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

Pilarcos prototype II

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

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

Pilarcos prototype II

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

University of Helsinki
Finland

Pilarcos prototype II

Markku Vähäaho, Juha-Pekka Haataja, Janne Metso, Timo Suoranta, 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, iv + 83 pages

Abstract

The rise of globalisation makes inter-organisational cooperation an essential requirements for current IT systems. On the software engineering arena, use of component technologies is encouraged by production and maintenance cost reductions. Distributed system research seeks improved functionality and cost reductions by more enhanced middleware solutions. Under these pressures, new middleware is expected to provide dynamic and automatically controlled facilities for inter-organisational cooperation.

Pilarcos project develops mechanisms for automatic management of large-scale, inter-organisational applications based on federation contracts. Federation contracts defines the business process in use between the partners, the selection rules for members in the federation, and the current members of the federation together with meta-information about their technical properties.

This document presents the new prototype implementation of the main Pilarcos services and sample applications. The new prototype and this document explore and demonstrate the following features of the Pilarcos architecture:

- using explicit architecture descriptions for federation management;
- automated multiple-service lookup;
- automated federation contract negotiation;
- federations across different technology domains (CCM and EJB); and
- using policy management facilities for putting federation contracts into effect at each involved domain.

Besides the implementations of the new middleware services (federation manager, Pilarcos trader, type repository, and factories), the results of performance measurements on the prototype software are reported.

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

General Terms:
Design, Documentation, Experimentation

Additional Key Words and Phrases:
Software system architectures, federated systems

Contents

Chapter 1

Introduction

Current information processing needs of companies require inter-organisational cooperation. New middleware is expected to provide dynamic and automatically controlled facilities for inter-organisational cooperation. Current middleware solutions already provide reasonable support for managing the technological heterogeneity of operating systems and network solutions and for adapting to dynamic changes in available resources. However, in a multi-organisational environment, decisions on provision of application-level service, on operational policies, on platform architectures, and on communication protocols can be done independently from other systems. Further difficulties encountered by inter-organisational applications include the need to adapt to the constant change in potential partners and the independently driven development of services in each of these systems.

These challenges should be addressed by improved middleware support. Facilities are needed for expressing service requirements, and for describing alternative conversation patterns between services. Furthermore, facilities are needed for negotiating joint rules on new cooperation relationships.

Therefore, new middleware should provide facilities for capturing the workflow, searching for available members for the workflow, and ensuring that a suggested set of members is able to interoperate both semantically and technically.

The Pilarcos project develops mechanisms for automatic management of large-scale, inter-organisational applications. Running an inter-organisational application is considered as a federation that is managed using an explicit federation contract. The contract essentially includes

- a description of the business process as an architecture description,
- selection rules for members in the federation, and
- references to the current members of the federation and metainformation about their technical properties.

At each domain, there are local servers that are able to

- search and locate potential members for a federation,
- negotiate and establish a federation contract,
- start and configure local services to act as members of the federation, and
- jointly establish communication between peer members in the federation across domain boundaries.

Pilarcos middleware provides application programmers with pervasive, platform-independent tools to manage federations. However, the use of the functions remains explicit. Providing federation management facilities in a middleware has the benefit of releasing programmers from re-implementing these facilities repeatedly in applications. Furthermore, inter-organisational interoperation can only be achieved through standard solutions and in this case, via standardising new middleware services and metainformation elements.

Separating design of business architecture descriptions and implementation of components improves the software engineering process in general. The natural lifetimes of these elements differ, as well as the skills required for the provision of them. The Pilarcos middleware service allows workers to concentrate on their component logic and provides business architectures and technology mappings as ready-made solutions, as instructed by the currently popular production line concept [5].

Providing tools for automating the administration of service composition reduces system maintenance cost. Because the federations directly support changes in membership, there is good support for adaptation to changes in the business situation and also to changes in the technical availability of services. The design incorporates into the same management model both private business within the organisation and external cooperations with varied and potentially contradictory requirements.

The Pilarcos project has had two phases. In the first phase, a middleware model based on the use of explicit architecture descriptions and dynamic federations was created. Also, in the first phase, a proof-of-concept prototype was developed and its performance evaluated.

In the second phase, a new prototype implementation of the main Pilarcos services and sample applications have been written. The new prototype and this document explore and demonstrate the following features of the Pilarcos architecture:

- using explicit architecture descriptions for federation management
- automated multiple-service lookup
- automated federation contract negotiation
- federations across different technology domains
- using policy management facilities for putting federation contracts into effect at each domain.

The focus of the second prototype has been on the interoperability between domains using different middleware technologies (CCM [16, 17] and EJB [20, 21]). In addition, essential changes have been made on federation coordination services.

The rest of this document is structured as follows. Chapter 2 gives an overview of the prototype. After the main ideas have been introduced, the overall structure and main components are described briefly. Management of federations is also discussed. After this, the document goes into more technical detail.

The most important concepts and data structures are discussed in Chapter 3. Chapter 4 presents the example application scenario used, the Tourist Information Service, and how the application components are constructed. Descriptions of the application programming interfaces (APIs) to Pilarcos infrastructure services are also included. Chapter 5 describes the Pilarcos infrastructure services: the Pilarcos trader, the type repository, the policy repository, different factories, and the federation manager. The benchmarking results of the prototype are presented and analysed in Chapter 6. Chapter 7 concludes the document.

Chapter 2

Overview of the prototype

Pilarcos prototype software includes middleware services for managing federations, services and bindings, and in addition, an experimental application scenario. The implementation environment of these software elements is heterogeneous, consisting of CORBA Component Model (CCM) and Enterprise Java Beans (EJB) platforms.

This section first introduces the new middleware functionalities suggested by the Pilarcos architecture also addressing the meta-information manipulated by the middleware services involved. The software elements of Pilarcos prototype are briefly introduced and the configuration and allocation of services is shown.

2.1 New middleware functionality

Inter-organisational cooperation creates a situation where a business process involves components that have independent administrations. This independence means for example, that the time of availability of a service is not commonly known and that decisions that effect the interactions related to the service can take place without the agreement of all cooperating partners. Also, independent technology decisions in each organisation may cause mediation needs without much warning.

So far, the method of adapting to the above mentioned problems have been manual management of software. When a cooperation partner changes versions or products for communication platform or provided services, corresponding changes or invention of adapting wrappers urgently takes place.

One of the major goals in Pilarcos project series is to provide facilities for automating much of this adaptation work. From this goal, two issues arise. First, how to express who the cooperating partners are and what their responsibilities are. Second, how to automate the checking of conformance and how to combine elements into a running system at each organisation. In addition, there must be a way to express both organisational and technical domains.

For the definition of partners and responsibilities, we have chosen to use business architecture descriptions. A *business architecture* is defined by a set of roles, interactions between roles and a set of policies. The business architecture description does not fix the identities of the participating systems. Instead, roles are associated with *service type* that defines the the class of service required. Members for the federation are selected based on service offers that describe the component's service type, technology requirements, conversation protocols expected, operational policies, cost, location, etc. *Policies* in a business architecture have two targets. First, a policy rule can be set to govern the behaviour of a component in a role. For example, different information retrieval strategies can be preferred depending whether there is need to save space or time in a search. This kind of expectation can be passed on to a component via a role related policy. Second, a policy rule can be set to govern interactions in the federation. The business architecture can model alternative interaction sequences and the policy value can be used to define the desired selection of these.

For creating a federation services are needed for discovering the services currently made available from each organisation. In addition, services are needed for checking the conformance of a component to a federation. For this, details of the conversation needed for using a specific service, and details of the information exchange protocols used for these conversations are required. For the necessary meta-information to be available, information providing facilities are needed. For example, tools are needed for designers to enter descriptions of services and the runtime environment to register services when they are installed or started. Furthermore, a variety of adaptors for different kind of mediation needs is needed, to be automatically used where necessary.

For structuring and organising management tasks, the Pilarcos approach uses two types of domains: administrative domains and technology domains. An administrative domain can be for example an organisation, a company or a department with authority to do independent operational decisions about the way it runs its business. Organisation-wide policy management allows IT-system administrators to reflect operational policies – such as restrictions to cooperation partners, payment related conversation styles and time of availability of offered services – consistently onto all applications of the organisation in an automated manner. Service descriptors, service management rules, and policies are defined at the administrative domain level in technology-independent terms.

Component management rise the need of separation between technology domains. For each technology, there are various facilities for deployment, instantiation and termination of components. A technology domain is here limited within an administrative domain for simplicity. At each technology domain, service descriptors and service management rules are mapped onto technical engineering solutions. Naturally, these mappings follow a pattern common to all administrative domains.

Another goal for the Pilarcos architecture is that components can be involved in multiple federations simultaneously. An organisation cooperating with many partners has the problem of having contradictory communication requirements from its partners. However, it is often the case that no task in itself involves all partners at the time. The components are expected to follow policy rules stated in the local policy repository. As a private set of policy rules can be applied for instances of services, instances with mutually contradictory policies can be running simultaneously without problem. Federation contracts are an integral part of component management mechanisms, because the relevant information they contain is stored into the policy repository. Consequently, simultaneous federations can be contradictory with each other, making it possible to simultaneously adapt to the needs of different cooperation partners.

The essential federation management functions required from the new middleware include

- definition of policies for components within an organisation,
- establishing federations, and

- terminating federations.

From these, federation establishment is of special interest. It comprises of

- selection of architecture for the federation,
- selection of members for the federation, involving check of conformance of behaviour and interfaces for the potential members against the requirements of the business architecture and each others, and
- selecting joint policy for the federation.

Once the federation contract has been established, the federation itself can be made functional. This means locally mapping the contract to local technical solutions in order to be able to start services and bind them together.

2.2 Pilarcos middleware services

The prototype middleware includes implementations of an architecture description-based trading service for looking up services in the network, federation management services for automated federation negotiation and establishment, and simplified application and channel instantiation services. The goal was to build a rather complete proof-of-concept implementation of the main Pilarcos components, and to explore interoperability between middleware platforms in practice.

Federation manager handles federations with other organisations. Federation manager implements the federation establishment protocol, in which one federation manager is chosen as the coordinator for the entire federation. The federation manager also stores federation contracts, and presents APIs for using the Pilarcos services to the applications.

The federation managers are responsible of running the protocol for negotiating, maintaining and re-negotiating federation contracts. For federation managers, the essential information element is of type *federation offer*. It is a combination of compatible service offers, one for each role in a specific business architecture. The federation establishment protocol is initiated by a client request. As a first step, a service offer that describes the client itself is positioned into a federation offer element. The Pilarcos trader then populates the rest of the roles. As a result, several suggested federation offers are returned for the client to choose from.

Pilarcos trading service creates federation contract offers for an entire community. It also acts as a directory of service offers as service providers export their offers to the trader.

The enhanced Pilarcos trader provides two main operations: exporting a *service offer* for a specific service type (`export`), and populating a business architecture with mutually compatible service offers (`populate`). The former operation could be used with an administrative tool by a service provider. The latter operation is used by the Pilarcos federation manager on behalf of the application wishing to establish a federation. The `populate` operation takes an incomplete federation offer as a parameter, and returns one or more completed federation offers. No separate constraint parameter is used; instead, the incomplete federation offer typically contains a *pre-filled* service offer for the populating role itself, defining its policies for the federation. Thus, the population process is completely symmetrical: any role that has been left empty in the incomplete federation offer is populated by the Pilarcos trader. This makes it easy to do partial re-populations for failure recovery or adaptation purposes.

Type repository supplies the trading service with type information about service types, architecture descriptions and service interfaces. This information is needed for testing whether a component is a suitable member in a federation.

The Pilarcos middleware design includes an enhanced version of ODP type repository [7] for holding relationship information between generic types (service types, binding types, interface types) that are technology-independent and used for matching purposes.

Policy repository holds a hierarchy of policy contexts within the organisation. These contexts contain administrative, application-specific and federation-specific policies that constrain the behaviour of applications and infrastructure services, all in a unified structure.

Pilarcos middleware expects that components can be managed by policies, much like in policy-based management systems (e.g. [14]). Administrators can create and set policies for component groups. Furthermore, federation contracts are stored into the policy repository and thus federation contracts become an integral part of component management mechanisms.

Service management is a generic interface implemented by the *service factory* facility. It is used for instantiating application components using services from technology-specific factories, the application homes.

Channel management is a generic interface implemented by the *service factory* facility. It is used for creating and configuring adapters for communication between technologically differing domains.

The Pilarcos middleware services reflect the administrative and technology domains and the need to cooperate across the domain boundaries. There are collaborative middleware services with a running agent at each administrative domain for negotiating federations and advertising available services. These agents take care of making requests to their peers at other domains, as there is no authority to otherwise invoke management actions at a foreign domain [10]. The requests carry contracts to pass relevant meta-information that identifies what should be done and how. On the other hand, there are local management facilities running at each technology domain for managing components. In addition, at each administrative domain there are agents responsible of mapping abstract federation contract information into a form usable at technology domains.

2.3 Demonstration application

The Pilarcos prototype software includes application services for a demonstration federation. In this section, an overview of that application is given.

The case is built around an idea of a portal service, the `Tourist Info`, which provides travellers access to vertical tourist services like travel information, hotel bookings, and weather services. It is assumed that neither the portal service nor the vertical services are free, and the traveller has some electronic payment instrument (e.g. credit card) available.

The prototype implements two business communities: tourist info community and hotel info community. Tourist info community contains three domains each with their own role (tourist info client, tourist info service, payment service) and describes the business community related to providing the portal service. Hotel info community also contains three domains each with their own role (hotel info client, hotel info service, payment service) and describes the business community related to providing the only implemented vertical service (hotel bookings). Tourist info client and hotel info client represent the users of corresponding services whereas tourist info service and hotel info service represent the providers of the services. Payment service represents a trusted third party used to mediate

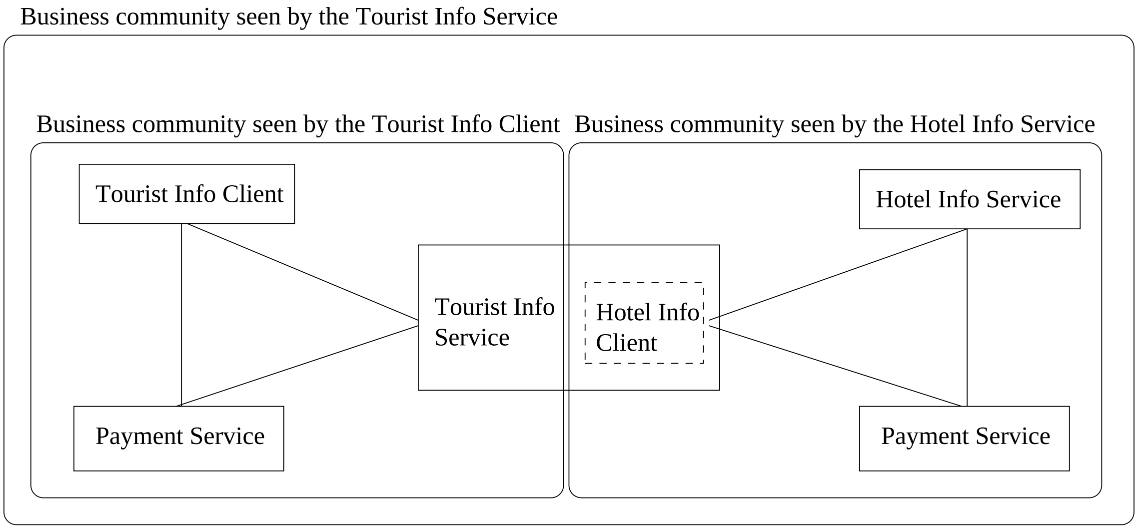

Figure 2.1: Business entities and communities in the `Tourist Info Service` case.

the transfer of funds between the other entities in the community. Figure 2.1 shows the business entities and the communities formed by them.

The federation structure is defined by an architecture description, which captures

- a set of roles,
- bindings between the roles representing their interactions, and
- assignment rules (property requirements) for the roles.

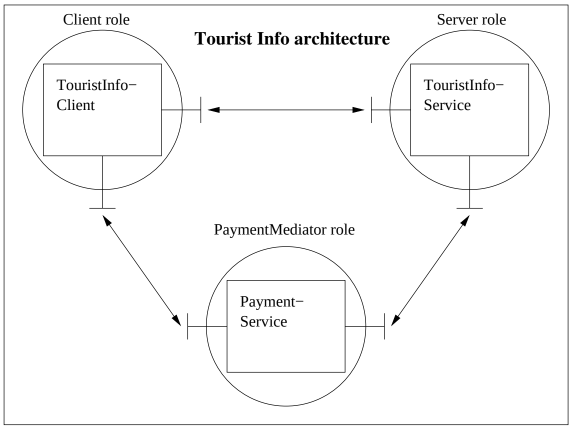

Figure 2.2: Tourist information service architecture illustrated.

Figure 2.2 illustrates the Tourist Information Service architecture description. Roles are drawn as circles; each role has an associated service type, drawn as a square, with interfaces to other parties. In the prototype, the architecture description is used for looking up compatible service providers as well as for establishing a federation – including communication channels – between the participating systems. Section 3.4 contains a more detailed description of architecture descriptions in the current prototype.

The process of using the Pilarcos trader to find suitable service offers for empty roles in the architecture is called *populating* the architecture. The Pilarcos trader verifies the compatibility of the individual service offers, and returns complete federation contract offers. Having populated an architecture, the client can then proceed to establish a federation on the basis of one of the offers with the help of the federation manager. In this process, a final federation contract is formed; this contract includes application-specific policies as well as technical interoperability data. Note that an application may participate in several federated communities at the same time, possibly in different roles, and that these communities are logically completely separate from each other.

2.4 Other required software elements

Bindings between components require that the peer components have compatible interfaces. In Pilarcos, we are concerned of semantical compatibility as described in this section, and allow adaptation to take place between interfaces. The adaptation elements can be automatically inserted in to the communication route described in Section 5.3.3.

Selecting components for the application involves tests for component conformity. Conformity of components covers questions such as interface type matching, component behaviour compatibility and contract breaches, and expected binding semantics, to mention a few. (A literature overview on the topic can be found by Spanish colleagues [25].) In a multi-organisational environment, components involved in a federation cannot inherit properties that would ensure interoperability between components. Therefore, several levels of interoperability problems must be solved based on explicit meta-information exchange.

First, the participants should have matching views of the logical business process they are involved in and matching policy decisions where alternative cobehaviour models are possible. For example, in a tourist service application, where clients can make hotel reservations through a tourist office, there might be alternative payment methods, such as pre-paid vouchers for hotels or credit card payment after the visit, embedded as alternative behaviours into a shared application logic.

Second, there should be matching views of the computational communication solution involving the participants. For example, a payment interaction can be computationally implemented by a sequence of protocol messages between buyer, seller and bank, and each party should have similar assumptions about the message formats and ordering; if not, some bridge solutions should be used between the parties for creating a coherent view.

Finally, there should be matching views of the engineering of the communication mediating solutions. For example, each party expects the transport protocol to preserve message sequences or support transaction transparency; if that is not the case at some domain, additional services can be used to intercept and upgrade the transport service.

Adapters are small elements that are able to do transformations. A transformation can be either syntactic or semantic in nature. a proxy reference or a remarshalling of a message. Adapters are implemented as components in the current prototype. In the prototype, adapters are essentially technology-bridging entities, and limited to technical transformations.

The federated binding mechanism configures an adapter into a binding when the interfaces are semantically similar, and a relationship between them is known and supported by an adapter.

Adapters need to be implemented as independent elements and stored into repositories for general use. Methods or tools for adapter implementation has not been studied in this project. Related research on the topic is described for example by SOFA project [1].

We assume that practical needs and market forces push for an efficient set of adapters to appear. However, the number of adapters will be fairly large and constantly evolving, thus making their management automatic once they have been created.

2.5 Prototype layout

In concrete terms, the Pilarcos prototype consists of Pilarcos infrastructure service components and Tourist Information Service application components that take advantage of the Pilarcos services. Infrastructure services and most of the application components have been implemented on two new CORBA Component Model platforms, the Java-based OpenCCM [18] and the C++-based MicoCCM [15]. In addition, one of the application domains is built on Enterprise JavaBeans technology, running on the JBoss application server [9]. Seamless interoperability between the three platforms has been a major focus in prototype development.

Figure 2.3 illustrates the main Pilarcos infrastructure components and their relations in a generic setting. The organisational domain borders separate the organisations running the client application, the server application, and the global trading and type repository services.

As can be seen from Figure 2.3, the communication between the so-called client and server domains takes place on three levels. The lowest level is the federation management level. On this level, messages (technically IIOP messages carrying operation calls) related to federation negotiation and establishment are exchanged between federation managers. Communication channels (in the prototype, IIOP connections) between applications are established on the channel management level, which is currently implemented as a naming service lookup. Finally, at the topmost level applications communicate with the help of adapters that bridge technological differences between the domains, automatically installed by Pilarcos services.

Pilarcos prototype is developed on heterogeneous environment. A significant part of Pilarcos software is based on CORBA Component Model and Enterprise Java Beans specifications. Within those specifications several implementation options are available with varying licensing, programming language and interoperability properties.

In general the Pilarcos prototype is open source, written in Java and C++, and run on GNU/Linux using Intel x86 compatible PCs. The Pilarcos prototype can be run on other operating systems and hardware environments as long as they have the proper CCM and EJB implementations available.

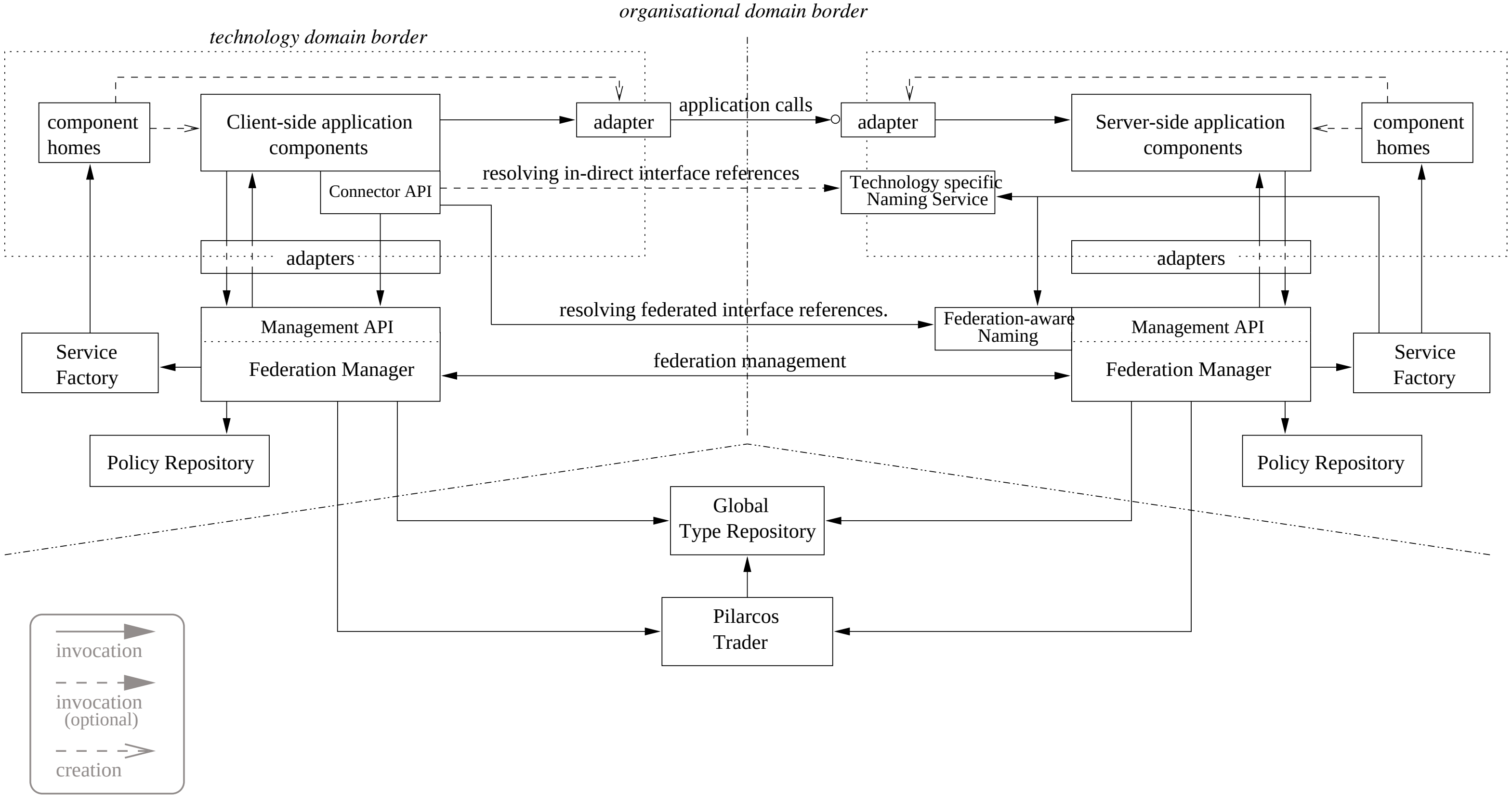

Figure 2.3: Overview of Pilarcos prototype components.

