- Core component model defines the semantics of the components. IDL3 extensions are specified for the definition of components, their communication ports, and component managers (i.e. homes),
- component programming model defines a framework (CIF) for component implementation. The focal point of CIF is the CIDL language. CIDL is used for defining how components interact with their run-time environment,
- component packaging model defines how components are packaged to deployment units. Packaging model is based on XML-based Open Software Description (OSD) model and enables packaging of single components as well as component assemblies,
- component deployment model defines how to automatically deploy and instantiate component based applications in distributed component servers,
- component execution model describes the component run-time environment. The run-time environment is called a container. The container provides an additional abstraction layer above the CORBA platform hiding the non-functional aspects like handling of CORBA services.

In addition to the CCM architecture the specification defines an inter-working architecture with EJB 1.1.

The first published standard proposal was huge and contained many major and minor issues to be corrected. Each model went through some changes but the most radical changes in the finalised version are the addition of a CIF meta-model, the removal of the component container architecture (container interfaces were not removed), and the removal of packaging and deployment meta-model (XML-descriptors were not removed). Other changes include fixes in IDL3, Interface Repository interfaces, CIDL language, component model and its IDL mappings, container interfaces, EJB and CCM integration, packaging and deployment XML DTDs, and component deployment interfaces.

The syntax and semantics of the CIDL language and the container interfaces were not majorly changed even though implementation experiences point out that implemented components are not portable across platforms from different vendors. It is assumed that the programming model and container model will go through some iterative changes in the future versions of the standard.

The CCM/EJB inter-working architecture has not been fully implemented anywhere, but enough proof-of-concept exists so that it will be kept in the finalised version with some improvements.

It took over a year before the first implementations of CCM began to show up. The first implementation published was OpenCCM [9] from LIFL GOAL group. The first C++ based implementation published was MicoCCM [7] from FPX. Recently implementations from Exolab, ICMG, and CIF-project at Sourceforge have been announced and more may be under way. All the implementations so far are in pre-alpha/alpha phases and suitable mostly to educative/evaluative/research purposes. It is expected that the first industrial quality implementations will be shipping in 2002/2003.

It can be said that the next couple of years in the implementation frontier are very crucial to wide adoption of the standard.

Summary of OpenCCM features and benchmarks

As is case with all the other implementations, OpenCCM 0.2 implements only parts of the CCM specification. Parts implemented so far are Interface Repository for IDL3, tools for loading IDL3 files into the repository as well as tools for generating IDL and Java files from the repository. A monolithic component server/container implementation is also provided. In future, the platform is supposed to include support for OSD-based component packaging, a sophisticated deployment machine, a framework for defining component based applications, and composite, adaptive container framework. From these, the adaptive container concept is interesting from Pilarcos point of view because it extends the CCM container model with new concepts that might be useful in implementing some of the Pilarcos infrastructure services.

The measurements made with OpenCCM 0.2 did not bring up anything extraordinary, but some interesting issues were revealed. Comparison of conventional IDL to Java generation with ORBacus tools and IDL3 to Java generation with OpenCCM tools showed that the amount of extra classes generated from IDL3 definitions is large. In case there are a large number of interfaces and components, the management of the generated classes becomes a problem at least without sophisticated software engineering tools. It is good to notice that the large amount of classes is mostly due to the complex IDL3 to IDL2 mapping for component homes.

In addition to code generation measurements, the memory consumption of starting a component platform and instantiating simple components and their homes was measured. As the platform is implemented in Java and a JVM instance must be running in each machine the platform is running, the memory consumption of the JVM itself gives a good baseline for memory consumption measurements. When interpreting the results it should be kept in mind that the JVM version and the used heap-size may greatly affect the memory consumption.

Assuming the default JVM heap size it can be said that the OpenCCM 0.2 component platform roughly doubles the memory consumption compared to the JVM instance alone. In case of component instantiation, it can be concluded that the memory consumption caused by adding facets to the component is minor compared to the memory consumption caused by adding business logic to the component except for very simple applications. Instantiating component homes takes a considerable amount of memory and in environments where memory is a scarce resource it might be a good idea to distribute the homes and the components they manage to different component servers. The ability to do this depends on the platform capabilities (i.e. support for light-weight proxy homes). In the future releases of the platform it should be possible to instantiate needed facet implementations on-demand without instantiating the other facets of the component. This makes it possible to reduce the memory consumption further.