Chapter 3

Concepts

This chapter describes concepts that are frequently used later in this document. First, policies and policy frameworks are introduced in the way they are used in the Pilarcos prototype. Later sections build on top of these fundamental elements describing more complex structures. Service types describe the characteristics of a single service. Architecture descriptions define the structure of a federation composed of individual services, and the agreed state of a federation is captured by service and federation contracts. All this is managed with policy contexts.

For reference, Section A.1 in Appendix A includes the IDL definitions for the data structures described here.

3.1 Policies and policy frameworks

In the Pilarcos model, *federation policies* are used to describe service properties and behaviour. Federation policies, or just *policies* for short, are also coded as name-value pairs. Although policies may carry both business policies and technical property information, the Pilarcos infrastructure makes no distinction between the different kinds of policies.

Related policies are collected together as *policy frameworks*. Because a policy framework, together with a description of its semantics, defines an ontology for policies, policy frameworks are subject to publication and standardisation. The structure of a policy framework is defined by its *policy framework type*, which is a named set of *policy types*. A policy type defines the name and, in our implementation, the OMG IDL type of the policy.

An example pseudocode definition of a policy framework type looks as follows:

```
policy framework type TouristInfoPolicies {
    doubleRange price;              // cost of service
    string      area;               // city
    stringSet   paymentPolicy;      // pre/postpayment
    stringSet   offeredServices;    // e.g. hotel service
    stringSet   supportedTerminals; // e.g. PC, PDA
}
```

Policy framework types are registered to the global type repository (Section 5.4), and are used in policy contexts and service types. To allow multiple logically different policy frameworks that share the same type to be used, a *policy framework* is formally defined as a named instance of a policy framework type.

Comparison of policy frameworks, which must be of the same type, is done by comparing the individual policies within the frameworks. Policies with equal values are defined to be compatible.

The Pilarcos services also recognise special policy value types that make it possible to use simple constraints as policies. Currently, special types are defined for expressing simple closed intervals of integers and real numbers, for example $[1, 5]$, and string set constraints. These types of policies are compatible if their intersections are non-empty.

A string set constraint policy consists of a constraint type and a set of string values. For example, a string set constraint policy of the form `protocol = one_of` {"`IIOP-1.1`", "`IIOP-1.2`", "`RMI-1.0`"} requires that exactly one of the listed protocols must be used as the final protocol. Three constraint types can be used. In increasing order of strength, they are

- `some_of`, which requires that one or more of the strings be present in the final policy;
- `one_of`, which requires the final policy to contain exactly one of the strings; and
- `exactly`, which requires that the final policy must contain exactly the original strings.

All constraint types are exclusive: they prohibit additional values.

The intersection of two string set constraint policies is a string set constraint policy whose constraint type is the stronger of the original types, and whose set of strings is the intersection of the original sets. For example, the intersection of `some_of` $\{A, B, C\}$ and `one_of` $\{B, C, D\}$ becomes `one_of` $\{B, C\}$. In the case of an `exactly` policy, the intersection is empty unless the intersection of the string sets is equal to the string set of the `exactly` policy. Note that it would be possible to have other types than strings as values with the same rules; strings were chosen for simplicity.

String sets include a possibility to specify different weights for different items. The weights are expressed as `doubles` in the range $[0, 1]$. Intervals have preferred values that express the most wanted value within the interval. The trader uses the weights and preferred values when choosing final policy values for federation offers, picking the most commonly preferred alternative.

Intervals and string set constraints are sufficient for expressing most common constraints, and yet are simple enough for efficient calculation. Most importantly, they have the property that calculating their intersections is an associative and commutative operation, which allows the Pilarcos trader to perform the calculations in any order and optimise the search process.

3.2 Interfaces of federated applications

The Pilarcos model has two layers of interface types: abstract and concrete. An abstract interface type represents a logical functionality, defined by an either formal or informal description; a concrete interface type defines the actual operations supported in a concrete interface definition language, such as OMG IDL. A single abstract interface can be implemented as multiple concrete interfaces. The distinction is similar to that between the computational and engineering viewpoints in the ODP Reference Model [8].

Abstract interface types are platform independent; concrete types are platform specific. This two-layered model is necessary to support federation of sovereign systems [11, 6.2], which may use differing implementation technologies. Interfaces are considered compatible if their concrete types are the same, or if an interceptor (adapter or bridge) is available for differing types. Interface types, interceptors and their relations are registered in the type repository, which provides type matching operations. The prototype implementations of the Pilarcos services code concrete interface types as strings, and the type repository simply compares them for equality.

The use of abstract service types is designed to directly support more complex interoperability tests, especially semantic matching or protocol based matching. A variety of existing research elsewhere can be combined in Pilarcos context [25]. Currently only platform related differences or simple application interface differences can be controlled. Also, type repository administrator tools are missing.

3.3 Service type

A service type definition consists of a set of required and provided abstract interfaces, and a set of policy frameworks attached to the interfaces. Defining required interfaces in addition to provided interfaces makes service types abstract, composable components [22].

Figure 3.1 is a condensed pseudocode example of the tourist service type; types are written with capitalised initial letters. Compare this with Figure 2.2.

```
service type: TouristService
{
    provides interface: TouristServiceInterface tourist_service_i;
    requires interface: BillingInterface        billing_i;

    policy framework: TouristServicePolicies tourist_service_pf
        attached to interface: tourist_service_i;
    policy framework: PaymentPolicies        payment_service_pf
        attached to interface: billing_i;
}

service type: PaymentService
{
    provides interface: BillingInterface billing_i;
    provides interface: PaymentInterface payment_i;

    policy framework: PaymentPolicies payment_service_pf
        attached to interface: billing_i, payment_i;
}
```

Figure 3.1: Pseudocode example of service type definitions.

The policy frameworks attached to an interface parameterise the behaviour associated with the interface. It is possible for a single policy framework to be attached to more than one interface, which implies that the policies are shared. In the tourist service example, the payment service has a shared policy framework for both the billing and payment interfaces. This ensures that the payment service, the biller, and the payer have compatible payment policies.

Service types are registered to the global type repository (Section 5.4). The service type attempts to define the external (abstract) interfaces of the service completely, so that a service provider can implement a service independently on the basis of the service type description. Service offers exported to the Pilarcos trader implement some registered service type. Note that in Pilarcos, even a client application that does not provide any services has its own service type. The reason for this symmetry is that each role in an architecture description must have an associated service type (Section 3.4).

3.4 Architecture

In Pilarcos, an *architecture*, or an architecture description, is a template model of a federated community and is used for automated service lookup (using the Pilarcos trader) and federation. The architecture description defines what services the community is composed of, and how they should be connected together. See below for a pseudocode example of an architecture description.

```
architecture TouristInfoArchitecture {
    role client {
        service type TouristInfoClient;
    }

    role server {
        service type TouristInfoService;
    }

    role paymentMediator {
        service type PaymentService;
    }

    binding (client.hotelInfoI, server.hotelInfoI);
    binding (client.paymentI, paymentMediator.paymentI);
    binding (server.billingI, paymentMediator.billingI);
}
```

In the prototype, an architecture consists of a set of roles and a set of bindings between the roles. A role is defined simply as a name with an associated service type, which must be registered in the type repository. Currently, bindings are always one-to-one, and connect the interfaces of two service types in two roles together.

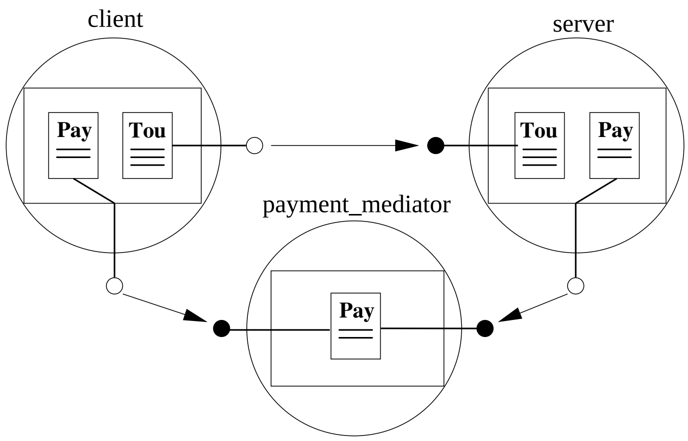

Figure 3.2: Connected policy frameworks in the example architecture.

When interfaces of two roles are connected by a binding, policy frameworks of the same type attached to the interfaces are also implicitly connected. For service offers for the roles to be compatible, their connected policy frameworks must then also be compatible. The compatibility requirement can even extend over several roles via service types that have shared policy frameworks, in the tourist service example payment policies are shared by all roles. For an illustration of this, see Figure 3.2.

Architectures are registered to the global type repository; service types used in roles must have been registered previously. The architecture description to be used is dictated by the client, which sets the appropriate fields in its role context (Section 3.6) before populating. Because the architecture is used only for federation establishment, service providers need not be aware of it at all. The infrastructure handles federation establishment completely transparently.

3.5 Service and federation contracts

The *service contract* structure is used to hold both service offers as well as finished contracts. The service contract represents a concrete service of some service type, and consists of interface references and policy values for the policy frameworks of the service type. In addition to the interface references, which serve to connect federated applications to each other, the service contract also contains an interface reference to the federation manager (Section 5.2). This reference is the initial contact point where a federated service is available.

The *federation contract* structure is used to hold both federation (contract) offers as well as final federation contracts. A federation contract is a collection of service contracts, one per each role in the related architecture.

A federation offer describes an entire community, whose structure is defined by its business architecture definition. It consists of one service offer for each role in the architecture. If one or more service offers are missing, the federation offer is *incomplete*, otherwise it is *complete*.

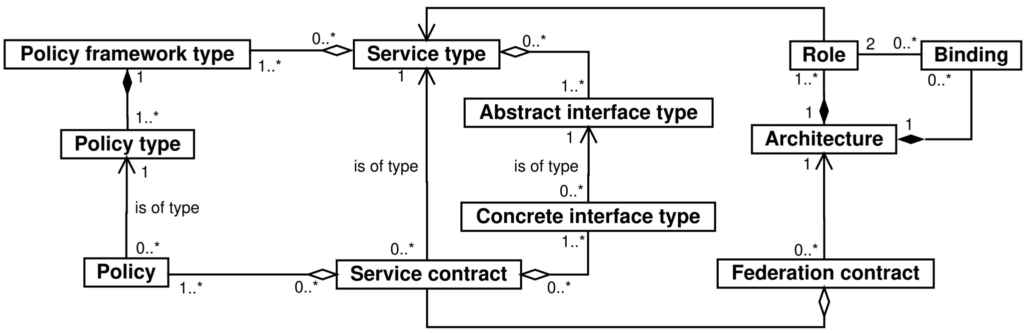

Figure 3.3: UML class diagram of Pilarcos policy concepts.

In other words, a federation contract represents a partially or completely populated architecture. The Pilarcos trader takes in a partially filled federation contract and finds candidates for the empty roles (Section 5.5). After a federation is established, the final federation contract is held by the federation manager (Section 5.2.4) acting as the coordinator, and replicated into other federation managers.

3.6 Policy contexts

Policy contexts are used as a structuring technique in policy management throughout the Pilarcos system. Management of application behaviour, service offers and federations is done by means of an integrated hierarchy of policy contexts, maintained by a central policy repository.

More concretely, a *policy context* is the set of policies (policy values) related to a context within an organisation. For example, each instance of a federated application has its own policy context. Policy contexts are arranged hierarchically, so that the policy contexts in the higher levels restrict the policies in contexts below them. The root of the policy context tree would ordinarily be the organisational policy context; below that, there can be several levels of contexts, with contexts related to individual federations as leafs.

The policy repository of the organisation maintains the hierarchy of policy contexts. Unlike federation contracts, policy contexts are directly visible to application programmers. Applications are obliged to follow the policies defined in the current context, otherwise a breach of contract occurs. However, in most cases, applications need not directly manage policy contexts; this is done by the Pilarcos infrastructure services, which create and modify contexts using the interface offered by the policy repository.

The prototype has four levels of policy contexts: application (instance) contexts, service (type) contexts, (service) offer contexts, and federation contexts, as illustrated in Figure 3.4.

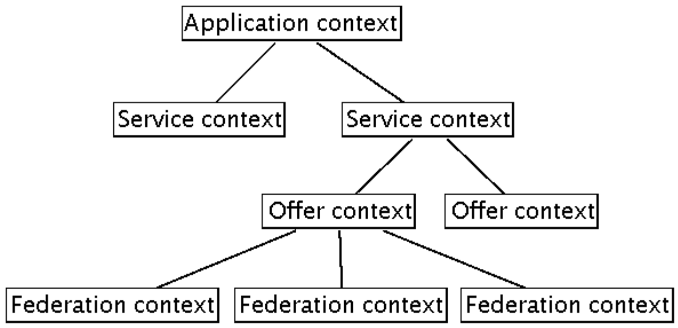

Figure 3.4: Policy context hierarchy in the prototype.

Each application instance has a unique *application context*, created by the service factory at instantiation time. In principle, the application context holds the policies governing the behaviour of the application. As its children, an application context has *service contexts*, one per service type. Service contexts are similarly created at application instantiation, and hold current default policies for service offers of each type.

On the server side, an *offer context* is created as a child of a service context when a service offer is exported to the Pilarcos trader. In the prototype, this happens during the deployment process. The created offer context contains a copy of the policies in the service offer.

On the client side, an application creates an offer context for populating a business architecture. In the population process, the offer context represents the service offer of the populator. Initially, the offer context receives policy values from its parent service context; the application may optionally change them before issuing the population request.

Finally, a new *federation context* is created per federation or federation offer. On the server side, a federation context is created when a new federation is established. On the client side, the federation offers received from the population process are represented to the application as federation contexts. The application may then select one or more of them for establishing a federation and discard the rest.

In the policy repository, new policy contexts are given unique identifiers (policy context ID's) by which they are referenced. Since the identifiers are unique within an organisation, they are used in the prototype also for identifying application instances and federations.

The policy frameworks of a policy context are defined by its *policy context type*. Each level in the policy context hierarchy has a different policy context type. However, policy context types have not been implemented in the prototype: for now, the service type (Section 3.3) acts as the policy context type. Therefore application contexts, which have no associated service type, cannot currently hold any policies; they act merely as identifiers for application instances.

Chapter 4

Application level services

This section discusses application development in Pilarcos environment. The key points of the supported software development and management processes are discussed, and the provided API for component programmers is introduced. The application services implemented for the Pilarcos prototype illustrate the use of these facilities.

4.1 Application development with Pilarcos infrastructure

An inter-organisational application is not developed by a single software engineering team. Instead, it uses services developed and managed separately at independently administered sites.

The establishment of a federation requires the following processes to have taken place

- definition and publication of business architecture specifications
- definition and publication of service type specifications
- implementation and deployment of application components
- implementation and deployment of adapter components

These processes are performed by different kind of personnel, either within a software providing company or consortia (like OMG) that sets de facto directives for cooperation. Business architectures can be specified by people focusing on business processes and commercial rules, company requirements, and the goals of cooperation. Type specifications and verification of their similarities require technical skills of interface and service definition, and knowledge about the requirements of the markets and business architectures. Also a view of development trends is beneficial. Publication and deployment tasks naturally fall on system administrators at each company. Implementations can be produced either by the companies themselves or bought as independent packages.

In the Pilarcos project, only a simple text based interface for publishing business architecture specifications is available. The style of description language can be seen in the application described in Section 2. However, the business process description languages and tools are of increasing interest. For example, Microsoft's XLANG specification [23] considers business processes as contracts between two or more parties, defining the behaviour of each party. Moreover, the business process definition shows how the individual service descriptions are combined, using a map that defines the connections between the ports of the services involved. IBM's WSFL (Web Services Flow Language) approach defines a public interface that allows business processes to advertise themselves as Web services [13]. The recent BPEL (Business Process Execution Language) specification captures features of both WSFL and XLANG [24]. Also UML has been uggested as an appropriate notation for ODP Enterprise language [?]; Enterprise language is the base of Pilarcos business architecture definitions.

In the Pilarcos project, no new software production tools are provided. However, we see the recent development of OMG MDA (model driven architecture) [19] tool chain a complementing approach. Here, a component is to be understood loosely a service implementation encapsulated in such a way that platform services are able to manage its life-cycle (deployment, instantiation, termination, activation and deactivation). Although Pilarcos project has used component based platforms (OpenCCM, MicoCCM, JBoss) for experimenting and prototyping, the Pilarcos architecture does not require component techniques to be used.

In the following subsections, the viewpoint taken is that of component implementor. The application component programmer has middleware level functions available for

- federation management (populating a business architecture, creating or terminating a federation, joining or leaving a federation) and
- policy management on the component itself (setting policy values that are allowed by the existing policy requirements).

The Pilarcos infrastructure is considered to provide a set of business architecture descriptions and service type specifications for programmers. Furthermore, the infrastructure is expected to provide policy management facilities.

4.2 Application programming interface (API)

Federations are managed at runtime by federation management services. Applications have access to the federation management functionality via common APIs which hide the complexities involved and promotes application portability. The application developer API is divided into three logical wholes

- Up-call API for federation management
- Down-call API for federation management
- Connector API for connection establishment

Each API is implemented as one or more interfaces. In this context interface means a logical entity which is realised either as an IDL-interface or a Java-interface in the prototype.

In the prototype the up-call API is implemented by each application component, the down-call API is implemented by the federation manager infrastructure component, and the connector API is implemented as a set of Java-classes which can be seen as a "connector library".

4.2.1 Up-call API for federation management

The up-call API consists of one interface `FederatedApplicationInterface`. Every application wishing to take part in federations must implement this interface.

The federated application interface contains operations for notifying applications of federation establishments and terminations. The notifications inform an application that it is invited to join a federation or that it has been removed from a federation. The exact behaviour resulting from the notifications depends on the application.

A single application always has exactly one federated application interface. If the application consists of multiple components (i.e. is a multi-component assembly) only one of the components implements the up-call API. How the notifications are processed within the assembly is application specific and hidden from the infrastructure.

4.2.2 Down-call API for federation management

The down-call API consists of three interfaces, `PolicyInterface`, `FederationInterface`, and `ChannelInterface`.

`PolicyInterface` provides operations for policy context management and is used whenever an application wants to create new policy contexts or manipulate the contents of an existing one. PolicyInterface provides generic operations for policy context manipulation as well as more specific "helper" operations for manipulating/accessing specific kinds of policy contexts for example service, offer, and federation contexts.

`FederationInterface` provides operations for federation management and is used whenever an application wants to create/join/leave federations. It provides operations for creating new run-time communities and establishing and terminating federations amongst those communities.

`ChannelInterface` acts as a contact point for retrieving interface references residing in federation contracts as well as initiating adapter instantiation processes. Channel interface is usually not directly used by application or adapter components but they use a technology specific "wrapper" called a connector.

4.2.3 Connector API

Connector API is used when resolving technology specific object references related to some established federation. The connector API contains one logical function `getConnection` for receiving object references. Connector API (i.e. getConnection-operation) is implemented separately for each supported technology. So far connectors for CORBA and RMI style object references are implemented. Client can choose which connector to use. In the prototype the EJB applications and adapters use RMI-connector API and CCM applications use CORBA-connector API.

There exists different versions of the `getConnection` function in the API. Which one of them is used depends on what information the API user has available. If, for example, a client application programmer has established a federation and needs to connect to one of the servers it used the version which is able to find the correct server reference from the federation contract and possibly instantiate adapters to the communication channel transparently. This is the default case.

The other versions are typically used in more specific situations like when an adapter or some special kind of application wants to establish inter-domain communication link to a pre-configured local server with no federations involved.

When used the function is given the set of parameters matching the signature used (federation context id, interface name and type in the default case) and it returns a technology specific object reference to the caller. In CORBA environment this implies returning an object of type `org.omg.CORBA.Object` and in RMI environment this implies returning an object of type `java.rmi.Remote`. The application or adapter in responsible for narrowing/casting these generic references to the correct technology specific type (e.g. PaymentInterface). This process could further be simplified by generating simple interface specific helper operations to the application components.

4.2.4 API usage scenario

A typical community creation and federation establishment scenario has the following phases from the API user point of view:

1. Application uses policy interface to create a service offer context for a specific service type. The infrastructure creates the corresponding context with "default" values and returns the id of the context to the application.

2. If the application has special needs it has the possibility to use the policy interface at this point for manipulating the policy values in the offer context.

3. Application uses federation interface to request infrastructure to populate a community according to the policy values in the offer context. Application is returned a sequence of federation context id's ordered according to applications preferences. There is one federation context id per populated community instance (i.e. federation).

4. At this point, application has the opportunity to retrieve and evaluate the contents of each federation context in order to decide which context(s) suite its needs. In case the application does not have any special needs it usually just selects the first id in the sequence.

5. Application then uses federation interface to establish federation(s) according to selected federation context(s). During the federation establishment procedure all the applications invited to the federation receive a federation establishment notification through the up-call API.

6. After successful federation establishment the application is able to use the services available through the federation. In order to use the services it retrieves the technology specific object references through which the services are accessed. If the application wants to use, for example, tourist info service then it uses connector API to resolve the interface of type `TouristInfoInterface`, named `touristInfo` in the service type used by the client. The application is returned an object reference which is bound to the tourist info service of the identified federation. During connection establishment all needed adapters are transparently configured into the communication channel. Whether a reference to a local adapter or to a remote server is returned is transparent to the client. After receiving the reference the client narrows/casts the received object reference to the type used in the program code and makes service requests using it.

4.3 Common management interfaces of applications

Applications willing to take part in federations must implement the interfaces common to all federated applications.

Each federated application component must implement the operations in the up-call API as well as implement the common configurable attributes (at the moment only application context id). In addition each application must have access to the down-call API and connector API. How this access is provided is implementation specific. In the prototype the access is provided via receptacle connections in CCM environment and via custom-made receptacles in the EJB environment.

The most effective approach would have been to implement the API as programming language specific library but this would have required a great deal more implementation effort than implementing the API using a standard middleware.

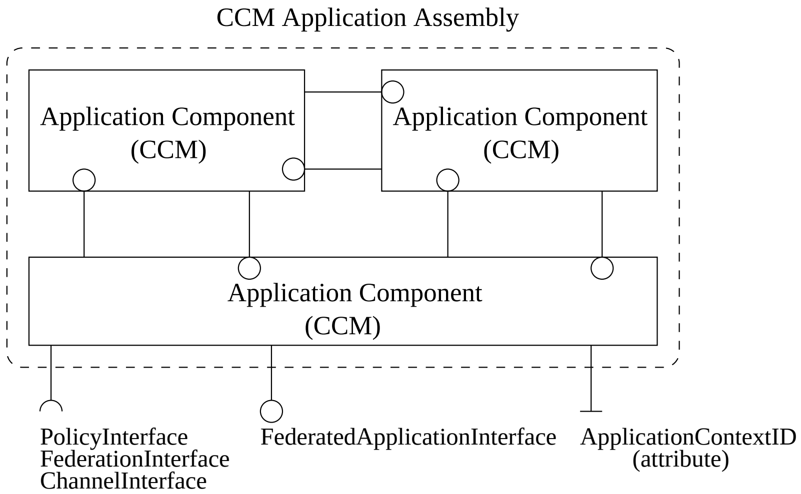

Figure 4.1: Common interfaces of all CCM-based federated applications.

Figure 4.1 visualises the common management interfaces of all CCM-based applications. It also visualises an application assembly which is the deployment and instantiation unit seen by the Pilarcos infrastructure. One assembly may contain several application components but the federated management interfaces need only be implemented per assembly basis not per component basis.

Since the infrastructure services are built using CCM, every non-CCM application must be provided with technology specific API adaptation layer. In the prototype this adaptation layer is implemented as application level adapters just as every other adapter. These API adapters are implemented as independent deployment units (i.e. assemblies). Figure 4.2 visualises the common interfaces of all EJB-based applications. Adapters are further discussed in section 4.4

It should be taken into account that the generic instantiation infrastructure is not fully implemented in the current prototype. For example the concept of assembly is, at the moment, more or less an abstract concept with no fully functional counterpart in the concrete implementation.

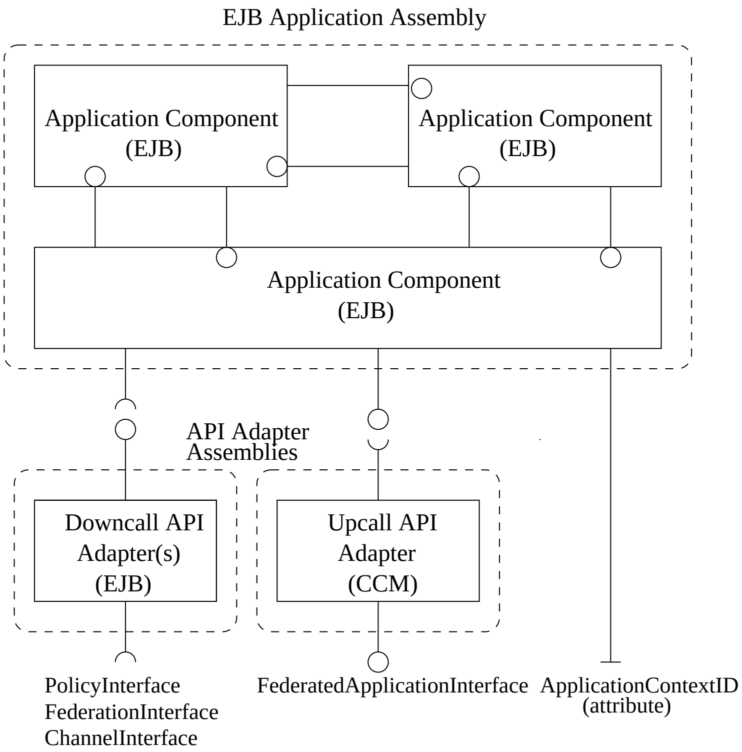

Figure 4.2: Common interfaces of all EJB-based federated applications.

4.4 Application services in the prototype

In order to facilitate prototyping of Pilarcos architecture a proper application scenario was needed. The tourist service -case was designed to provide complex enough environment to support the development of Pilarcos infrastructure services. The case does not specify a complete tourist service and must not be considered as such.

The tourist service -case specifies two business communities and four management domains. The four management domains are tourist client, tourist service, payment service, and hotel service.

The tourist client and payment server were both implemented as a single CORBA component. Tourist server was implemented as two CORBA components (tourist application and hotel client) residing in the same domain and locally connected together. Hotel server was implemented as two locally connected Enterprise Java Beans, one for service session management and one for making hotel queries and reservations. The concept "locally" meaning a non-federated intra-domain connection like normal CORBA or RMI connection.

Since components residing in the hotel info domain are EJBs and all other components are CORBA components there are two inter-domain adaptation points in the prototype. One between hotel info client and hotel info service and one between hotel info service and payment service. These adaptation points were implemented as two adapter components. CCM to EJB adaptation point was implemented as a CORBA component and EJB to CCM adaptation point was implemented as an Enterprise Java Bean. All CORBA components were implemented as (stateful) session components and all EJBs were implemented as

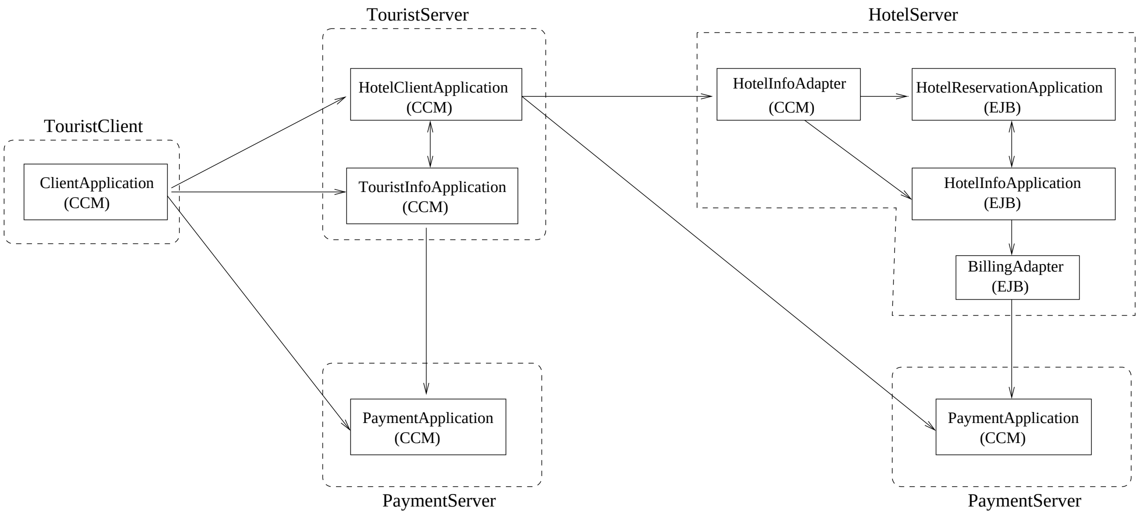

Figure 4.3: Application components (and adapters) in each community of the `Tourist Info Service`-case.

stateless session beans.

Figure 4.3 shows all application components and application adapters implemented in the prototype as well as their inter-domain communication relations.

4.4.1 Collaboration sequences of prototyped applications

In each of the two business communities the basic interaction scenario is almost identical. The client starts a service session and receives necessary service interfaces and possibly a pre-payment bill. Client then pays the pre-payment bill (if needed) and proceeds to service usage phase. When service usage phase is finished client retrieves and pays the post payment bills and ends the service session.

From the tourist info clients point of view the application interactions can be divided into four separate phases:

- Phase I - Start Session
- Phase II - Get Information
- Phase III - Make Reservation
- Phase IV - End Session

Each phase consists of several interactions involving components from both tourist info and hotel info communities. The following part of the section gives a detailed overview of each phase.

Phase I - Start Session

Client starts the service usage by initiating a service session to tourist info service. Service session is an application state which roughly equals to the situation resulting from a login procedure. As a result of the start session phase the client receives object references to the components implementing the actual service provisioning logic. In addition client receives pre-payment bills which must be paid before proceeding to actual service usage.

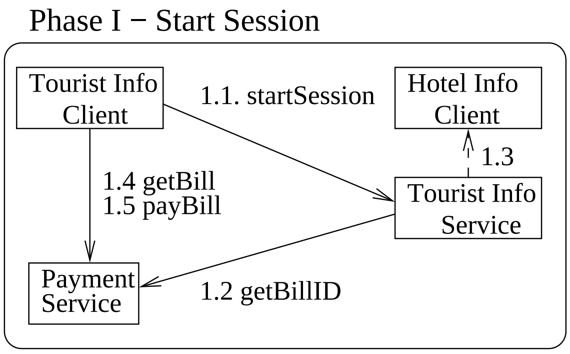

Figure 4.4: Application interactions in Start Session -phase.

Service session is logically independent of the federation concept but in the prototype one federation may only have one simultaneous service session. This restriction was made for ease of implementation. It does not affect the development of the Pilarcos infrastructure services in any way but it simplifies the application components.

Start Session -phase consists of five interactions:

1. Tourist client starts a service session to tourist info server
2. Tourist server generates a pre-payment bill and stores it to payment server. It receives bill id from the payment service.
3. Tourist server gets reference to needed local subcomponents and returns their object references and the pre-payment bill to the client.
4. Tourist client fetches the pre-payment bill from the payment server and verifies that it is not corrupted in any way.
5. Tourist client pays the verified pre-payment bill.

Figure 4.4 shows the application collaborations.

Phase II - Get Information

After a service session is started and pre-payment bill is paid the client can proceed to information gathering. The client makes queries in order to find a suitable hotel. Before processing the queries the tourist info server must verify that the client has properly started a service session and that all pre-payment bills are paid.

If everything seems to be in order then tourist info service processes the client query, possibly contacting several hotel services, starting service sessions to them and then querying information from them. After receiving the information from the hotel services it combines the results to a coherent whole for the client.

Get Information -phase consists of six interactions:

1. Tourist client asks for hotel information from the hotel client which is the subcomponent implementing the contact point to the hotel service.

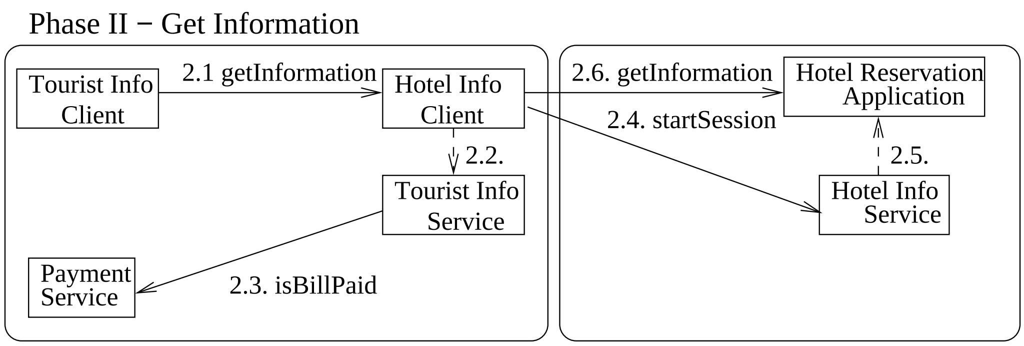

Figure 4.5: Application interactions in Get Information -phase.

2. Hotel client asks tourist server to verify whether the pre-payment bill is paid for this service session.

3. Tourist server contacts payment server and asks whether the tourist client has paid the bill and returns the information back to hotel client.

4. If prepayment bill has been properly paid then the hotel client proceeds to starting a service session to all relevant hotel services.

5. Hotel service gets reference to needed subcomponents and returns their object references to the hotel client. Hotel service does not require any kind of pre-payment for querying information.

6. After receiving the references the hotel client makes queries for hotel information, combines the results and returns the combined query to the tourist client.

Figure 4.5 shows the application collaborations.

Phase III - Make Reservation

When the tourist info client has received and processed all received hotel information it contacts tourist info service and asks for a hotel reservation. Tourist info service forwards the request to the correct hotel info service. Hotel info service generates a bill for the reservation and sends it to the payment service. After receiving the bill id from the payment service the tourist info service fetches the payment data of the reservation from the hotel info service. It then generates its own bill which is a copy of the hotel info bill plus the added fee that the tourist info service bills for itself. This bill is registered to the payment service in the tourist info community.

After paying for the reservation the tourist info client assumes that the hotel is reserved. Actually the hotel reservation is in a pre-committed phase. Tourist client has paid for the tourist info service but the tourist info service has not yet paid for the hotel info service. The tourist info service will pay all the bills of the reserved hotels (there may be several reservations in one service session) and bills from other possible business transactions at a later stage in the End Session -phase. It is only then that the reservations become fully committed from the hotel service provider view.

Make Reservation -phase consists of eleven interactions:

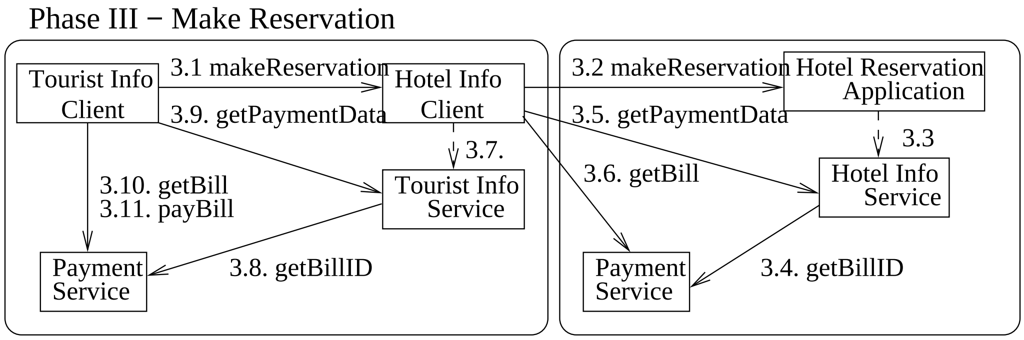

Figure 4.6: Application interactions in Make Reservation -phase.

1. Tourist client asks hotel client to perform a hotel reservation.
2. Hotel client proceeds in making the reservation to hotel reservation application.
3. Hotel reservation application asks hotel service application to generate a bill for the reservation.
4. Hotel service application sends the bill to the payment service and receives a corresponding bill id.
5. Execution comes back to hotel client which fetches from the hotel reservation application the payment data identifying the reservations made.
6. Hotel client uses the bill id in the payment data to fetch the reservation bill from the payment service and generates its own post-payment bill which is a copy of the hotel reservation bill added with some additional fee.
7. Hotel client asks the tourist service to send the generated bill to payment service.
8. Tourist service sends the generated bill to payment service and receives a bill id.
9. Execution comes back to tourist client which asks for the hotel reservation related payment data.
10. Tourist client uses the bill id in the payment data to fetch the bill from the payment service.
11. Tourist client pays the hotel reservation bill to payment service.

Figure 4.6 shows the application collaborations.

Phase IV - End Session

After tourist info client has finished with all reservations it tries to close the service session to the tourist info service.

After checking from the payment service that all bills of the tourist info client are properly paid the tourist info service proceeds in paying the bills of every sub-service tourist info client has used. It then tries to end the service sessions to all the sub-services.

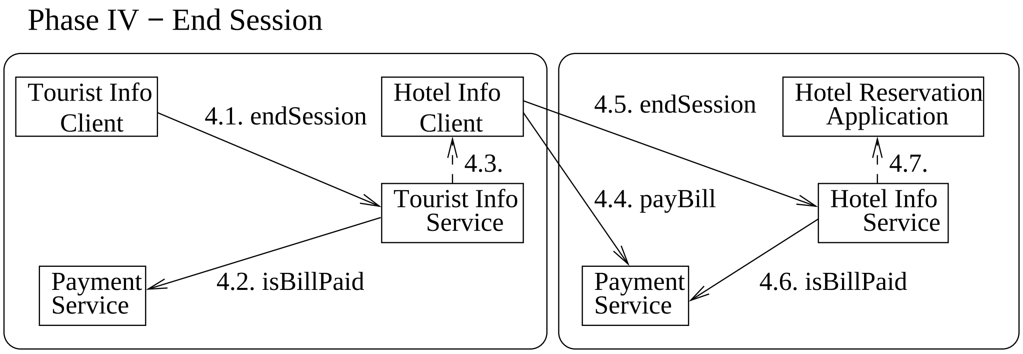

Figure 4.7: Application interactions in End Session -phase.

Sub-services, like the hotel info service, checks whether all the bills related to the service session have been paid and if yes then finally commits the business interactions.

End Session -phase consists of seven interactions:

1. After paying the hotel reservation bill, tourist client asks tourist service to end the service session.
2. Tourist service checks from payment service whether all the clients bills are paid.
3. If client has paid all its bills the tourist service informs hotel client about the ending of the service session.
4. Hotel client pays all the hotel reservation bills related to ending the service session.
5. After paying the hotel reservation bills the hotel client asks the hotel service to end the hotel service session.
6. Hotel service checks whether hotel client has paid all the hotel reservation bills.
7. If all bills are properly paid the hotel service notifies the hotel reservation application about the ending of the service session.

Figure 4.7 describes the application collaborations.

4.5 Application dependent support elements

The most important application dependent support elements are the adapters. Adapters can be divided into 'vertical' API adapters and 'horizontal' application communication adapters. All adapters are implemented as application level services regardless of their intended usage. Technically adapters and applications do not differ from each other. The only difference is that applications must implement the federated management interfaces required from a federated application. There exists no fundamental differences between client- and server-side adapters either.

Adapter adapts one or several interfaces of its native technology to one or several interfaces of the target technology. For example CCM-based adapters adapt from CCM-technology domain to some other technology domain like EJB-domain. In addition to

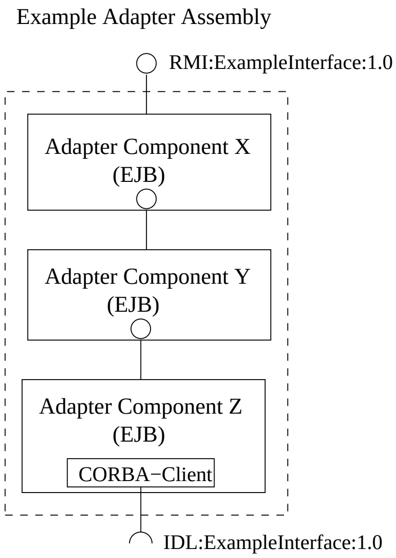

Figure 4.8: Example of multi-component EJB to CCM adapter.

adaptations between technology domains, semantic or syntactic adaptation can be done within one technology domain.

Adapters are combined into adapter assemblies. Adapter assemblies represent one logical adaptation 'whole' and may contain several adapters components. The idea is to construct adaptation units for a specific purpose, combine them into an assembly with a unique name, and use this assembly as a selection, deployment, and life-cycle management unit. The assembly may contain technology adapters, semantical adapters, syntactical adapters and so on. The important thing is that the assembly is managed as a single unit by the infrastructure.

Figure 4.8 visualises an example adaptation unit which contains an EJB to CCM adapter in addition with additional adaptation component for arbitrary syntactic and semantic adaptations. All adapters implemented in the prototype consist of single adapter component, however.

Chapter 5

Infrastructure services

In this section, the new Pilarcos middleware services are discussed in detail. The functionality and interactions of Pilarcos trader, type repository, policy repository, federation manager and various factories are described. The text focuses on the current prototype implementation.

5.1 Policy repository

Policy repository is the component that stores policies of the organisation centrally in a hierarchy of policy contexts (Section 3.6). It provides interfaces for managing and looking up policies, and ensures the integrity of the policy context hierarchy. This section covers the interfaces and the functionality offered by the policy repository implementation, and briefly discusses possible future enhancements.

5.1.1 Interfaces

In the prototype, the policy repository provides just one interface: the policy context management interface. The interface is used directly only by the federation manager. Other components, including federated applications, use a similar interface offered by the federation manager, which delegates the operations to the policy repository.

The policy context management interface provides functionality for managing policy contexts within a hierarchy. Policy contexts are referenced by unique policy context identifiers, which are issued by the policy repository for new contexts. Operations are provided for creating policy contexts, traversing the policy context tree, reading and changing policy values and deleting policy contexts.

5.1.2 Functionality

The prototype version of the policy repository maintains the policy context trees of the organisation, of which there may be several — one per application, since policy context levels above the application level have not been implemented. The policy repository guarantees the uniqueness of policy context identifiers and the validity of the policy context trees even in the presence of concurrent modifications.

Ensuring the compatibility of lower level policies to the upper level policies in the hierarchy has not been implemented. Therefore, the current policy repository implementation cannot detect inconsistencies or violations of policies within a policy context tree.

5.1.3 Future enhancements

As mentioned in Section 3.6, policy context types are not implemented. They are essential for implementing policy integrity checking and supporting a single organisation-wide policy context tree. A possible design for policy context types is sketched out below.

The type of the policy context defines the policy frameworks of the policy context. In likeness to service types (Section 3.3), the policy frameworks within a policy context type definition are named uniquely.

Every distinct level in the policy context hierarchy needs to have its own policy context type; in some cases, even policy contexts on the same level can have different policy context types. The policies within a policy framework of a policy context are constrained by all policy frameworks of the same name in its ancestor contexts. For example, the policies in a policy framework named foo in the root policy context would place constraints on all of the organisation's policy frameworks named foo. (Of course, the link between policy frameworks could be realised by other means than name equality, too.)

The above structure needs to be augmented with a mechanism within the policy repository that checks policy integrity within the tree on updates. Obviously, such a mechanism needs to perform well even with frequent updates, and thus poses a design and implementation challenge.

In addition to introducing policy context types, integrity checking and a unified policy context tree, the implementation technique for the policy contexts should be rethought. From applications' point of view, a clean way would be to make policy contexts real CORBA objects. This poses its own implementation difficulties, the least of which is not adaptation of interfaces in a heterogeneous environment. Additionally, it would be unnecessarily expensive to have all accesses to policy contexts to be remote. Some kind of local proxy objects would be needed.

5.2 Federation manager

The federation manager is the central component in the Pilarcos middleware architecture. Each administrative domain has a federation manager that negotiates federations with federation managers in other domains, and coordinates federation establishment within its own domain.

Within a federation, the federation manager in the client (initiator) domain acts as the federation coordinator. It mediates federation negotiations and keeps a primary copy of the federation contract. Replicas of the contract are also kept by the other federation managers, so that any of them can take over the coordinator role in case of failure. (In the prototype, changing the federation coordinator has not yet been implemented.)

5.2.1 Interfaces

The federation manager provides and uses several interfaces as shown in Figure 5.1. Application components access the Pilarcos infrastructure services only through the application programming interfaces (APIs) provided by the federation manager. These APIs, shown uppermost in the figure, were discussed in Section 4.2. This arrangement simplifies implementation in technologically heterogeneous environments, since only one adapter between an application component and the Pilarcos infrastructure needs to be used. The federation manager also provides more friendly interfaces for applications than the raw Pilarcos services, which are not designed to be used directly by applications.

The `FederationManagement` interface is used for communication between federation managers, and contains operations for establishing and terminating federations. The `FederatedNaming` interface provides a federation-aware naming service that is used to resolve interface references between domains. These are the only communication points used between Pilarcos middleware services in different administrative domains.

`ServiceManagement` and `ChannelManagement` are provided by the local service factory, and used by the federation manager for service and channel instantiation. The federation manager also uses the local policy repository via the `PolicyContextManagement` interfaces. The `Lookup` and `TypeRegistration` interfaces are used to access the Pilarcos trader and the type repository, respectively, for populating architectures and enquiring type information.