In addition to code generation and memory consumption measurements, some benchmarks were conducted in the same environment with a client-server configuration. The benchmarks were divided to simple marshalling and scalability tests. The average invocation time with an empty operation call took less than 1ms. This illustrates the time it takes to route the invocation from the client application to the server application. Adding growing amounts of octet-data as a parameter makes the invocation time grow linearly. If we would have used complex data structures like complex IDL-structs or any-values there would probably have been additional marshalling delays involved. In any case, it can be concluded that the marshalling delays dominate the invocation routing times.

The scalability benchmark tested what happens when several concurrent clients are generating load to the server component with empty operation calls. The scalability tests brought up clear differences between the different POA threading policies. The thread pool model was clearly more deterministic and produced less variation in server throughput than threaded model. From a performance point of view it can be concluded that if variation in invocation times must be short and the invocations arrive in short bursts the pooled model should be used. Otherwise a model with no thread pool is recommended because of smaller overhead.

Summary of OpenCCM experiences

The OpenCCM platform was used to implement Pilarcos architecture in the context of a Tourist Information Service business case.

The component concept was found to be useful both as a design concept as well as a deployment concept. The ability to group interfaces to implementation and deployment units and the ability to define their dependencies on external interfaces were recognised as key benefits. In fact the idea of the component as a unit suitable for automatic deployment was found to be the most important change compared to conventional CORBA programming. The lack of concrete design methodology for CORBA components was, however, a complicating factor.

The code generation paradigm was found to be effective especially for programmers who have little experience in CORBA programming. When experience grows the advantage gained gets smaller.

Because the generated parts of the implementations must be regenerated each time the IDL definitions change, it is advisable to design the IDL to be as stable as possible. A design/implementation cycle where IDL definitions are changed at a fast pace and in an iterative manner causes extra work because the existing business logic must be copied from the old implementation skeletons and integrated into the new ones in each iteration. This problem exists to some degree in conventional CORBA development too, because changing the IDL usually causes some changes to the implementations. With generated code the problem is, however, much wider since the old implementation skeleton becomes totally useless when a new skeleton is generated. Software development tools may decrease the problems involved.

Adding the business logic was a rather straightforward procedure except for the fact that components which are intended to work in a multi-threaded environment must be programmed to be thread-safe. Multi-threading is a problem which concerns the whole paradigm of component based development. Even if most of the technical details can be abstracted away from component programmers, like use of connections and use of CORBA services, the problems with concurrent execution of multiple threads can not be. The thread-safety of a component must be clearly stated in its documentation in order for it to be safely used.

Finally, since component ports are fundamentally designed for client-server associations and component receptacles lack introspection capabilities it is difficult to implement multi-associative services, like federation management services, as CORBA components.

Concluding words

CORBA Component Model is the most sophisticated component model in the market and despite the delays in its finalisation work still holds great potential. In order for the CCM model to gain success in the future following key requirements must be fulfilled:

- Mature, industrial strength implementations must be available relatively soon,
- interworking capabilities with other technologies must be improved,
- the poorly standardised parts must be rapidly improved,
- no more fundamental problems should be discovered.

Even if all these requirements are met, the popularity of the CCM model will ultimately depend on the limits of the existing, first generation, component models. It cannot be assumed that CCM will replace existing technologies, like EJB, in the current application environments, but its main strengths are in (future) application scenarios which are beyond the reach of existing technologies.

Bibliography

[1] CIF-CCM web-site. http://sourceforge.net/projects/cif/.

[2] IONA / ORBacus web-site. http://www.ooc.com/ob.

[3] K2-CCM web-site. http://www.componentworld.nu/corp/ccm/ccm.overview.asp.

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

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

[6] Marvie, R., and Merle, P. CORBA Component Model: Discussion and Use with OpenCCM. Tech. rep., June 2001. http://corbaweb.lifl.fr/OpenCCM/docs/2001_06_Informatica.ps.

[7] MicoCCM web-site. http://www.fpx.de/MicoCCM.

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

[9] OpenCCM web-site (LIFL). http://www.lifl.fr/OpenCCM.

[10] Open web-site (Exolab). http://openccm.exolab.org.

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