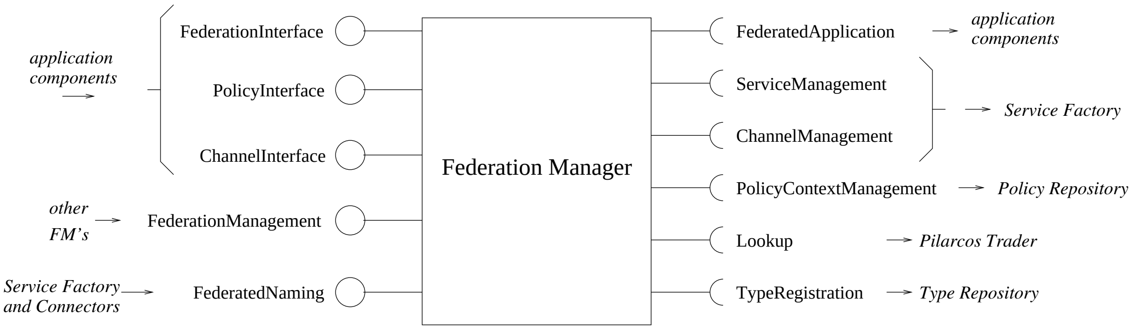

Figure 5.1: Interfaces provided and used by the federation manager.

5.2.2 Federation contracts

The federation manager essentially manages federation contracts (Section 3.5). Federation contracts are constructed by the Pilarcos trader (Section 5.5.6), and become final contracts after they are accepted in all federation managers participating in the federation. Between administrative domains, federation contract identifiers are used to identify federations, since the Pilarcos trader guarantees them to be unique.

Application components, however, do not directly see federation contracts. Instead, they use policy contexts (Section 3.6) and the APIs provided by the federation manager for populating architectures, establishing federations and resolving interface references. The federation manager constructs the initial incomplete federation offer for populating, and maintains a mapping between federation contracts and federation contexts thereafter, hiding contracts from the applications altogether.

5.2.3 Population process

The population process is initiated by a client application by creating an offer policy context and issuing a population request to the federation manager, as described in Section 4.2.4. The federation manager creates an initial, incomplete federation offer by filling in the client's service offer, which is constructed on the basis of the policies in the offer policy context. The federation manager then issues a population request to the Pilarcos trader, asking it to fill in the empty roles in the federation offer. After receiving federation contracts (that is, complete federation offers) from the trader, the federation manager creates federation policy contexts corresponding to the contracts, and hands the contexts back to the application. This process is illustrated in Figure 5.2.

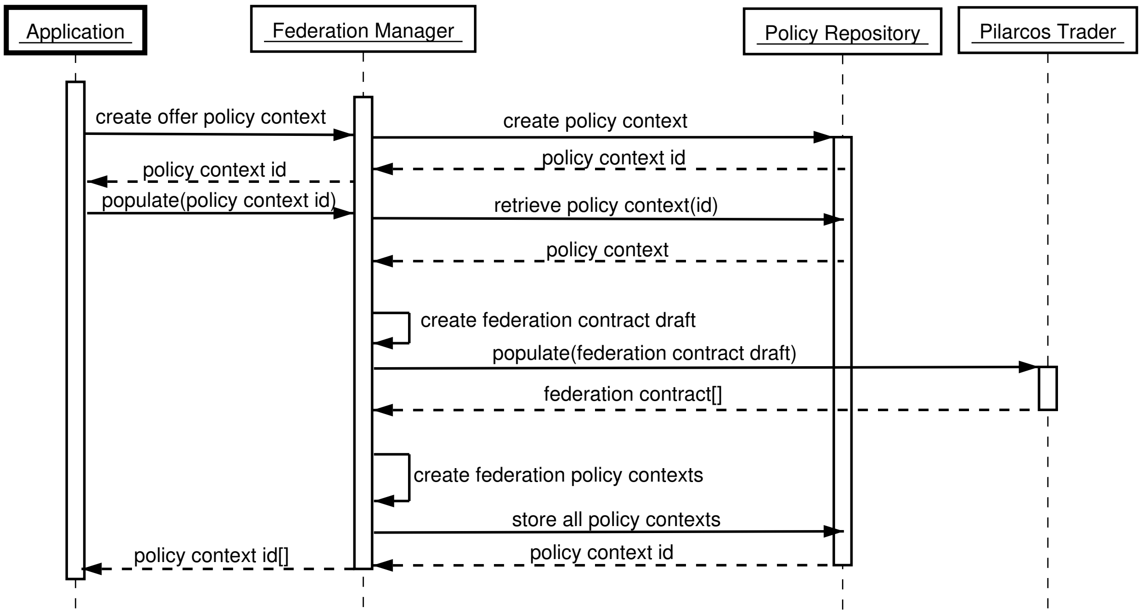

Figure 5.2: Population process in the federation manager.

5.2.4 Federation establishment

In the Pilarcos infrastructure, the local service factory (Section 5.3.1) instantiates and destroys instances of application components according to their life-cycle policies. The federation manager issues an instantiation request for an application to the service factory when a new federation is established; this request is identified by a "*cookie*". The cookie is used later for the reverse operation when the federation is terminated. The federation manager does not know how many instances of an application component exist at a time. A cookie may be associated with a new application component or not, depending on the life-cycle policy.

A client application initiates federation establishment by issuing a create-federation request to the local federation manager, with a federation context as a parameter. An overview of the federation establishment process in the client (initiator) domain is shown in Figure 5.3. For symmetry, the federation manager requests the service factory for an instance of the client application, although it already exists, and receives a cookie for it. An association between the cookie, the federation contract and the federation context is made in the data structures of the federation manager as well as the service factory. After this, the federation manager begins negotiation with remote federation managers by issuing federation establishment requests to them. The federation manager in the client domain automatically becomes the federation coordinator.

The negotiation, which at present includes only one request that may be denied or accepted, proceeds concurrently with all parties. The negotiation is an atomic operation, since if any participant denies or does not respond, the federation cannot be established. For efficiency, two-phase commit is not used, so the federation coordinator must send

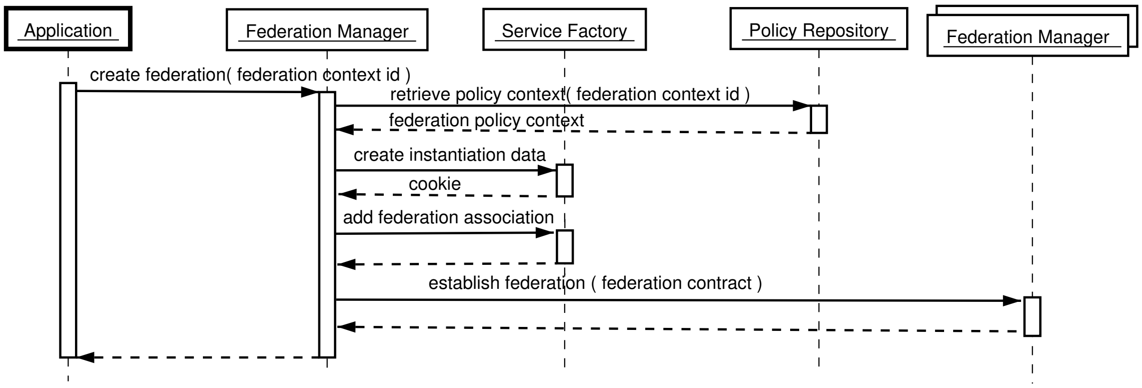

Figure 5.3: Federation establishment on client side.

explicit rollback requests if the negotiation fails. Even then, the protocol may fail if the federation coordinator fails.

When the federation managers in server-side domains receive the federation establishment request, they start the local establishment process, as illustrated in Figure 5.4. The process begins with an application instantiation request to the local service factory. A federation policy context is then created for the federation. The service factory compares the federation contract with the currently running application instance, and instantiates adapters if needed. Adapters, too, can have different life-cycle policies, which are heeded by the service factory. Finally, the federation manager notifies the application about the new federation through the up-call notification API.

The federation creation request from the client application returns only after the federation establishment has been successfully completed, or it has failed.

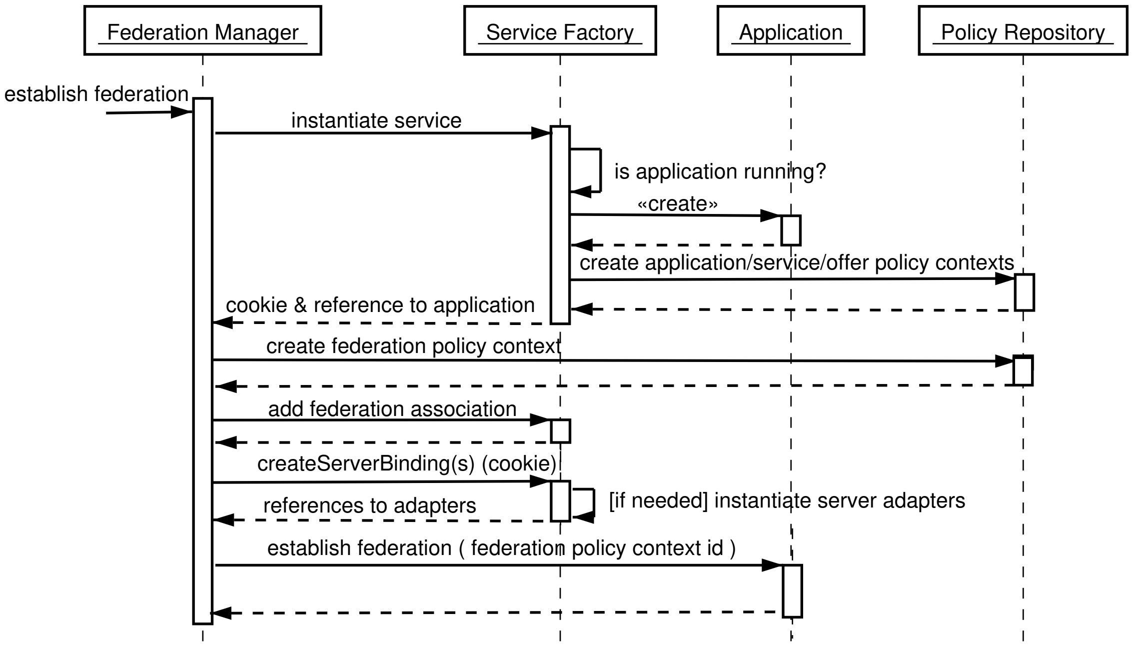

Figure 5.4: Federation establishment on server side.

5.2.5 Federation termination

Any application that participates in a federation may issue a request to leave the federation to the federation manager. In the prototype, this causes federation termination requests to be issued to all other federation managers concurrently. On receiving it, each federation manager notifies the local application component about the termination through the up-call API. After this, the federation managers request the local service factories to remove the application instance and associated adapters, which are identified by the cookie.

5.3 Factories and connectors

Factory services are responsible for managing the life-cycle of federated applications and their support elements (e.g. adapters). Connectors implement the connector API and are responsible for establishing inter-domain connections with the support from factory services.

Life-cycle of federated applications can be divided into five phases

1. Application instantiation
   - Server-side application is instantiated either at its deployment time or at run-time during federation establishment. This depends on the life-cycle policy selected ('conservative' for pre-instantiation vs. 'federation' for runtime instantiation). If server-side application is instantiated at deployment time it only needs to be reconfigured at federation establishment time.
   - Client-side application is instantiated at its deployment time when someone needs to use it.
2. Adapter instantiation
   - Server-side adapters are instantiated at federation establishment time if the technology or interfaces provided by the server application differ from those in the agreed federation contract. If this is the case then an assembly containing one or more adapter components is instantiated and bound to server application. If the adapters are capable of handling many-to-many connections it is possible to have only one instance of an adapter capable of adapting multiple applications. In this case the instantiation just means connecting the server application to the existing adapter.
   - Client-side adapters are instantiated on-demand at service usage time. After federation establishment the client application can use the connector API to receive object references it needs to use. If the interface used by the client differs from the one agreed in the federation contract this results in automatic instantiation of a proper adapter assembly. Like with server-side adapters the process of selection and instantiation of a client-side adapter may result in a new adapter instance or just reconfiguration of an existing adapter.
3. Inter-domain connection establishment
   - After needed client-side adapters are instantiated the connector proceeds with the inter-domain connection establishment. At minimum, the connection establishment means making a lookup to the federation-aware naming service located

in the server domain. If direct binding is used then the federation-aware naming service returns the object reference to be used by the client. If in-direct binding is in use the reference returned by the federation-aware naming service is in-direct and contains a <name, technology specific naming service> pair. In this case another lookup from the technology specific naming service is needed.

4. Adapter deletion

   - Server-side adapters are deleted at federation termination time. If the adapter life-cycle policy is such it exceeds the federation life-cycle then the deletion means only re-configuring the adapter.
   - Client-side adapters are deleted at federation termination time similarly to server-side adapters. If the adapter life-cycle policy is such that exceeds the federation life-cycle then deletion means re-configuring the adapter exactly like in the server-side case.

5. Application deletion

   - Server-side applications are deleted either at federation termination time or by an administrative decision. This depends on the applications life-cycle policy.
   - Client-side applications are always deleted by an administrative decision. Their life-cycle is independent of the federation life-cycle.

5.3.1 Service Factory

Factories are used for instantiating application and adapter components. In Pilarcos architecture there exists two generic life-cycle management interfaces `ServiceManagement` and `ChannelManagement`. Service management interface is used to manage the life-cycle of instances of service applications. It provides operations for instantiating, configuring and deleting the application components. Channel management interface is used to manage the life-cycle of client- and server-side bindings (i.e. managing the life-cycle of adapters). It provides operations for instantiating and deleting adapter components.

The management interfaces are implemented in one monolithic factory component called `ServiceFactory`. Service factory provides the generic service and channel management interfaces and can be used in all management domains in the prototype. The service factory implementation is generic in the context of the Pilarcos prototype but to make it truly generic in all contexts would require full specification of generic assembly descriptors. Defining them is left for future development. Whether it would be better to implement the channel and service management interfaces in separate channel and service factory components is also left for future consideration. Service factory utilises platform specific factories (like component homes) to instantiate the components in different technology domains like EJB and CCM.

Service factory is normally used during federation establishment procedure by federation manager to instantiate services on-demand. Service factory can also be used to instantiate service components before federation establishment time or, for example, federative clients.

It should be noted that the concepts of (service) instance and (service) instantiation have separate meanings for a service factory. Instance is a concrete runtime object created as a result of an instantiation. Instances of applications are identified with unique application policy context id's. Instantiation, however, may not always produce a new concrete instance. Instantiation results in a logical instantiation object which is identified by a cookie generated by the factory. The logical instantiation object may be bound to an existing application instance or a new application instance can be created for it.

When a new adapter is created it is bound to the logical instantiation object and implicitly to an application object. If adapter has a conservative life-cycle policy then "adapter creation" just means binding some existing adapter instance to the logical instantiation object.

When a service is deleted the logical instantiation object is destroyed but whether the concrete application and adapter instances bound to it are destroyed depends on their life-cycle policies.

Interfaces

As mentioned above the service factory provides two management interfaces. One for application and one for binding life-cycle management. In addition to them it provides a configuration interface called `FactoryConfiguration`. Factory configuration interface is used, for example, to configure the available assemblies and locations of naming services to the service factory.

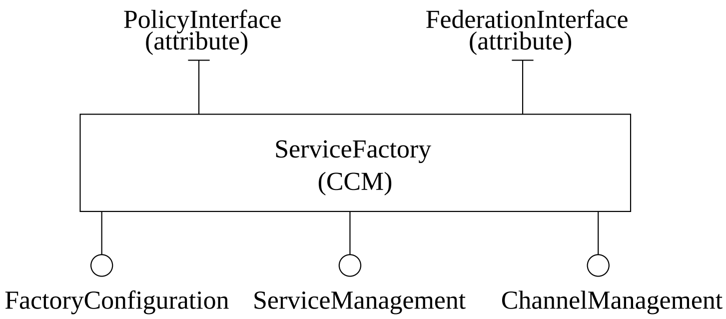

Figure 5.5: External dependencies of service factory.

In addition to providing the interfaces, service factory also provides two attributes `PolicyInterface` and `FederationInterface`. They are used for storing the references of the down-call API-interfaces to the service factory. Service factory connects these references to the application components at their instantiation time.

Figure 5.5 visualises the external dependencies of the service factory.

5.3.2 Connectors

Connector is a client side entity which is used by applications and adapters to resolve federated interface references into technology specific object references. Connector is typically used by applications after federation contract has been negotiated and application wants to resolve the object references of the servers it needs to connect to. The connector uses the channel interface API to create client side bindings. If adapters are created as part of the client side binding they also use a connector to perform the intra-domain connection establishment.

A connector is implemented separately for each platform used (e.g. CORBA, RMI) since it needs to be able to resolve object references in technology specific format.

Implementation

Connector is implemented as a Java-class providing a static operation `getConnection`. Operation getConnection is used when resolving object references from foreign domains. There are several overloaded versions of this operations which take in different parameters.

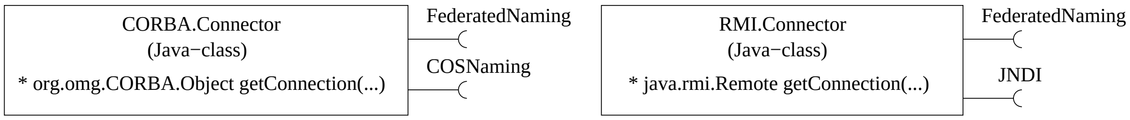

Figure 5.6: External dependencies of connector.

Since connection factories are technology specific the `getConnection` operations signatures vary between platforms. For example CORBA connector returns CORBA references whereas RMI connector returns RMI references.

Figure 5.6 visualises the external dependencies of the connector.

5.3.3 Application and adapter selection

Applications and adapters can be selected and their life-cycle managed at runtime. This requires information and related algorithms in the infrastructure.

In the current Pilarcos prototype the problem of selection and instantiation is resolved with the introduction of instantiation units, assemblies, and by defining what their relationship is to the federation contract.

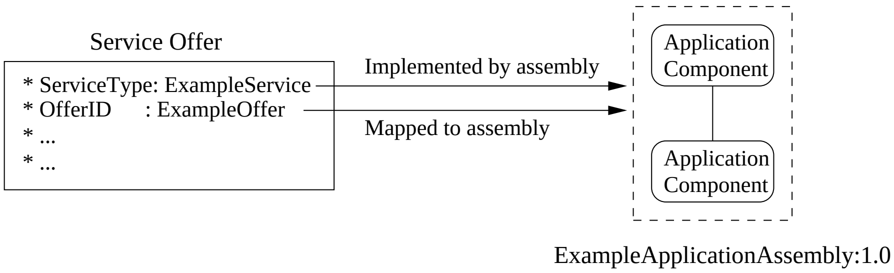

Figure 5.7: An example of a simple application selection mapping.

Both applications and adapters are logically grouped to assemblies. Before a service provider exports a service offer to the Pilarcos trader it maps a specific application assembly to this service offer and registers the relationship to service factory. Whenever a request to establish a new federation arrives to the domain the service factory knows from this relationship which assembly needs to be instantiated and/or configured to provide the service.

Figure 5.7 shows an example of how applications are mapped to service offers.

As far as the adapters are concerned the situation is a bit more complicated. What adapters fundamentally do is that they make adaptations between two different realisations of one interface type.

Each realisation of an interface type is given a unique name. An adapter assembly must be mapped for every 'realisation-realisation' pair that needs adaptation. The adapter assembly can be constructed from several independent adapter components and same adapter

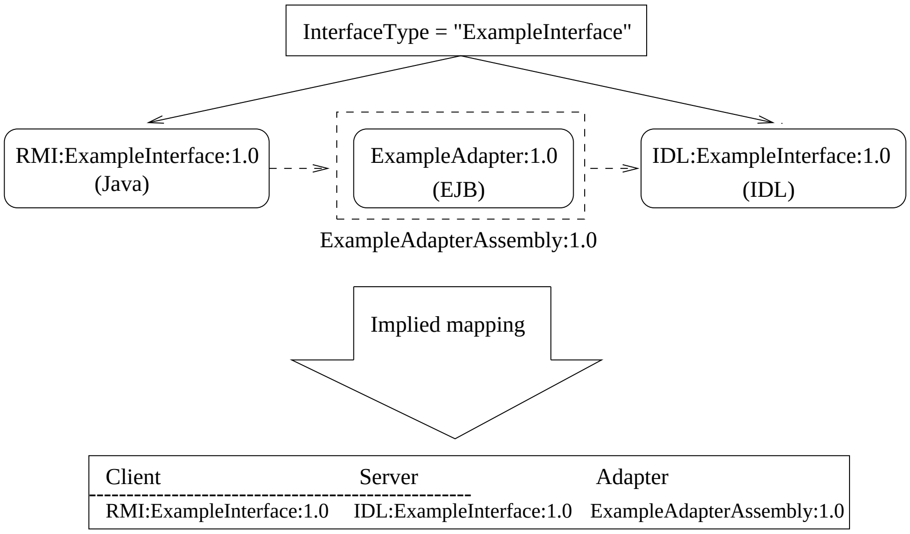

Figure 5.8: An example of a simple adapter to service offer mapping.

components can be used in different assemblies. The service offers of each service provider and service requester contain the identifiers of the interface type realisations they are using in their domain. This enables the life-cycle management services to compare the realisation used by the client and the realisation used by the server in each binding and, in case they differ select, instantiate and/or configure a proper adapter assembly in to the communication channel.

Figure 5.8 shows an example of how adapters are mapped to service offers.

5.3.4 Direct and in-direct referencing

In order to support object reference transportation across technology boundaries a generic, technology-independent, reference is needed. Most convenient way to do this would be to have a generic object reference format to and from which other formats could be mapped.

In the prototype there exists two types of object references, CORBA inter-operable object references (IORs) and RMI remote references but there exists no common format which these references can be mapped to. This forces us to use an in-direct referencing approach where an object reference is first stored to a naming service supporting its referencing style (e.g. COSNaming and JNDI) and then the the address (URL) of the naming service and the name inside it are stored into an in-direct object reference. The in-direct object reference consists of two strings and can be passed to all technology domains supporting strings.

The in-direct object referencing is not, however, performance effective. For this reason another referencing approach, direct referencing, is also supported. Direct-referencing can be used if the object referencing format agreed in the federation contract is a 'native' format. In the existing prototype the federation management services are in CORBA technology domain so whenever servers are referenced with CORBA IORs the direct referencing scheme can be used. Otherwise in-direct referencing must be utilised.

At the server side either an in-direct or a direct reference is created when federation is established. This depends on whether a 'native' or 'non-native' technology has been agreed in the contract. The federation contract contains the reference to the federation-aware naming service where the created reference is stored and information whether a direct or in-direct naming scheme is used. Client-side connector is able to find out at connection establishment time whether to run an in-direct or direct establishment protocol.

If direct protocol is effective the client connector resolves a native (which in this case is CORBA) reference from the federation-aware naming service and if an in-direct protocol is effective then the client connector first resolves the in-direct reference, contacts the technology specific naming service and resolves the concrete object reference using the information in the in-direct reference.

The dynamic nature of service instantiation system causes the fact that the additional in-direction resulting from the use of federation-aware naming service cannot be further optimised. This would only be possible if pre-instantiated services with no adaptation were the only ones allowed. In this case the reference to server could be created before federation establishment time and stored to trader at service offer export time (like is situation with conventional trading). If more dynamic behaviour is to be supported one in-direction round is inevitable.

5.3.5 Life-cycle scenarios

In order to illustrate the behaviour of the infrastructure components during the different phases of the application and binding life-cycle a few exemplary scenarios are provided. Scenarios show examples of client- and server-side application instantiation, client- and server-side adapter instantiation and inter-domain connection establishment.

Client-side application instantiation scenario

Figure 5.9 shows an exemplary instantiation sequence which is further explained below

1. Service factory instantiates, configures, and connects client component(s). It uses technology specific factories (e.g. component homes) to assist in instantiation.

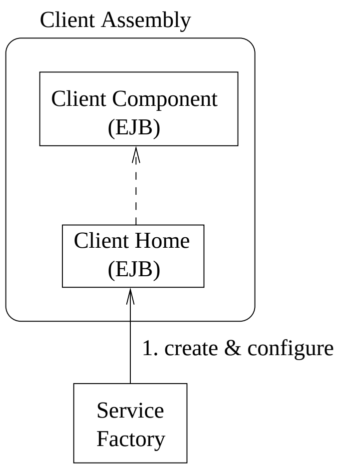

Figure 5.9: Client-side application instantiation scenario.

Server-side application instantiation scenario

Figure 5.10 shows an exemplary instantiation sequence which is further explained below

1. After `establishFederation` request has arrived to federation manager it asks service factory to create the service according to federation contract.

2. Service factory instantiates, configures, and connects server component(s). It uses technology specific factories (e.g. component homes) to assist in instantiation. If an instance of the server is already up, the factory decides, based on the life-cycle policy of the server, whether a new instance is needed or whether the old instance should only be re-configured.

3. Service factory binds the object reference(s) of the server component to local naming service (e.g. JNDI). The object reference can be bootstrapped from the naming service if in-direct referencing is in use.

4. Service factory binds the object reference(s) of the server component to federation-aware naming service implemented by the federation manager. If the server application is implemented on top of the 'native' middleware platform (which in the prototype is CORBA) a direct and in-direct version of the object references are stored. If server application is implemented on top of other middleware (like RMI) then only in-direct reference is stored (direct referencing cannot be used).

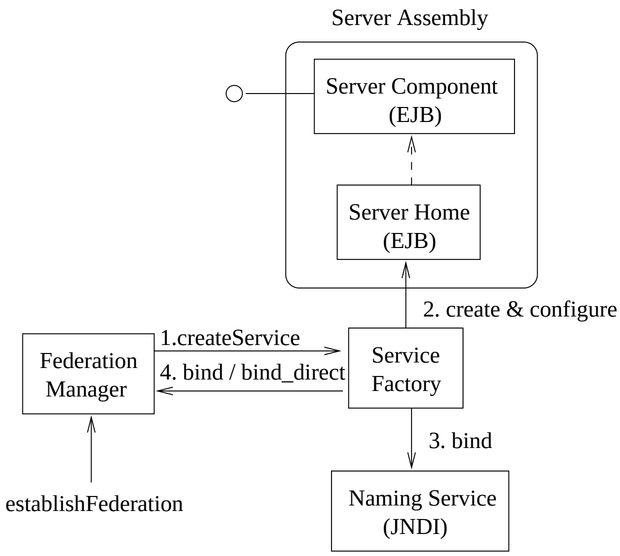

Figure 5.10: Server-side application instantiation scenario.

Server-side adapter instantiation scenario

Figure 5.11 shows an exemplary instantiation sequence which is further explained below

1. After server application is instantiated federation manager asks the service factory to instantiate server-side bindings. This is done for each interface separately.

2. Service factory checks whether the interface realisation provided by the server application matches the one agreed in the contract. If it does not then the service factory instantiates, connects, and configures a proper adapter. In case a proper adapter is already instantiated the service factory decides, based on the life-cycle policy of the adapter, whether a new instance is needed or whether it should only re-configure the old instance.

3. Service factory binds the object reference(s) of the adapter to local naming service (e.g. JNDI). The object reference can be bootstrapped from the naming service if in-direct referencing is in use.

4. Service factory binds the object reference(s) of the adapter to federation-aware naming service. If the adapter is implemented on top of the 'native' middleware platform a direct and in-direct version of the object references are stored. If adapter is implemented on top of other middleware then only in-direct reference is stored.

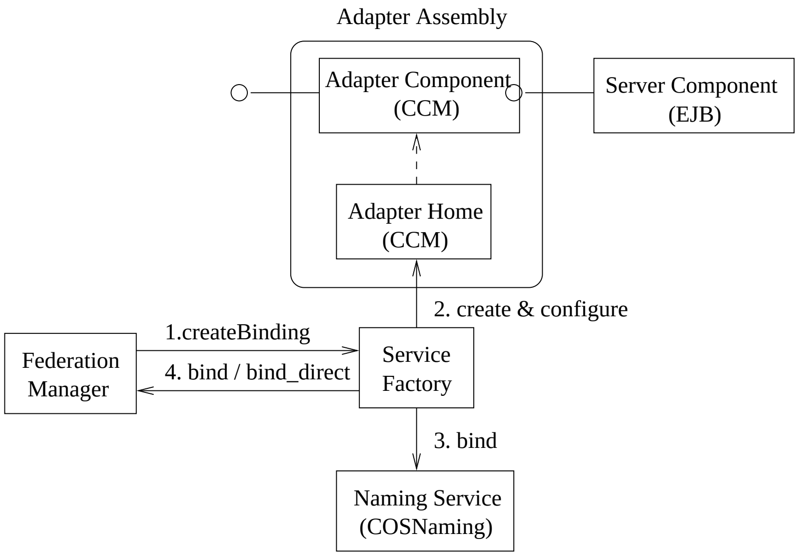

Figure 5.11: Server-side adapter instantiation scenario.

Client-side adapter instantiation scenario

Figure 5.12 shows an exemplary instantiation sequence which is further explained below

1. Client application requests an object reference from the technology specific connector it is using. The connector forwards the request to federation manager using the channel interface in the down-call API.

2. Federation manager asks service factory to create client-side binding. Service factory compares the interface reference found in the federation contract to the interface reference used by the client and decides whether an adapter is needed. If adapter is not needed then service factory returns the interface reference stored in the contract.

3. If an adapter needs to be instantiated, the service factory instantiates, connects, and configures a proper adapter assembly. Whether a new adapter needs to be instantiated or an existing instance can be used depends on the life-cycle policy of the selected adapter.

4. If the adapter is implemented with the 'native' middleware technology the service factory registers the object reference(s) of the adapter using the federation-aware naming interface provided by the federation manager. This is an effective way to make the reference available for the connector. If the adapter is implemented in some other middleware technology an alternative method must be used and the object reference(s) of the adapter must be bound to the technology specific naming service (e.g. JNDI) and an in-direct reference must be registered to the federation manager.

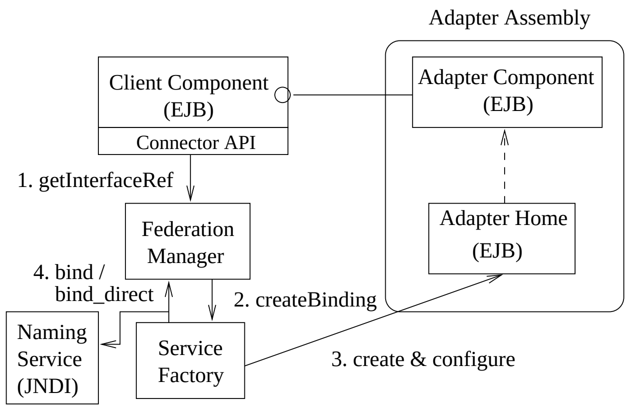

Figure 5.12: Client-side adapter instantiation scenario.

Inter-domain connection establishment scenario

Figure 5.13 shows an exemplary connection establishment sequence which is further explained below

1. The connector starts the connection establishment algorithm by checking whether it needs to resolve a direct or in-direct interface reference. Then it contacts remote federation-aware naming server and retrieves the interface reference to be used. If direct referencing is in use, the federation-aware naming service returns the technology specific object reference (e.g. CORBA reference) to the caller. If in-direct referencing is used the federation-aware naming service returns an in-direct object reference containing a naming service URL and a name.

2. If direct referencing is in use, this phase is skipped. If in-direct referencing is in use the connector resolves the direct object reference from the technology specific naming service.

3. The connector returns the object reference to adapter (or application depending on which initiated the call) which narrows it to correct programming language type and uses it to make service calls.

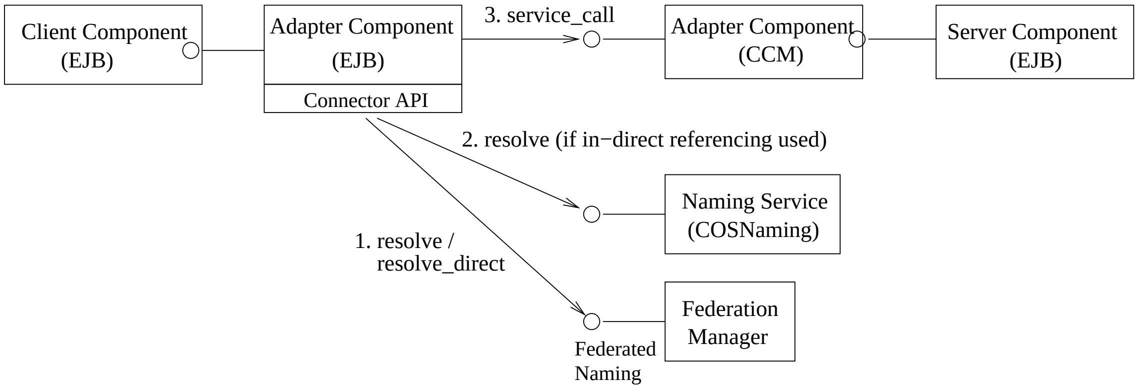

Figure 5.13: Inter-domain connection establishment scenario.

5.4 Type repository

Type repository is a global component in the Pilarcos prototype, meaning that there is only one shared type repository. According to the Pilarcos architecture plan, there would be local type repositories in each domain, which would be federated for cooperation. However, federation of type repositories has not been implemented.

The type repository is used for registering different kinds of types. Currently type repository is used by Pilarcos trader and the federation manager.

Type repository stores currently three different types. These types are policy framework type, service type and business architecture. Type repository uses three different data structures to store registered types in main memory, one for each type.

Type repository uses the CORBA Service Type Repository service to store service types for the CORBA trader. This because Pilarcos trader uses CORBA trader as a back end. Pilarcos service types are transformed to CORBA service types before storing to CORBA trader.

5.4.1 Interfaces

Type repository provides one interface to describe and register types. This interface is mainly used by Pilarcos trader. Separate functions are provided for registering and describing each type. Type repository also uses ServiceTypeRepository-interface from CORBA trader. Internal functionality in provided interface is described later in this section. Operations in provided interface are described in appendix A. For example Pilarcos trader uses this interface to describe architectures it uses in population.

5.4.2 Data structures

Type repository stores three different types of data structures. These are policy framework type, service type and architecture.

Type repository uses three internal data structures. Each structure is a map for each type that can be registered. Structures are named `ArchitectureStorage`, `ServiceTypeStorage` and `PolicyFrameworkTypeStorage` respectively.

5.4.3 Registering policy framework type

Policy framework type is essentially a sequence of policy types. When a policy framework is registered type repository checks if a policy framework type is already registered under the same name. No other checks are performed for policy framework types.

5.4.4 Registering service type

When a service type is registered various checks is done for it. Firstly type repository looks for service type with the same name. Secondly type repository looks for all policy framework types used in service type. Next type repository checks that internal policy framework type names in all interfaces is connected to a policy framework type used in service type.

Finally Pilarcos service type is converted to CORBA service type with the following way. Pilarcos service type is somewhat different from CORBA service type. Pilarcos service type can have several different interfaces while CORBA service type can only have one. Pilarcos service type also has specific policy frameworks where different policy types are listed. CORBA service type only has properties which represent policies.

When Pilarcos service type is transformed to a CORBA service type. All interfaces in Pilarcos service type are coded to mandatory and read-only properties in CORBA service type. Policy frameworks in Pilarcos service type are coded also to properties. These are currently read-only properties. They are not mandatory because Pilarcos policy types can have special value which means that it has no value. This is coded in CORBA service type as missing property of that specific no value policy. Property names are generated from policy frameworks and interfaces used by Pilarcos service type.

5.4.5 Registering architecture

When architecture is registered type repository performs various checks. Firstly type repository checks that there is not a registered architecture with the same name. Secondly type repository looks for all service types that are used in current architecture. Thirdly role names are verified to be unique in the architecture. Next type repository verifies all connections from one role to other roles. Both interfaces in a connection between two roles must exist in the correct end of connection. Each role has the same interfaces which the service type used in the role has.

Type repository searches for policy framework graphs in registered architectures. This search is done in the end of registration with the following algorithm. Policy framework graph is a data structure that lists all policy frameworks in a architecture that must have compatible values in policies. type repository parses architecture with a specific algorithm developed for this task. Algorithm is discussed in detail in this subsection and illustrated in Figures 5.14 and 5.15. Small arrows represent movement between roles in the figures. Currently used interface is marked with bold and each role has its own internal names for each policy framework type.

Algorithm is based on deep-search in a net. Search advances from a role (node in net) one interface at a time as seen in Figure 5.14, section b. If role is connected via interface to another role (forms an arch) this connection is used to get to another roles as seen in Figure 5.14, sections b and d. One policy framework graph has only one kind policy framework type members. Algorithm works like this:

When algorithm reaches a role it checks if it is marked. If it is not marked it is marked as visited. This is shown in Figures 5.14 and 5.14 as strengthened borders in a role. Next frameworks related to used interface are added to same graphs where same type frameworks at the previous role are as seen in Figure 5.14, sections b and c.

If a reached role is marked visited, algorithm checks if frameworks related to current interface are already a part of a graph. If framework is already in a graph, the graph and the graph where framework in previous role are merged. Next algorithm returns to previous role. This phase is visible in Figure 5.15, section f and h.

Each role is processed one interface at a time. If one interface has more than one connection, all these connections are processed before advancing to next interface.

When all connections from start role are processed all policy framework graphs in the architecture are found. Frameworks are added to graphs with index pairs: role index and policy framework in role index. This is enough to recognise if a policy framework is a part of one policy framework graph or not.

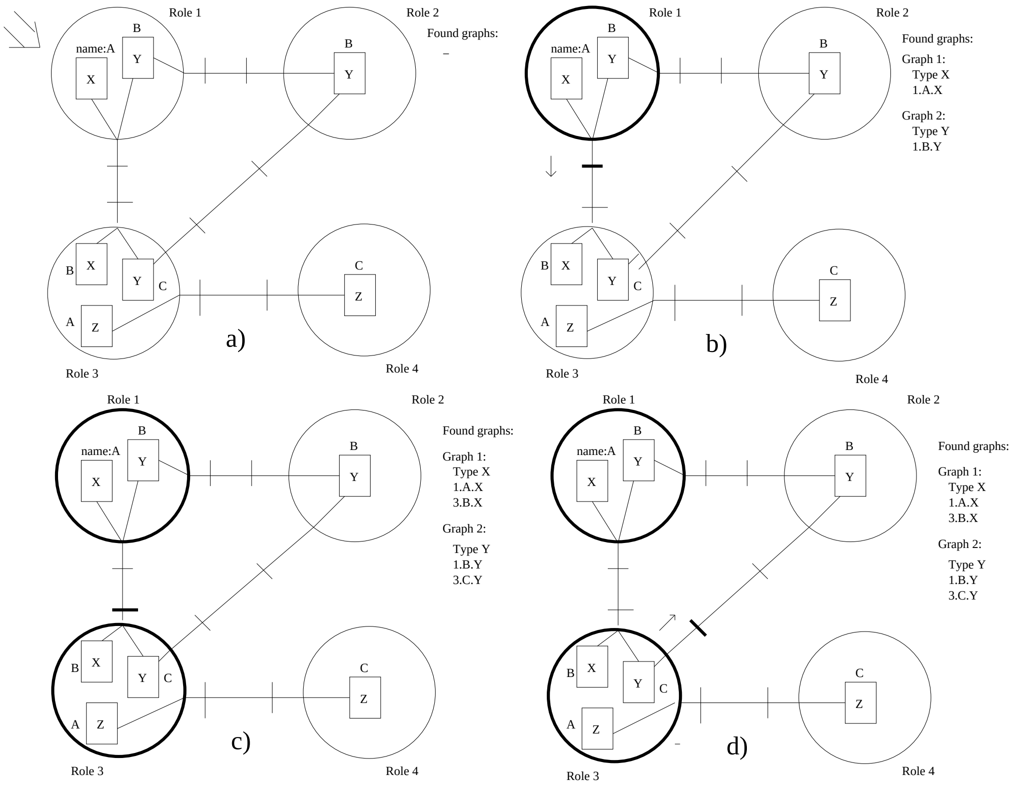

Figure 5.14: Policy framework finding algorithm in an architecture, part I

After the policy framework search type repository links the found graphs to the architecture. The linking is provided in two ways. First way is the graphs. The graphs provide linking from the graphs to the architecture. Second way is providing linking from policy framework types to the graphs.

Finally after linking type repository stores the architecture with the found and linked graphs.

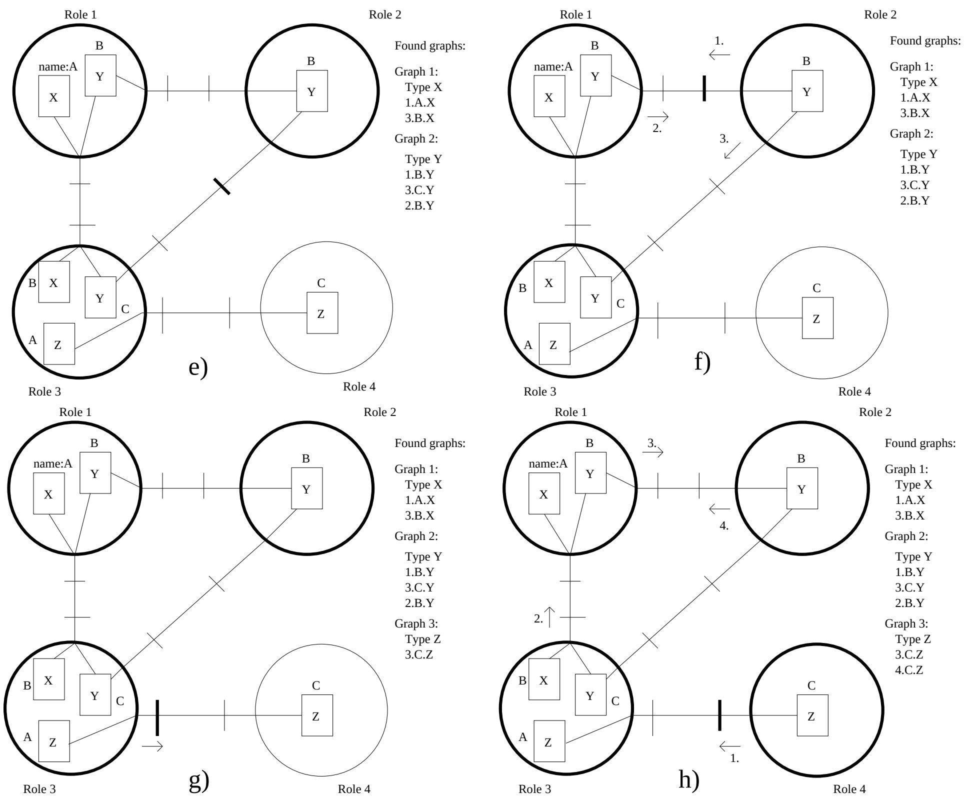

Figure 5.15: Policy framework finding algorithm in an architecture, part II

5.4.6 Other functionality in provided interface

Stored types can be described from type repository. Type repository describes types by returning the requested type. Naturally type repository cannot describe types that are not registered.

Type repository can also be reseted. Resetting type repository removes all registered types from type repository. Type repository also removes all CORBA service types from CORBA type repository which represents Pilarcos service types. Currently single types can not be removed from Pilarcos type repository.

5.4.7 Future development

Type repository could store more data than it currently does. Pilarcos infra needs some other data that can be stored in type repository. For example interface compatibility data could be saved to type repository.

Some sort of service type inheritance is also a useful feature. Currently type repository does not provide it at all. Another feature could be single type removing from type repository. Currently type repository does not support this.

5.5 Pilarcos trader

The enhanced Pilarcos trader provides two main operations: exporting a *service offer* for a specific service type (`export`), and populating a business architecture with mutually compatible service offers (`populate`). The former operation is used by an administrative tool used by a service provider. The latter operation is used by the Pilarcos federation manager on behalf of the application wishing to establish a federation. The `populate` operation takes an incomplete federation offer as a parameter, and returns one or more completed federation offers. No separate constraint parameter is used; instead, the incomplete federation offer typically contains a *pre-filled* service offer for the populating role itself, defining its policies for the federation. Thus, the population process is completely symmetrical: any role that has been left empty in the incomplete federation offer is populated by the Pilarcos trader. This makes it easy to do partial re-populations for failure recovery or adaptation purposes.

The Pilarcos trader implementation has two alternative modes of operation: stand-alone or using a standard CORBA Trading Service implementation for service offer storage. Stand-alone mode is considerably faster, since interprocess communication is avoided.

Using a CORBA Trading Service for service offer storage has a number of other advantages, however. Mature implementations are available, with different options for storing and caching service offers. Linking of CORBA traders is directly available, which enables the Pilarcos trader to scale to larger systems. By adhering to the standard, interoperability can be achieved, and it becomes possible to directly take advantage of the research efforts to improve the scalability of the CORBA trading model [3, 2].

If a CORBA Trading Service is used, on exporting the Pilarcos trader converts service offers into a form suitable for the CORBA trader and registers them there. The reverse conversion is performed during the population process. The Pilarcos trader imports service offers from the CORBA trader in blocks of five offers at a time to limit offer transfer overhead, since the Pilarcos trader and the CORBA trader reside in different processes and perhaps even on different servers.

In both operating modes, the Pilarcos trader prototype keeps the service offers in main memory. This is typical of traders, which need to have a small response time to queries. At the very least, the most frequently used service offers would be cached in main memory.

5.5.1 Design

The primary requirement for Pilarcos trader is to process population operations efficiently. In a typical use scenario the potential federation offer space is huge, possibly millions of service offer combinations. Most of the service offer combinations are unlikely to fulfil compatibility requirements, which are Pilarcos trader's form of constraint expressions. Details of the compatibility requirements are given in Section 5.5.3.

Running the population process is determined by policy values of service offers, so to achieve two goals. First, this ensures that compatibility requirements are met. Second, it minimises work on combinations which are not compatible, making the population process efficient. The algorithm implementing the population process is discussed in detail in Section 5.5.4

Pilarcos trader is also fair to both exporters and importers. When service combinations from a huge potential set are considered, the order in which the combinations are considered is randomised. If this was not done, few service offers would be overused while others would not be used hardly at all.

5.5.2 Interfaces

Pilarcos trader functionality is exposed through two interfaces. Service providers (exporters) can register and withdraw their service offers through offer registration interface. Clients (importers) looking for partners can do so with the lookup interface.

Exporters set their policy choices in the service offer they register. Importers specify their policy choices in the incomplete federation offer, and the specified policy values are also used to constraint the search. No separate constraint expressions is used. Importer may pre-fill in any desired number of roles in the incomplete federation contract. Additionally the importer sets limits to the number of federation contract offers to be returned by the Pilarcos trader and to the amount of time spent. The return value is a list of federation contract offers that meet the compatibility requirements. To achieve fairness, the trader randomised the search order for each query, and because of this the result of a query is likely to be different from call to call.

5.5.3 Compatibility requirements

The compatibility requirements discussed in this section is the way to express constraints in lookup queries in Pilarcos trader.

In the population process, the Pilarcos trader searches its database for compatible service offer combinations. Compatibility between service offers is determined by interface and policy compatibility of the offers. Interfaces are compared by querying the type repository for interface compatibility. This is done only once per binding in the architecture.

Policy compatibility is directed by bindings in the architecture. When interfaces of two roles are connected by a binding, policy frameworks of the same type attached to the interfaces are also implicitly connected. For service offers for the roles to be compatible, their connected policy frameworks must then also be compatible. The compatibility requirement can even extend over several roles via service types that have shared policy frameworks, as in the tourist service example where payment policies are shared by all roles.

In the general case, the implicit connections between policy frameworks form undirected graphs, called *policy framework graphs*, with roles as vertices and bindings as edges. Each policy framework graph represents a policy framework implicitly shared between the roles in the graph. When a business architecture is registered in the Pilarcos type repository, it finds the policy framework graphs of the architecture by a simple depth-first algorithm and stores them for later use by the Pilarcos trader.

5.5.4 Search algorithm

In the population process, the Pilarcos trader searches its database for compatible service offer combinations. Due to the combinatorial nature of the problem, the number of possible service offer combinations can be very large; in fact, in the general case the problem of whether complete federation offers exist for a given architecture and a given offer database can be proven to be NP-complete. Yet the population process should be relatively quick to be practical. This is achieved by two means: using an optimised search algorithm and restricting the extent of the search.

The populator can restrict the search by giving two limits: the maximum number of federation offers to be returned, and the maximum duration for the search process within the trader. The search terminates when either limit has been reached or the entire offer database has been searched without success. In most cases, returning one or a couple of federation offers suffices.

The search algorithm proceeds depth-first with respect to the roles of the architecture in order to find the first complete federation offers as quickly as possible. This avoids having to search the entire offer space. Each role has an associated service type; since the trader indexes service offers by their service type, the candidate offers for each role can be accessed rapidly. Pre-filled roles are handled just like other roles, except that they only have one candidate offer.

To reduce the size of the search tree, the Pilarcos trader sorts the roles into ascending order according to the number of candidate service offers. The search algorithm visits the offers in the roles recursively, beginning with the first role. The interfaces and policies of a candidate offer for the current role are compared with those of the offers selected for the previous roles; if they are compatible, the search proceeds to the next role, otherwise the next candidate offer for the role is tried until none are left. When a compatible offer is found for the last role, the selected offers form a complete federation offer.

The number of interfaces per service type is typically small, so comparing them does not pose a significant overhead. The number of policies in a service offer can be much larger; therefore it is important to minimise the number of policy comparisons. This is done by taking advantage of policy framework graph data provided by the type repository.

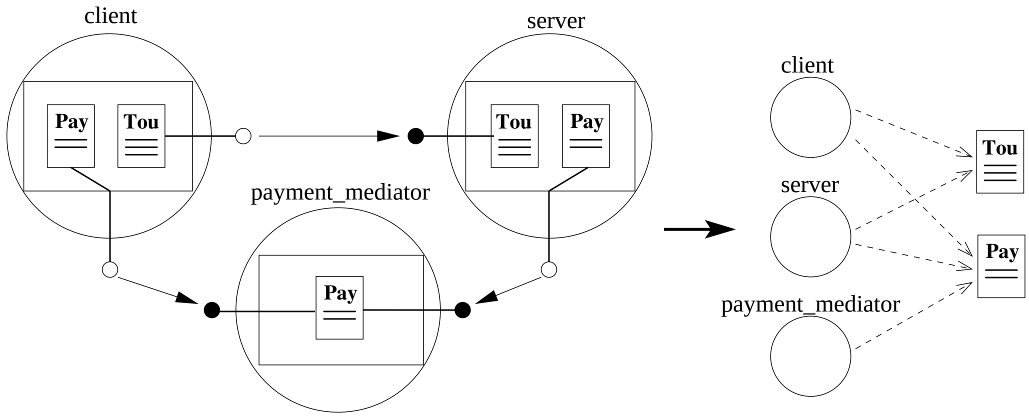

Figure 5.16: Graph-specific policy frameworks in the example architecture.

At the beginning of the search process, the Pilarcos trader requests both the architecture definition and the accompanying policy framework graph data from the type repository. It then allocates space for one *graph-specific policy framework* per each graph. For the tourist service example, graph-specific policy frameworks are illustrated in Figure 5.16; policy framework types are marked “Tou” and “Pay”. Policy comparisons are done by calculating the intersections of the policies in the candidate offer with those in the graph-specific policy frameworks. If the intersections are non-empty, the offer matches, and the intersections are stored in the graph-specific policy frameworks. Since the algorithm is recursive, each invocation has its own copy of the graph-specific frameworks, containing the intersections of the policies in the previously selected offers. This minimises the number of calculations needed.

Because the federation offers returned by the Pilarcos trader are to be used as basis for federation establishment, where only the compatible subset of interval and string set

Algorithm **Populate**(incomplete federation offer I):

1. Retrieve architecture definition and policy framework graphs of I from the type repository.
2. Collect candidate service offers for each role R_i as follows:
   (a) If there is a pre-filled offer in I for R_i, mark it as the only service offer for R_i; else
   (b) Search the service offer database for offers of R_i's service type.
3. Sort the roles $(R_1 \ldots R_n)$ into ascending order according to the number of candidate service offers.
4. Initialise empty list of complete federation offers L.
5. Invoke **SearchRole**(R_1, empty federation offer, set of empty graph-specific policy frameworks).
6. Return federation offer list L.

Algorithm **SearchRole**(role R_i, federation offer F, set of graph-specific policy frameworks P):

1. For all candidate service offers O_j of role R_i do:
   (a) Query type repository for compatibility of O_j's interfaces with adjacent offers in F.
   (b) If an interface is not compatible, return from the algorithm.
   (c) Calculate intersections of corresponding policies in P and O_j, storing the results back in P.
   (d) If any intersection is empty, return from the algorithm.
   (e) Add service offer O_j to federation offer F.
   (f) If R_i is the last role, then:
      i. Copy policy values from P to corresponding policy frameworks in the service offers of F.
      ii. Add F to federation offer list L.
   otherwise:
      i. Invoke **SearchRole**(role R_{i+1}, copy of F, copy of P).
2. Return from the algorithm.

Figure 5.17: Outline of the Pilarcos trader population algorithm.

constraint policies can be used, the Pilarcos trader replaces the original policy values with their intersections before returning the results. In the present algorithm, when a complete federation offer has been found, intersections of policies for it are already available and can be written over the originals in the federation offer. The entire algorithm is outlined in Figure 5.17.

As presented here, the search algorithm performs the least possible number of comparisons, and uses space only in linear proportion to the number of roles and the number of policy framework graphs. This yields good results in practice, as will be seen in the following section.

From a combinatorial standpoint, each newly formed federation offer should be adequately different from the earlier so that the populator does not receive almost identical offers. Furthermore, the service offer space should be covered uniformly instead of always returning the same set of offers for identical queries. To address these aspects, the Pilarcos trader arranges the service offers for each role randomly before starting the search. After a complete federation offer has been found, the offers are again ordered randomly and the search is begun anew. Possible duplicate federation offers are skipped. Since the federation offer space is typically quite large, this procedure works well to provide evenly distributed results.

Currently, all federation offers that match with the original incomplete federation offer are returned. It would be possible to add a *preference* expression, as in the CORBA Trading Service, according to which found federation offers would be sorted. Another possibility, more in line with the Pilarcos trading model, would be to add preferences to individual policies and sort the service offers according to them before starting the search.

The service offers traded include sets or intervals of alternative business policy values or technical property descriptions presented as service offer properties. These sets and intervals get narrowed down to values that are acceptable for the combination of services in the federation. However, the selection of final policy values or technical properties need to be unambiguous. For now, we let the Pilarcos trader choose those final values, without any further negotiation.

5.5.5 Data structures

The way the search algorithm implements the compatibility requirements directs the use of data structures. The data structures required to describe the architecture and policy framework graphs are provided by type repository which is described in Section 5.4.7. The rest of the data structures are sets and maps so that roles, service types, policy framework graphs and service offers, and their id's, can be efficiently cross-referenced as needed.

5.5.6 Future work

The implementation so far works efficiently and is fair. Possible future work is summarised in the following list:

- A considerable part of the time (about half) as seen by clients is spent by the CORBA implementations, which can be seen in 6. Removing the use of CORBA's Any -type might reduce the marshaling overhead.
- There are no modifiable nor dynamic policies. A proper dynamic policy system could involve sophisticated bidding system, a subject of its own.
- To provide more scalability a linking of Pilarcos traders could be implemented.
- The full CORBA trading service interface could be implemented to the stand-alone Pilarcos trader.
- It might be possible to further optimise the population process by maintaining statistics about role and policy compatibilities and or by researching compatibility heuristics in the average case.
- Trader policies could be implemented to choose alternative preferences for trader operations, for example tradeoff between efficient and fair operation, or how policy values are narrowed into single values.

Chapter 6

Performance measurements

6.1 Pilarcos trader performance

The Pilarcos middleware feasibility is much dependent on the scalability of Pilarcos trading. The measurements show that the algorithm behaves well under changes in service offer space and business architecture complexity.

6.1.1 Measurement parameters

For performance measurements, computer-generated service offer databases and business architectures were used with separately controllable parameters. The measurement parameters were:

Number of roles in business architecture. The generated business architectures consist of roles bound together in the form of a chain, without cross bindings; however, the form of the architecture makes no difference in the search algorithm. Each role in the architecture has a service type of its own.

Number of service offers. Service offers were generated separately for each role in the architecture. Their distribution was controlled by two parameters: minimum and maximum number of service offers per role. The number of offers for the roles was linearly interpolated between the two.

Number of policies. The total number of policies per service offer, distributed evenly between policy frameworks. One half of the policies were intervals and string set constraints, the other half was integers and boolean values.

Offer match ratio. This is the ratio of federation offers to all possible combinations of service offers in the offer database.

Since policy frameworks are attached to interfaces, the number of interfaces per role is not an independent parameter; adding interfaces has the same effect on performance as adding policies.

Per data point, 20 randomised service offer databases were generated. The cut-off role determining federation offer compatibility or incompatibility in the generated databases was the third role.

Parameters for the baseline case were: four roles, 40 to 100 service offers per three roles (210 total), four interfaces per role, 32 policies per service offer in four policy frameworks, and an offer match ratio of 30 %. This represents a rather heavy but, in our opinion, realistic usage scenario. In most tests, the Pilarcos trader was set to return 20 federation offers per request; however, in the results, only the time to find the first federation offer is reported, since it is largest and the differences are small. No time limit was set.

The measurement client created an incomplete federation contract with one pre-filled role, and called the Pilarcos trader `populate` operation to fill the remaining three roles. Thus, in the baseline case, the Pilarcos trader had a total of 280 000 possible service offer combinations to search, of which 84 000 were valid federation offers.

6.1.2 Measurement environment

The performance measurements were conducted on two 1 GHz Pentium III workstations with 512 MB of RAM, connected by a closed 100 Mbps Ethernet LAN. The measurement client program was run on one workstation and the Pilarcos trader on the other workstation. For the CORBA trader we used the Java-based ORBacus trader 2.0.0 [6] running on IBM Java2 1.4.0 in compiled mode on the same workstation as the Pilarcos trader.

Time spent in the Pilarcos trader search algorithm, time spent in the CORBA trader and the response time seen by the measurement client were measured separately. The Pilarcos trader and the ORBacus trader were restarted at the beginning of each measurement series, and were warmed up with three population requests before the actual measurements. For each data point, 20 runs were conducted, one per generated random database. The service offer database was emptied between each run.

6.1.3 Results and analysis

When a search limit of one federation offer was used, the population process in the baseline case took an average of 22 milliseconds. The Pilarcos trader was in standalone mode, as with other results except where noted otherwise. In addition, transferring the request and the result over CORBA took an average of 30 ms, raising the total to 52 ms. The federation offer data structure is designed for readability and flexibility, not speed: it contains several nested sequences and makes heavy use of the CORBA `Any` type, making marshalling very performance-intensive. The marshalling delay could be significantly reduced by using known CORBA IDL optimisation techniques. For the rest of the results, only the time spent in the search process is presented, since the marshalling delay is constant and predictable.

The result of varying offer match ratio from the baseline case is presented in Figure 6.1. Both the average time and the variation grow significantly with low match ratios, but are still tolerable even at a match ratio of 5 %. Based on these results, the practical usage area for the Pilarcos trader is with offer match ratios at and above 5 %.

Effect of the number of service offers was also measured. With offers distributed evenly between the roles, no significant effect on search time was seen with database sizes of up to 2550 offers. This behaviour was expected, since the population algorithm never needs to search the entire offer database. Instead, search time is directly proportional to the number of requested federation offers.

In the measurement series presented in Figure 6.2, the number of policies per service offer was varied. As expected, search time grows in linear proportion to the number of policies. In the baseline case there were 32 policies per offer, which is a rather a high estimate, over 64 policies would be exceptional. In this respect, the Pilarcos trader scales

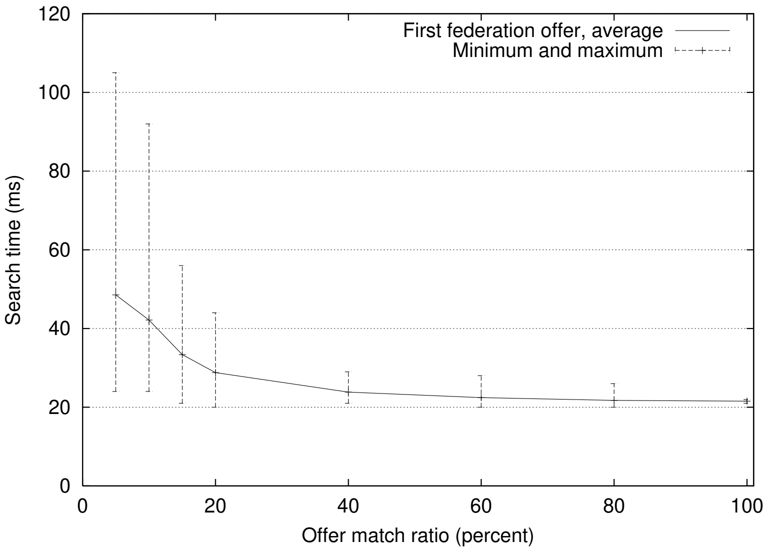

Figure 6.1: Effect of offer match ratio on search time.

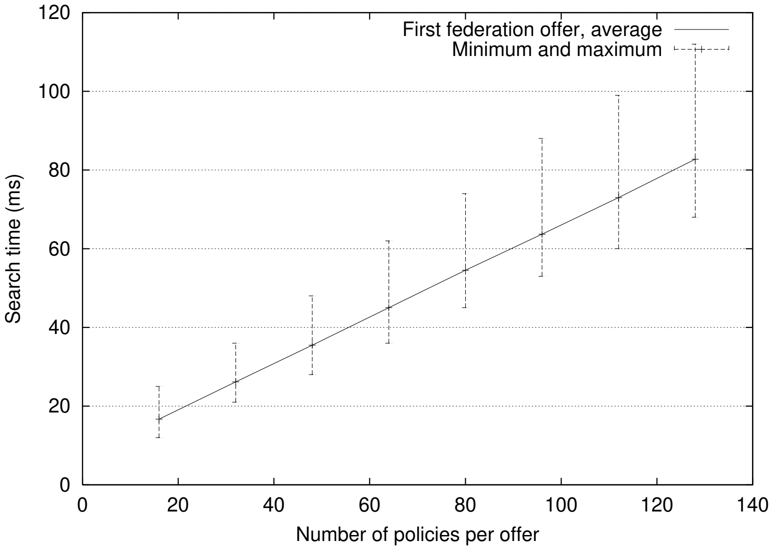

Figure 6.2: Effect of number of policies on search time.

very well. Moreover, the results could be improved significantly by reducing the amount of copying and insertion and extraction from CORBA `Any` types in the policy handling code.

Figure 6.3 illustrates the effect of varying the number of roles in the business architecture. Again, the dependency is linear, as expected. Based on these results, large architectures with up to ten roles would be practical; most probably, for such large architectures other aspects than the population process are more significant.

For the measurements in Figure 6.4, ORBacus trader was used as service offer database. The search times are nearly ten-fold compared to the standalone case (Figure 6.1), with significantly larger variation. Also, an initial cost of over 100 ms is incurred by the first

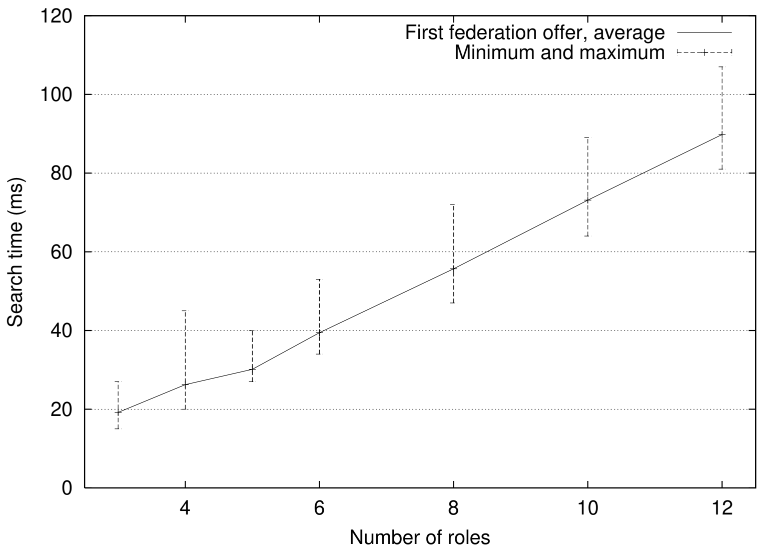

Figure 6.3: Effect of number of roles on search time.

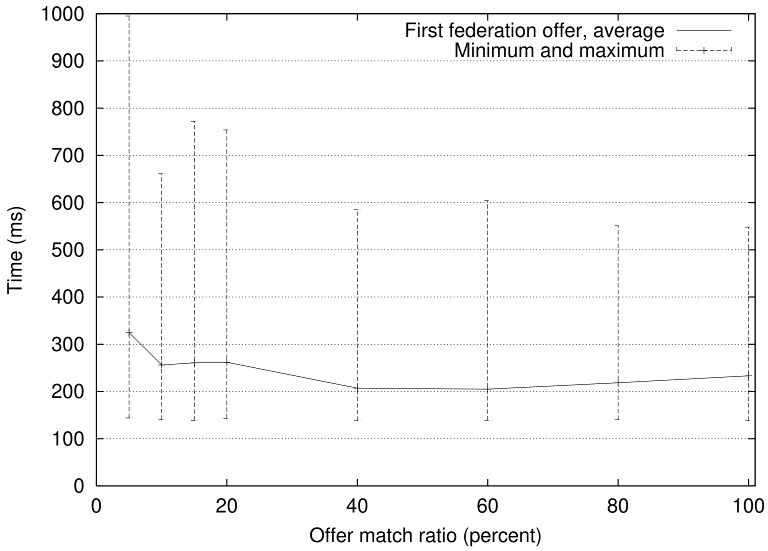

Figure 6.4: Effect of offer match ratio on search time with ORBacus trader.

ORBacus trader queries. The initial cost as well as the large variation results from the block-wise transfer of service offers from the ORBacus trader. Additional, seemingly random fluctuations in the curve are probably caused by garbage collection in the Java process. According to more detailed measurements, transferring service offers between the ORBacus trader and the Pilarcos trader takes more than half of the ORBacus trader query time. When a CORBA trader is used as a back-end, a significant significant speedup could be achieved by collocating the Pilarcos trader and the CORBA trader in the same process.

However, the fact that transferring service offers is so performance-intensive also has implications for using federated (linked) CORBA traders. In some federated cases, the service offers would be transferred across multiple links before reaching the Pilarcos trader, multiplying the performance costs. The CORBA Trading Service has not been designed for cases where the entire contents of service offers need to be transferred, and would need to be modified to support such cases practically. As with federation offers, this could be done with known CORBA IDL optimisation techniques, for example by replacing CORBA `Any` types by a more restricted set of possible property types.

6.2 Prototype performance

Pilarcos prototype provides functionality for creating federations dynamically (this is not possible in traditional implementations). As might be expected, provided functionality produces some overhead to system. The primary goal of the committed measurements was to estimate the cost of this added functionality.

6.2.1 Measurement parameters

to estimate the cost of flexibility provided by Pilarcos compared to traditional CORBA implementation of similar demonstration case. Another goal was to provide time slicing in different phases of Pilarcos both from clients point of view and in servers. Pilarcos prototype was measured with the following parameters

- time usage at different phases of Pilarcos,
- number of clients,
- Pilarcos vs. traditional implementation without simulated load, and
- Pilarcos vs. traditional implementation with simulated load.

Time usage was measured to get exact costs of different phases in our implementation and to determine how much time Pilarcos infrastructure uses compared to actual applications in our scenario. These phases include federation establishment, federation termination, bindings between clients and servers, and application calls.

Measurements with different numbers of clients were made to simulate the behaviour of the prototype in more real world case. Unfortunately, because of technical problems already mentioned, these measurements we were not able to do as we would have wanted.

Benchmarking against traditional implementation with and without load were made to estimate the cost of Pilarcos infrastructure. Measuring with load and without load were done to see if differences between Pilarcos and traditional implementation are constant or relative.

6.2.2 Measurement environment

Measurement environment consists of seven 1GHz Pentium III workstations with 512MB RAM memory. Workstations were connected with closed 100 Mbps Ethernet. Operating system used in workstations was CS Linux with kernel 2.4.18. Different components of Pilarcos prototype were distributed to workstations, one per each machine. Background clients used two machines and were evenly distributed in them. Distribution of components are shown in Figure 6.5.

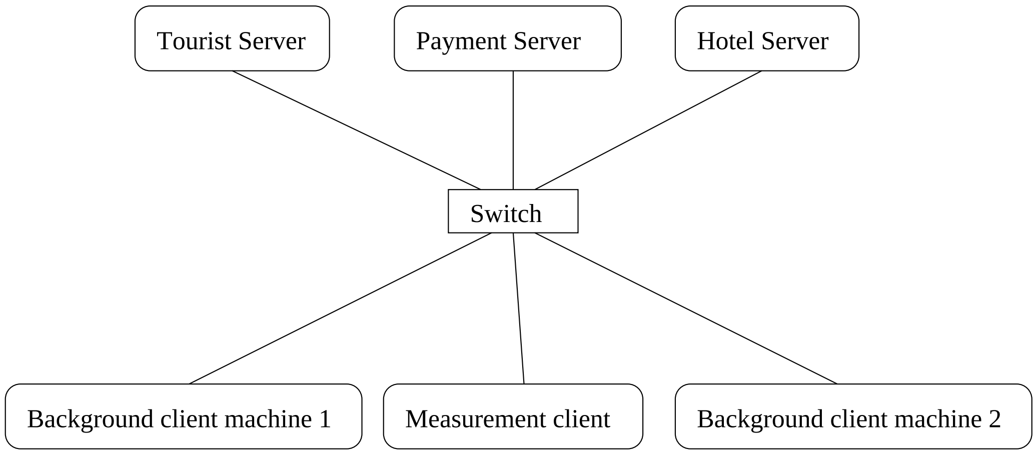

Figure 6.5: Distribution of Pilarcos components in measurement environment.

Before measurements the system was warmed up with 3000 measurement rounds done by client. Each measurement consisted off 8000 measurement rounds. Throughput optimization flags were used for Java virtual machines. In all measurements IBM Java 1.4.1-platform was used.

6.2.3 Results and analysis

Increasing number of clients and therefore number of federations per second response times and processor usage were measured. Processor usage increases linearly when more federations are executed in second, as can be seen in Figure 6.6. From the figure it can clearly be seen that HotelServer becomes the bottleneck in the prototype.

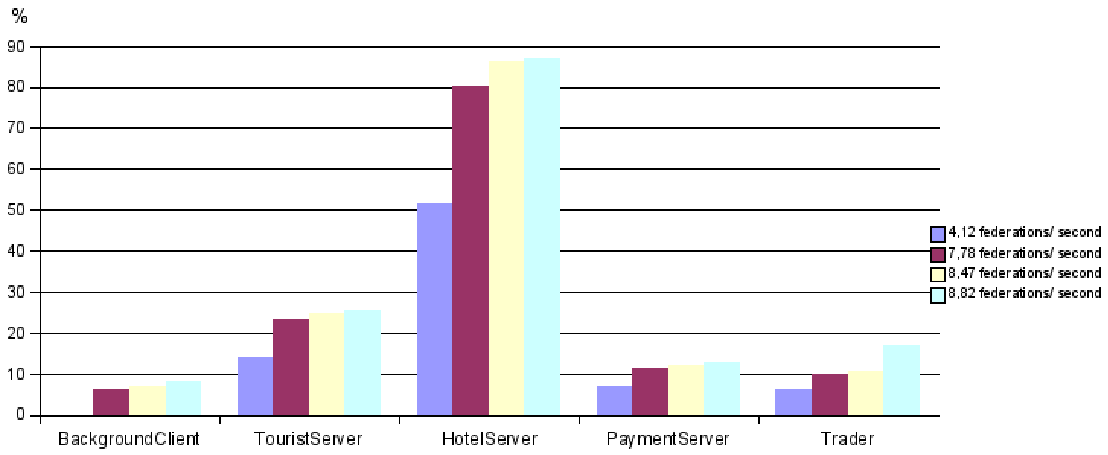

Figure 6.6: Processor usage in servers.

There are several reasons for this. Firstly HotelServer is the only domain where two platforms are used simultaneously (OpenCCM and JBoss). This generates a large number of context switches between processes in that domain (machine) which slows down processing and takes time. Secondly, all adaptation in the prototype is done in HotelServer domain. As the adaptation is currently implemented in component-level adapters there is a high amount of additional marshalling in the operation calls. As can be seen from Figure 6.6 HotelServer-domain is operationg close to its saturation point and is clearly holding back other processing. Other servers are behaving predictably and could be assumed to continue behaving similarly if federations per second value was increased.

The increase in response times seen by the client is clearly visible in Figure 6.7 when more federations are executed in second. Response times follow the increase of processor usage. Two of the highest values in this figure belong to createFederation and leaveFederation operations. Both of these operations go through the first federation and affect also the second federation. In these operations effectively two federations are created and terminated in one operation visible to client.

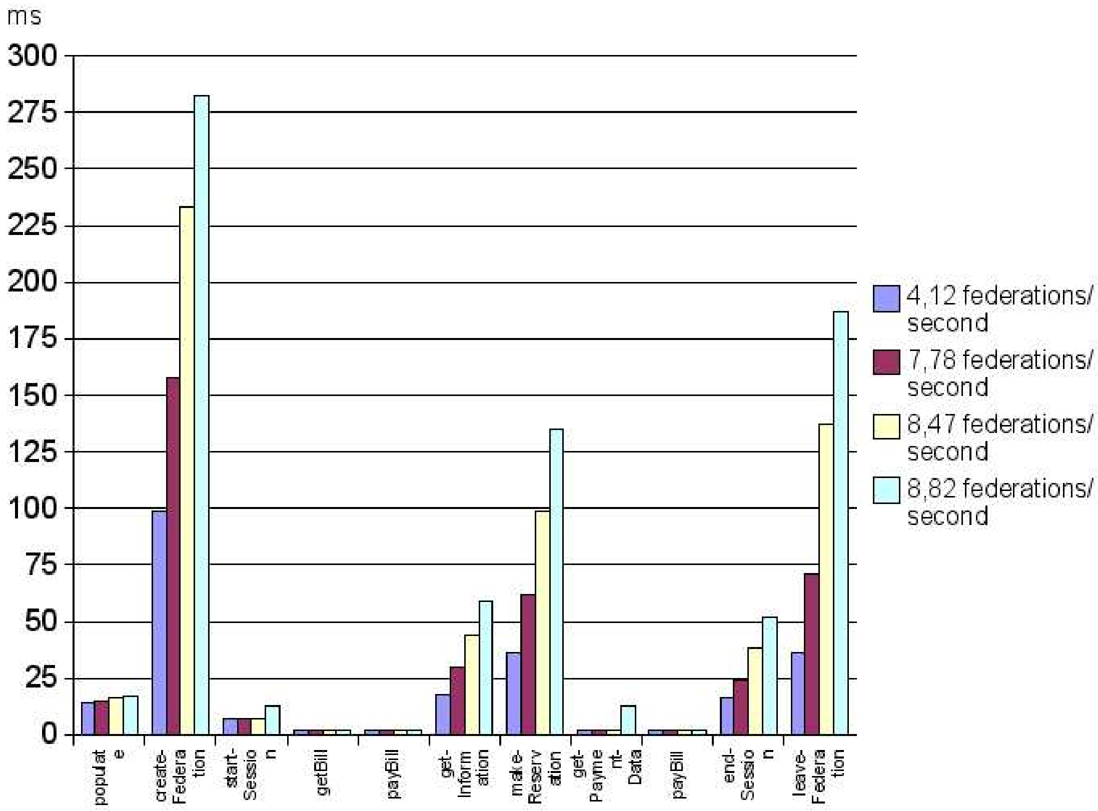

Figure 6.7: Client response times.

Further measuring should be done when the two platforms in HotelServer domain would be running on two different computers. This should decrease the processor usage and allow higher federations per second values.

Pilarcos prototype was also benchmarked against traditional implementation. In traditional implementation the services to be used are determined beforehand and resolved from nameservice during startup. This way the selection of partners becomes static and the flexibility of dynamically determining partners is lost. In addition, only one platform is used so that no adaptation is reguired.

For benchmarking simulated load was added to some operations. Simulated load does not represent any actual relation to how much time the operations would take in real world. The load was generated by calculating simple operations in a loop to prevent optimization.

Numbers in Figure 6.8 show that traditional case has only a limited advantage over Pilarcos system. Figure 6.8 also shows that federation management operations use somewhat little time compared to application calls. Input rate in these figures is the same as the lowest in Figure 6.6.

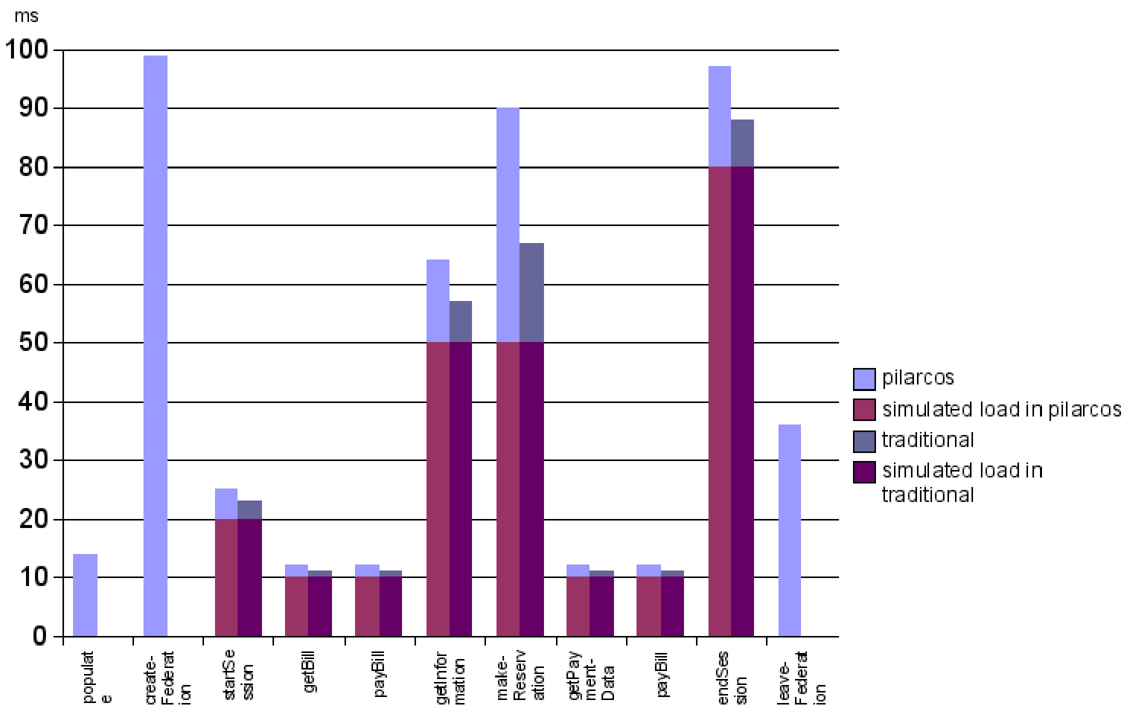

Figure 6.8: Time used in different phases seen by client with simulated load.

Creating federation and terminating it are the biggest operations when there is no simulated load as seen in Figure 6.9. It is important to remember that when client creates a federation actually two federations is created as the HotelClient creates instantly its own federation to get subservices. As Figure 6.10 shows biggest single operation is creating the HotelServer. Also in federation termination two federations is terminated because HotelClient terminates the subservice federation. Like in federation creation, terminating the HotelServer is the largest single operation in federation termination.

Detailed information of costs of adaptation and other operations is shown in Tables 6.1, 6.3 and 6.2. It appears that something should be done to adaptation model as it seems to have an effect to the performance. As can be seen in Table 6.3 adapted call from HotelServer to PaymentServer takes two milliseconds and regular call from TouristClient to PaymentServer takes one millisecond.

Some problems occurred while benchmarking the prototype. One of these was that only seven clients were able to run simultaneuosly when operating in separate machines (a measurement client on one host, three background clients per machine on two hosts). This problem was traced to OpenCCMs stubs. The problem prevented us from conducting

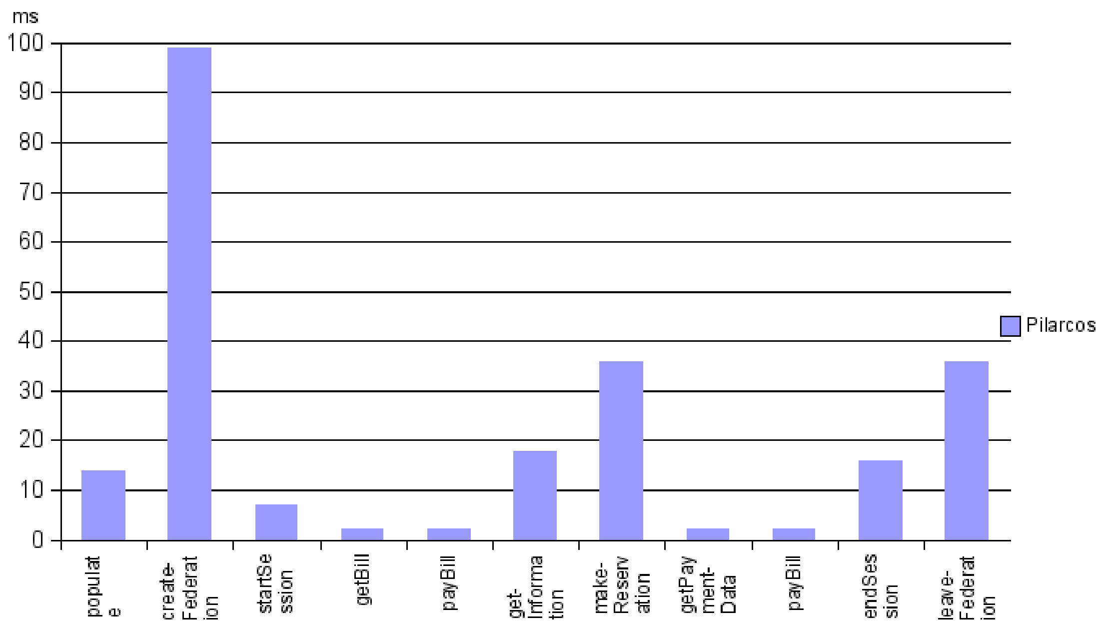

Figure 6.9: Response times seen by client.

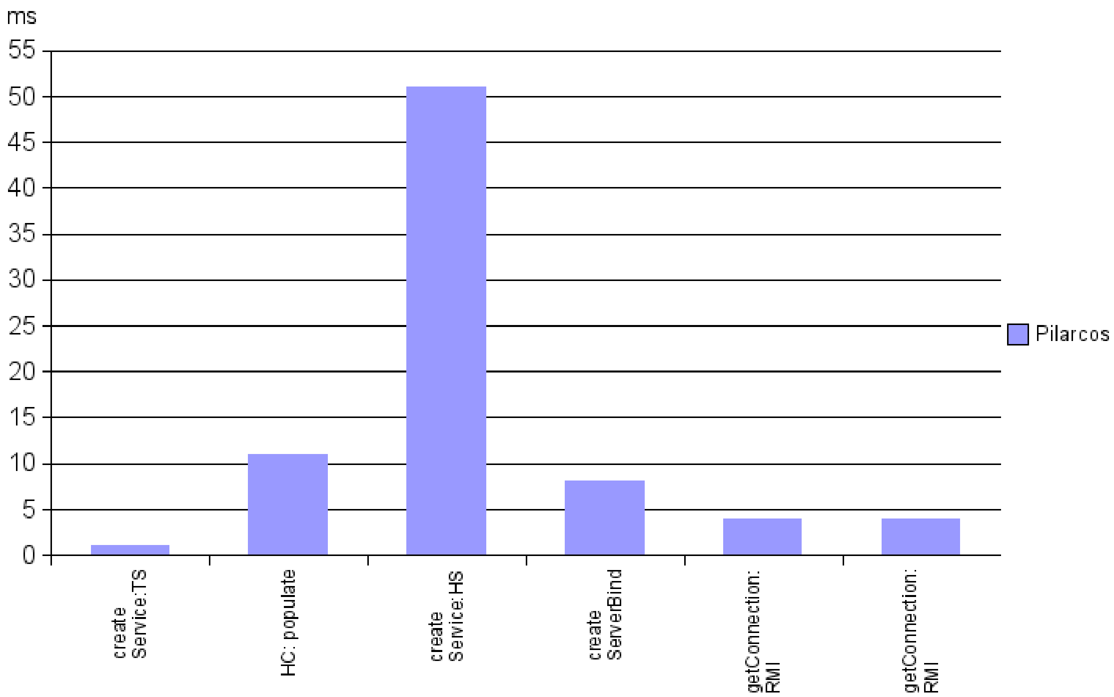

Figure 6.10: Time used in createFederation.

better scalability tests for the prototype. Unfortunately we were not able to fix these problems due to tight schedule. The other problem was mysterious peaks in response

Domain	Component	Operation	Used time (ms)
TC	TouristClient	populate	15
TC	TouristClient	createFederation	93
TC	FederationManager	createClient	0
TS	FederationManager	createService	0
TS	FederationManager	createServerBinding	0
TS	TouristServer	establishFederation	87
TS	HotelClient	populate	11
TS	HotelClient	createFederation	76
TS	FederationManager	createClient	0
HS	FederationManager	createService	51
HS	FederationManager	createServerBinding	8
HS	HotelInfoAdapter	getConnection:RMI	4
HS	HotelInfoAdapter	getConnection:RMI	4
PS	FederationManager	createService	1
PS	FederationManager	createServerBinding	0

Table 6.1: Time used in different operations during createFederation

Domain	Component	Operation	Used time (ms)
TC	TouristClient	leaveFederation	33
TS	TouristServer	terminateFederation	28
TS	HotelClient	leaveFederation	27
HS	FederationManager	terminateFederation	22
HS	FederationManager	removeService & Bindings	16
PS	FederationManager	terminateFederation	2
PS	FederationManager	removeService & Bindings	1
TS	FederationManager	removeService & Bindings	0
TC	FederationManager	removeService & Bindings	0

Table 6.2: Time used in different operations during leaveFederation

times during benchmarking. These peaks were found in both our prototype and in simple ping-program and on both Java platforms we used (IBM and Sun). These results indicate that the problem is in platforms used.

Domain	Component	Operation	Used time (ms)
TC	TouristClient	getConnection TouristInfo	2
TC	TouristClient	getConnection Payment	2
TC	TouristClient	startSession	7
TS	TouristServer	getConnection Billing	5
TS	TouristServer	getBillID	2
TC	TouristClient	getBill	2
TC	TouristClient	payBill	1
TC	TouristClient	getInformation	18
TS	HotelClient	getInformation	6
TS	TouristServer	isBillPaid	2
TS	HotelClient	getConnection HotelInfo	2
TS	HotelClient	startSession	6
TC	TouristClient	makeReservation	33
TS	HotelClient	makeReservation	26
HS	HotelRegistrationApp	makeReservation	19
HS	HotelRegApp	getConnection:RMI_Billing	16
HS	HotelRegApp	getBillID	1
TS	HotelClient	getPaymentData	4
TS	HotelClient	getConnection:Payment	2
TS	HotelClient	getBill	2
TC	TouristServer	getBillID	2
TC	TouristClient	getPaymentData	2
TC	TouristClient	getBill	2
TC	TouristClient	payBill	0
TC	TouristClient	endSession	15
TS	TouristService	isBillPaid	2
TS	TouristService	invalidateBill	2
TS	HotelClient	payBill	2
TS	HotelClient	endSession	7
HS	HotelInfoApplication	isBillPaid	2
HS	HotelInfoApp	invalidateBill	1

Table 6.3: Time used in different operations during application calls

Chapter 7

Conclusion

This document gives an implementation oriented view of the Pilarcos middleware services, together with the performance measurements collected.

The major elements of the architecture are

- Pilarcos trader that is responsible of populating business architectures with compatible service offers,
- federation managers that run federation establishment and termination protocols, and
- federated binding mechanism for coordinating the creation and configuration of adapters and exchange of addresses in a heterogeneous environment.

The middleware services were implemented on the MicoCCM and OpenCCM CORBA Component Model (CCM) platforms. The application components were implemented on OpenCCM and the JBoss EJB application server. During the prototyping efforts it was discovered that the CCM offers a powerful and flexible component programming model, but the CCM platforms are not mature yet. In contrast, the the JBoss platform was discovered to be of very high standard, and can be recommended for production use [4].

It is clear that without Pilarcos middleware the application components could not establish federations in such a flexible way. Inter-organisational interoperation can only be achieved through standard solutions and in this case, via standardising new middleware services and meta-information elements. Currently, most service-oriented architectures built with Web services middleware, in Java environments or with CORBA support (plain or with components) concentrate on service discovery in client-server type of situations. Most middleware solutions have facilities for composing services into larger elements to be provided for inter-organisational use. Services are considered as a set of independent resources in the global network, as building blocks, from which an application programmer is responsible to construct a new added-value service. Most middleware solutions solve the interoperability problems by either trusting a shared language context (Java Virtual Machine) and transfer of code, or trusting a shared protocol environment (CORBA, Web services). In Pilarcos, interoperability tests use explicit information on the shared platform facilities, thus making it possible to use different technologies side-by-side as long individual federations find common communication facilities.

The Pilarcos model provides on multilateral contracts. A business architecture can be considered as an external work-flow template that describes potential choreographies between independent parties. Each potential party provides a local work-flow that fulfils its published service contract. The local work-flows are hidden and no specific requirements for their running environment is specified. The Pilarcos middleware acts as a matchmaker for the parties, and then specialises the shared external work-flow according to the membership, corresponding business policies and technical facilities. The design deliberately differs from the common goal for distributed work-flow systems by not granting access and visibility to partner's information processing system.

The measurement results from the Pilarcos prototype show that the population process and federation establishment phase can be made scalable and the basic cost tolerable for the intended use. Pilarcos middleware is designed with dynamically changing cooperation schemes in mind. Such schemes appear for instance as virtual enterprises that provide a new service constructed by minor services run at member organisations computing facilities. Although there is a noticeable cost at establishing the federation, the duration of the federation is not only for single interactions but for a lengthier user session or more naturally, for the lifetime of the virtual enterprise.

The Pilarcos middleware is not restricted to any specific application area. The Pilarcos middleware is designed with service-oriented architectures in mind, capturing discovery of services and ensuring interoperability between them [12]. Although Pilarcos project has used component based platforms (OpenCCM, MicoCCM, JBoss) for experimenting and prototyping, the Pilarcos architecture does not require component techniques to be used.

However, there are two types of environment into which Pilarcos model is not suited. The open nature of Pilarcos model is such that highly secure systems cannot be supported: trust building mechanisms can be incorporated, but still, critical systems should use more static and closed solutions. The nature of Pilarcos middleware overhead cost is such that it is not suitable for example for hard real-time systems or systems where response times are not allowed to vary a lot.

Especially beneficial the Pilarcos model can be for SMEs (Small and Medium Size Enterprises). The SMEs gain and keep their competitive position best by networking with companies that provide complementary services. Current integration solutions are founded on a single authority directing business models and their technical development, thus benefiting only very large companies.

For future, there are several interesting development areas for the Pilarcos middleware. First, for the Pilarcos services to be more usable, the current prototype should be transformed to be accessible by protocols in the Web Services family. Second, for the Pilarcos trader to really answer the needs of globalisation and the number of service offers involved, two different federation schemes can be tested; one where federation takes place at the Pilarcos trader level, and one where federation takes place at the underlying OMG trader level. Third, future enhancements on the business architectures increase flexibility and adaptability of federations. We intend to expand business architectures with epochs and cardinalities of roles. An epoch is an interval during which services provided by a federation stay identical and the set of roles involved is unchanged. A change in services or roles starts a new epoch. In addition, extensions are needed so that federations can be made to overlap or form hierarchies without application components to be involved in the administration.

Bibliography

[1] BALEK, D. Connectors in Software Architectures. PhD Thesis, Charles University, Czech Republic, 2002.

[2] BELAID, D., PROVENZANO, N., AND TACONET, C. Dynamic management of CORBA trader federation. In *4th USENIX Conference of Object-Oriented Technologies and Systems (COOTS)* (Santa Fe, New Mexico, 1998). Also `http://www.usenix.org/publications/library/proceedings/coots98/full_pap%ers/belaid/belaid.pdf`.

[3] CRASKE, G., TARI, Z., AND KUMAR, K. R. DOK-trader: A CORBA persistent trader with query routing facilities. In *International Symposium on Distributed Objects and Applications* (1999), pp. 230–240. Also `http://rmit.edu.au/~zahirt/Papers/doa99.pdf`.

[4] HAATAJA, J.-P., METSO, J., SUORANTA, T., AND VÄHÄAHO, M. Platform experiences. Tech. rep., Jan. 2003. C-2003-NN.

[5] HERZUM, P., AND SIMMS, O. *Business Component Factory*. Wiley Computer Publishing, 1999.

[6] IONA TECHNOLOGIES. *ORBacus Trader, version 2.0.0*, 2001. `http://www.iona.com/products/orbacus/trader.htm`.

[7] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Type Repository Function.* IS14746.

[8] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model for Open Distributed Processing.* IS10746 1-4.

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

[10] KUTVONEN, L. Management of Application Federations. In *International IFIP Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)* (Cottbus, Germany, Sept. 1997), H. Konig, K. Geihs, and T. Preuss, Eds., Chapmann & Hall, pp. 33 – 46.

[11] KUTVONEN, L. *Trading services in open distributed environments.* PhD thesis, Department of Computer Science, University of Helsinki, 1998.

[12] KUTVONEN, L. Automated management of interorganisational applciations. In *EDOC2002* (2002).

[13] LEYMANN, F. Web services flow language (wsfl 1.0). Tech. rep., 2001. `http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[14] LUPU, E., AND SLOMAN, M. A policy based role object model. In *Proceedings of the 1st International Enterprise Distributed Object Computing Conference (EDOC'97), Gold Coast, Queensland, Australia* (October 1997), pp. 36–47.

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

[16] OBJECT MANAGEMENT GROUP. *CORBA Component Model - Volume 1.* Framingham, MA, USA, 1999. OMG Document orbos/99-07-01.

[17] OBJECT MANAGEMENT GROUP. *CORBA Component Model v3.0.* Framingham, MA, USA, 2002. OMG Document formal/02-06-65.

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

[19] SIEGEL, J. *Developing in OMG's Model-Driven Architecture.* Object Management Group, Nov. 2001. White paper, revision 2.6.

[20] SUN MICROSYSTEMS, INC. *Enterprise JavaBeans Specification v1.1*, 1999.

[21] SUN MICROSYSTEMS, INC. *Enterprise JavaBeans Specification v2.0*, 2001.

[22] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 1998.

[23] THATTE, S. Xlang. Tech. rep., 2001. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.html`.

[24] THATTE, S. Business process execution language for web services, version 1.0. Tech. rep., July 2002. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`.

[25] VALLECILLO, A., HERNÁNDEZ, J., AND TROYA, J. M. Component interoperability. Tech. rep., Dept. Lenguajes y Ciencias de la Computación, University of Málaga, July 2000. ITI-2000-37.

Appendix A

Selected IDL definitions

This appendix contains the OMG IDL definitions of the central interfaces and data structures in the prototype. Some of the less significant interfaces and data structures as well as raised exceptions have been left out for clarity.

A.1 Common

```
module Pilarcos {
    //
    // ---------- Policies ----------
    //

    typedef string                          PolicyTypeName;
    typedef string                          PolicyFrameworkTypeName;
    typedef string                          PolicyFrameworkName;
    typedef sequence<PolicyFrameworkName>   PolicyFrameworkNameSeq;

    typedef any                             PolicyValue;
    typedef sequence<PolicyValue>           PolicyValues;
    typedef sequence<PolicyValues>          PolicyValuesSeq;


    // omitted policy value
    struct NoValue {
        long no_value;
    };

    // TODO: this should be made internal to Trader
    typedef sequence<double>              PreferredValues;

    struct LongRange {
        long            low;
        long            high;

        // TODO: should be
        // long         preferred;
        PreferredValues preferred_values;
    };

    struct DoubleRange {
        double          low;
        double          high;
```

```
    // TODO: should be
    // double       preferred;
    PreferredValues preferred_values;
};

enum ConstraintType {
    some_of,
    one_of,
    exactly
};

typedef sequence<string>             StringList;
typedef sequence<double>             WeightFactors;

struct StringSet {
    StringList                       strings;
    WeightFactors                    weight_factors;
    ConstraintType                   constraint_type;
};

//
// ---------- Interfaces ---------
//
typedef string                       FederationContractID;

typedef string                       NameServiceRef;
typedef string                       PSIType;
typedef string                       PIIName;
typedef string                       FederatedName;
typedef string                       NamingFormat; // CORBA, JNDI, ...

typedef sequence<PSIType>            PSITypeSeq;
typedef sequence<PIIName>            PIINameSeq;

struct IndirectRef {
    string         name;
    NameServiceRef ns_loc;
};

interface FederatedNaming;
struct InterfaceRef {
    PSIType                          type;
    // name of the interface in the service type (e.g. "billing")
    PIIName                          pii_name;

    FederatedNaming                  federated_ns;
    FederatedName                    federated_name;
    boolean                          direct_naming;
};

typedef sequence<InterfaceRef>       InterfaceRefSeq;

interface FederatedNaming {
    void bind_direct(in FederationContractID     id,
                     in FederatedName            name,
                     in Object                   obj);
    void bind       (in FederationContractID     id,
                     in FederatedName            name,
```

```
                    in IndirectRef              ref);
    void unbind     (in FederationContractID    id,
                    in FederatedName            name);
    Object      resolve_direct(in FederationContractID  id, in FederatedName name);
    IndirectRef resolve       (in FederationContractID  id, in FederatedName name);
};

//
// ---------- Service contract structure ----------
//

typedef string                      ServiceTypeName;
typedef string                      RoleName;
typedef string                      ArchitectureName;
typedef string                      ServiceContractID;

typedef sequence<ServiceTypeName>   ServiceTypeNameSeq;

struct ServiceContract {
    ServiceContractID               service_contract_id;
    ServiceTypeName                 service_type;
    PolicyValuesSeq                 policy_values;
    InterfaceRefSeq                 interfaces;
    Object                             federation_manager;
};

//
// ---------- Federation contract structure ----------
//

struct ServiceForRole {
    RoleName                        role;
    ServiceContract                 service_contract;
};
typedef sequence<ServiceForRole>    ServiceForRoleSeq;

struct FederationContract {
    FederationContractID            federation_contract_id;
    ArchitectureName                architecture;
    ServiceForRoleSeq               service_contracts;
};
typedef sequence<FederationContract> FederationContractSeq;


//
// ---------- Policy context structure ----------
//

typedef long                        PolicyContextID;
typedef sequence<PolicyContextID>   PolicyContextIDSeq;

// Currently this is equal to ServiceTypeName.
// (Really, ServiceType should include a PolicyContextType,
// which would define the policy frameworks of the service type.)

typedef string                      PolicyContextTypeName;

struct PolicyContext {
    PolicyContextID                 id;
```

```
        // for now, equal to ServiceTypeName; empty for application contexts
        PolicyContextTypeName           type;

        // similar to ServiceContract.policy_values;
        // zero-length for application contexts
        PolicyValuesSeq                 policy_values;

        // for offer contexts and federation contexts; empty for others
        ServiceContractID               service_contract_id;

        // only for federation contexts; empty for others
        FederationContractID            federation_contract_id;
    };

    typedef sequence<PolicyContext>     PolicyContextSeq;
};
```

A.2 Federation manager, service factory, and federated application

```
module Pilarcos {
    module FederatedApplication {

        component FederationManagerComponent;

        // ==================================================
        // Generic Configuration Interface
        // ==================================================

        typedef sequence<string> ConfigurationData;
        interface Configuration {
            // At the moment contains the arguments to ORB
            void configure(in ConfigurationData configuration_data);
        };

        interface AdapterConfiguration {
            void configure(in ConfigurationData configuration_data,
                           in Pilarcos::InterfaceRef if_ref);
        };

        // ==================================================
        // Application Component
        // ==================================================

        // Called by federation manager to notify application.
        interface FederatedApplicationInterface {
            void establishFederation(in PolicyContextID federation_context_id)
            void terminateFederation(in PolicyContextID federation_context_id);
        };

        // ==================================================
        // Service Factory
        // ==================================================

        typedef string          AssemblyID;
        typedef string          ComponentType;
```

```
typedef string           InterfaceName;
typedef sequence<string> ComponentSourceSeq;

struct ServiceTypeInformation {
    ServiceTypeName service_type;    // Implemented service type

    // "Concrete" interface types implemented by the assembly
    PSITypeSeq      interface_types;

    // names of the interface types in the service type
    PIINameSeq      pii_names;
};

typedef sequence<ServiceTypeInformation> ServiceTypeInformationSeq;

// 'federation', 'conservative', 'client'
typedef string LifeCyclePolicy;

struct Assembly {
    // ID of the Assembly
    AssemblyID                      assembly_id;
    // Assembly life cycle policy
    LifeCyclePolicy                 life_cycle;
    // Information related to service types
    ServiceTypeInformationSeq       service_type_information;
};

struct ServiceData {
    AssemblyID                   assembly_id;
    PolicyContextID              application_context_id;
    FederatedApplicationInterface federated_application_interface;
};
typedef sequence<ServiceData> ServiceDataSeq;

// Used to identify a specific instantiation (not necessarily instance)
typedef string ServiceCookie;
typedef sequence<ServiceCookie> ServiceCookieSeq;

struct ServiceInstanceData {
    ServiceData               service_data;
    ServiceCookie             cookie;
    Pilarcos::InterfaceRefSeq int_refs;
};
typedef sequence<ServiceInstanceData> ServiceInstanceDataSeq;

interface ServiceManagement {
    void addFederationAssociation(in ServiceCookie cookie,
                                  in FederationContractID contract_id,
                                  in PolicyContextID context_id);

    ServiceInstanceData createClient(in ServiceContract service_contract);
    ServiceInstanceData createService(in ServiceContract service_contract,
                                      in FederationContractID federation_contract_id);
    ServiceInstanceData createServiceInstanceData
                                     (in PolicyContextID application_context_id);
    void                destroyServiceInstanceData(in ServiceCookie cookie);
    ServiceInstanceData getServiceInstanceData(in ServiceCookie cookie);
    void                destroyService(in ServiceCookie cookie);
};
```

```
interface ChannelManagement {
    Pilarcos::InterfaceRef createServerBinding
                                ( in Pilarcos::InterfaceRef ref_to_server,
                                  in Pilarcos::InterfaceRef ref_from_contract,
                                  in ServiceCookie          cookie);
    Pilarcos::InterfaceRef createClientBinding
                                ( in Pilarcos::PSIType      type_of_client,
                                  in Pilarcos::InterfaceRef ref_from_contract,
                                  in ServiceCookie          cookie);
    /** Generic binding operation -> not yet implemented
    Pilarcos::InterfaceRef createBinding(in Pilarcos::PSIType type_from_assembly,
                                         in Pilarcos::InterfaceRef ref_from_contract,
                                         in boolean server_side);
    **/
    void destroyBindings(in ServiceCookie service_cookie);
};

interface FactoryConfiguration : Configuration{
    void addAssembly (in Assembly assembly);
    void removeAssembly (in AssemblyID assembly_id);
};

// ==================================================
// Federation Manager
// ==================================================

interface ChannelInterface {
    // Operation returns the reference to the adapter
    Pilarcos::InterfaceRef getInterfaceRef(in long federation_context_id,
                                           in string interface_name,
                                           in Pilarcos::PSIType interface_type);
};

interface PolicyInterface {
    // Adds a new policy context as the child of the given
    // parent. Updates the parent accordingly.
    PolicyContextID addPolicyContext(
        in PolicyContextID parent_context_id,
        in PolicyContext new_context
    );

    // For updating values of a policy context.
    void updatePolicyContext(
        in PolicyContext policy_context
    );

    // Retrieves a policy context.
    PolicyContext retrievePolicyContext(
        in PolicyContextID context_id
    );

    // Deletes a policy context.
    // (Also deletes all children and updates parent.)
    void deletePolicyContext(
        in PolicyContextID context_id
    );

    boolean isPolicyContextStored(
```

```
        in PolicyContextID context_id
    );

    PolicyContextID getParentContext(
        in PolicyContextID context_id
    );

    PolicyContextIDSeq getChildContexts(
        in PolicyContextID context_id
    );

    // Creates an offer context under a service context,
    // duplicating the policy values of the service context.
    PolicyContextID createOfferContext(
        in PolicyContextId application_context_id,
        in ServiceTypeName service_type
    );

    // Additional operations related to contract-context mapping.
    PolicyContextID getFederationContext(
        in FederationContractID federation_contract_id
    );

    FederationContract getFederationContract(
        in FederationContractID federation_contract_id
    );
};

interface FederationInterface {
    // Returns offers as federation contexts.
    PolicyContextIDSeq populateArchitecture(
        in PolicyContextID  offer_context_id,
        in ArchitectureName architecture,
        in RoleName         my_role,
        in long             max_offers,
        in long             max_time_ms
        );

    // Deletes extraneous offers (federation contexts), freeing resources.
    oneway void discardOffers( in PolicyContextIDSeq federation_context_ids );

    // Returns federation contract ID, because for now it is required
    // to make federated operation calls
    FederationContractID createFederation( in PolicyContextID federation_context_id );

    oneway void leaveFederation ( in PolicyContextID federation_context_id,
                                  in boolean remove_offer_context );
};

interface FederationManagement {
    void establishFederation( in FederationContract   contract,
                              in long                 role_index );

    void terminateFederation( in FederationContractID contract_id );
};

component AdapterComponent {
    provides AdapterConfiguration configuration;
};
```

```
component ApplicationComponent {
    // API (upcall and downcall)
    uses PolicyInterface      policy_interface;
    uses FederationInterface  federation_interface;
    uses ChannelInterface     channel_interface;

    provides FederatedApplicationInterface fed_app_interface;

    // root context for application
    attribute PolicyContextID application_context_id;
};

component FederationManagerComponent {
    // Naming API
    provides Pilarcos::FederatedNaming        federated_naming;

    // API for application
    provides PolicyInterface                  policy_interface;
    provides FederationInterface              federation_interface;

    provides ChannelInterface                 channel_interface;

    // Service Factory interface
    uses     ServiceManagement      service_management;
    uses     ChannelManagement      channel_management;

    // provided for other federation managers (including the federation coordinator)
    provides FederationManagement   federation_management;

    // access to Policy Repository
    uses PolicyRepository::PolicyContextManagement policy_context_management;
    uses Trading         ::Lookup                  lookup;

    // access to Type Repository
    uses TypeRepository::TypeRegistration type_registration;

    // my own FederationManagement facet (attribute kludge)
    attribute FederationManagement my_federation_management;
    attribute FederatedNaming my_federated_naming;
};
home FederationManagerHome manages FederationManagerComponent {
};

component ServiceFactoryComponent {
    //  To be connected to app and ib
    attribute PolicyInterface     policy_interface;
    attribute FederationInterface federation_interface;
    attribute ChannelInterface    channel_interface;
    attribute FederatedNaming     federated_naming;

    provides ServiceManagement              service_management;
    provides ChannelManagement              channel_management;
    provides FactoryConfiguration           configuration;

};
home ServiceFactoryHome manages ServiceFactoryComponent{
};
```

```
    }; // FederatedApplication

}; // Pilarcos
```

A.3 Policy repository

```
module Pilarcos {
    module PolicyRepository {

        interface PolicyContextManagement {
            // Adds a new policy context as the child of the given
            // parent. Updates the parent accordingly.
            PolicyContextID addPolicyContext(
                in PolicyContextID parent_context_id,
                in PolicyContext new_context
            );

            // For updating values of a policy context.
            void updatePolicyContext(
                in PolicyContext policy_context
            );

            // Retrieves a policy context.
            PolicyContext retrievePolicyContext(
                in PolicyContextID context_id
            );

            // Deletes a policy context.
            // (Also deletes all children and updates parent.)
            void deletePolicyContext(
                in PolicyContextID context_id
            );

            boolean isPolicyContextStored(
                in PolicyContextID context_id
            );

            PolicyContextID getParentContext(
                in PolicyContextID context_id
            );

            PolicyContextIDSeq getChildContexts(
                in PolicyContextID context_id
            );

            // Creates an offer context under a service context,
            // duplicating the policy values of the service context.
            PolicyContextID createOfferContext(
                in PolicyContextId application_context_id,
                in ServiceTypeName service_type
            );
        };

        component PolicyRepositoryComponent {
            provides PolicyContextManagement context_management;
        };
        home PolicyRepositoryHome manages PolicyRepositoryComponent {
        };
```

```
    }; // PolicyRepository
}; // Pilarcos
```

A.4 Type repository

```
module Pilarcos {
    module TypeRepository {

        //
        // ---------- Interface types ----------
        //

        typedef string InterfaceType;
        typedef string InterfaceTypeName;

        //
        // ---------- Policies ----------
        //

        struct PolicyType {
            PolicyTypeName  name;
            CORBA::TypeCode type;
        };
        typedef sequence<PolicyType> PolicyTypeSeq;

        struct PolicyFrameworkType {
            PolicyFrameworkTypeName name;
            PolicyTypeSeq           policy_types;
        };

        struct PolicyFramework {
            PolicyFrameworkName     name;
            PolicyFrameworkTypeName policy_framework_type;
        };
        typedef sequence<PolicyFramework> PolicyFrameworkSeq;

        //
        // ---------- Service type ----------
        //

        typedef string ServiceInterfaceTypeName;

        enum InterfaceRole {
            provider,
            user
        };

        struct ServiceInterfaceType {
            InterfaceRole            interface_role;
            ServiceInterfaceTypeName name;
            InterfaceType            interface_type;
            PolicyFrameworkNameSeq   service_policy_framework_names;
        };
        typedef sequence<ServiceInterfaceType> ServiceInterfaceTypeSeq;

        struct ServiceType {
            ServiceTypeName         name;
```

```
    PolicyFrameworkSeq      policy_frameworks;
    ServiceInterfaceTypeSeq service_interface_types;
 };

//
// ---------- Architecture ---------
//

typedef string RoleName;
typedef string ArchitectureName;

struct Role {
    RoleName        name;
    ServiceTypeName service_type;
};
typedef sequence<Role> RoleSeq;

struct Binding {
    RoleName                 role_1;
    ServiceInterfaceTypeName role_1_interface;
    RoleName                 role_2;
    ServiceInterfaceTypeName role_2_interface;
};
typedef sequence<Binding> BindingSeq;

struct Architecture {
    ArchitectureName name;
    RoleSeq          roles;
    BindingSeq       bindings;
};

//
// --------- Architecture w/ policy framework graphs ----------
//

// These are used only between type repository and trader.

struct RolePolicyFramework {
    long architecture_role_index;      // index of role in architecture
    long role_policy_framework_index;  // index of p.framework in role
};
typedef sequence<RolePolicyFramework> RolePolicyFrameworkSeq;

struct PolicyFrameworkGraph {
    PolicyFrameworkType    policy_framework_type;
    RolePolicyFrameworkSeq policy_frameworks_in_roles;
};
typedef sequence<PolicyFrameworkGraph> PolicyFrameworkGraphSeq;

// index = index of policy framework in role (service type)
typedef sequence<long> PolicyFrameworkGraphIndexSeq;

// index = index of role in architecture
typedef sequence<PolicyFrameworkGraphIndexSeq> RoleIndexSeq;

struct ArchitectureWithGraphs {
    Architecture            architecture;
    PolicyFrameworkGraphSeq policy_framework_graphs;
    RoleIndexSeq            architecture_role_index;
```

```
        };

        //
        // --------- TypeRegistration interface ----------
        //

        interface TypeRegistration {
            void registerPolicyFrameworkType (
                in PolicyFrameworkType policy_framework_type
             );
            void registerServiceType (
                in ServiceType service_type
            );
            void registerArchitecture (
                in Architecture architecture_description
            );
            void emptyTypeRepository ();
            PolicyFrameWorkType describePolicyFrameworkType (
                in PolicyFrameworkTypeName name
            );
            ServiceType describeServiceType (
                in serviceTypeName name
            );
            Architecture describeArchitecture (
                in ArchitectureName name
            );
            ArchitectureWithGraphs describeArchitectureWithGraphs (
                in ArchitectureName name
            );
        };

        //
        // --------- TypeMatching interface ----------
        //

        interface TypeMatching {
            boolean isCompatibleInterface (
                in PSIType first_end,
                in PSIType second_end
            );
        };

        component TypeRepositoryComponent {
            provides TypeRegistration type_registration;
            provides TypeMatching type_matching;
            uses CosTrading::TraderComponents corba_trader;
        };
        home TypeRepositoryHome manages TypeRepositoryComponent {
        };

    }; // TypeRepository
}; // Pilarcos
```

A.5 Trader

```
module Pilarcos {
    module Trading {
```

```
        // note that ServiceOfferID != ServiceContractID
        typedef string ServiceOfferID;

        //
        // --------- OfferRegistration interface ----------
        //

        interface OfferRegistration {
            ServiceOfferID export(
                in ServiceContract service_offer
            );
            void withdraw(
                in ServiceOfferID service_offer_id
            );
            ServiceContract describeServiceOffer(
                in ServiceOfferID service_offer_id
            );
            void emptyTrader();
        };

        //
        // --------- Lookup interface ----------
        //

        interface Lookup {
            FederationContractSeq populate (
                in FederationContract  initial_federation_offer,
                in unsigned long    max_offers,
                in unsigned long    max_time_ms
            );
        };

        component TraderComponent {
            provides OfferRegistration                offer_registration;
            provides Lookup                           lookup;
            uses     TypeRepository::TypeRegistration type_registration;
            uses     TypeRepository::TypeMatching     type_matching;
            uses     CosTrading::TraderComponents     corba_trader;

        };
        home TraderHome manages TraderComponent {
        };

    }; // Trading
}; // Pilarcos
```

A.6 Public tourist service

```
module Pilarcos {
    module TouristServicePublic {

        // This will be removed when implicit contexts start working
        typedef string FederationContractID;

        typedef string           QueryExpression;
        typedef string           ReservationData;
        typedef string           InformationPage;
```

```
        interface HotelReservationProxy;
        struct SessionData {
            // BillID for possible pre-payment
            PaymentServicePublic::BillID      pre_payment_id;

            // Refs to all subservice proxies
            HotelReservationProxy             hotel_reservation_proxy;
            // CarRentalProxy etc..
        };

        struct PaymentTransaction {
            PaymentServicePublic::BillID  bill_id;

            // true=paid, false=needs payment
            boolean                       status;
        };
        typedef sequence<PaymentTransaction> PaymentTransactionSeq;


        // The only subservice available
        interface HotelReservationProxy {
            InformationPage getInformation(in QueryExpression query,
                                           in FederationContractID contract_id);
            void makeReservation(in ReservationData reservation,
                                 in FederationContractID contract_id);
        };

        interface TouristInfo {
            // Starts a service session.
            SessionData startSession(in FederationContractID contract_id);

            PaymentTransactionSeq getPaymentData(
                in FederationContractID contract_id);

            // Ends the service session. also acts as a post-payment
            // confirmation
            void endSession(in FederationContractID contract_id);
        };

    }; // TouristServicePublic
}; // Pilarcos
```

A.7 Public payment service

```
module Pilarcos {
    module PaymentServicePublic {

        typedef string FederationContractID;

        typedef string BillID;
        typedef string Account;
        typedef string Certificate;

        struct Bill {
             // the person who logged in with username xxx
            string biller;
            double amount;
            string description;
```

```
        };

        interface Payment {
            Bill getBill(in BillID bill_id,
                         in FederationContractID contract_id);
            void payBill(in BillID bill_id,
                         in Account account,
                         in Certificate certificate,
                         in FederationContractID contract_id);
            void refuseBill(in BillID bill_id,
                            in FederationContractID contract_id);
        };


        interface Billing {
            BillId getBillID(in Bill bill,
                             in Account account,
                             in FederationContractID contract_id);
            void invalidateBill(in BillID bill_id,
                                in FederationContractID contract_id);
            boolean isBillPaid(in BillID bill_id,
                               in FederationContractID contract_id);
        };

    }; // PaymentServicePublic
}; // Pilarcos
```

A.8 Public hotel service

```
module Pilarcos {
    module HotelServicePublic {

        typedef string FederationContractID;

        // These could be different from those defined in
        // TouristServicePublic

        typedef string          QueryExpression;
        typedef string          ReservationData;
        typedef string          InformationPage;

        struct PaymentTransaction {
            PaymentServicePublic::BillID  bill_id;

            // true=paid, false=needs payment
            boolean status;
        };
        typedef sequence<PaymentTransaction> PaymentTransactionSeq;

        interface HotelReservations {
            InformationPage getInformation(
                in QueryExpression query,
                in FederationContractID contract_id);

            void makeReservation(
                in ReservationData reservation,
                in FederationContractID contract_id);
        };
```

```
        interface HotelInfo {
            HotelReservations startSession(
                in FederationContractID contract_id);

            PaymentTransactionSeq getPaymentData(
                in FederationContractID contract_id);

             // Ends the service session. also acts as a post-payment
             // confirmation
            void endSession(in FederationContractID contract_id);
        };

    }; // HotelServicePublic
}; // Pilarcos
```

A.9 Connector

```
package Pilarcos.Connections.CORBA;
public class Connector {

    public static org.omg.CORBA.Object getConnection(
                String federation_contract_id,
                int federation_context_id,
                String interface_name,
                String interface_type,
                Pilarcos.FederatedApplication.ChannelInterface channel_interface){}

    public static org.omg.CORBA.Object getConnection(
                String federation_contract_id,
                InterfaceRef if_ref){}

    public static org.omg.CORBA.Object getConnection(IndirectRef id_ref) {}

}

package Pilarcos.Connections.RMI;
public class Connector {
    public static java.rmi.Remote getConnection(
                String federation_contract_id,
                int federation_context_id,
                String interface_name,
                String interface_type,
                Pilarcos.FederatedApplication.RMI.ChannelInterface channel_interface){}

    public static java.rmi.Remote getConnection(
                String federation_contract_id,
                InterfaceRef if_ref) {}

    public static java.rmi.Remote getConnection(IndirectRef id_ref) {}

}
```

Helsinki 2003