Pilarcos demonstration prototype – design and performance

Markku Vähäaho, Egil Silfver, Juha Haataja, Lea Kutvonen and Timo Alanko

Helsinki December 21, 2001
UNIVERSITY OF HELSINKI
Department of Computer Science

Contents

Chapter 1

Introduction

The Pilarcos project develops middleware solutions for automatic management of interorganisational applications. In the first part of the project, a federable middleware architecture [3] was designed and prototyped on the OpenCCM platform [2].

The prototype is a proof-of-concept implementation of the central Pilarcos middleware services that allow description, discovery and automated federation of autonomous services in the network. To support both conceptual and technical development of the middleware services, a business case called "Tourist Information Service" was used as the example application.

When considering the feasibility of the Pilarcos architecture, its effects on performance must also be evaluated. For this purpose, a series of benchmarks was executed to measure the relative cost of using Pilarcos middleware services. Again, the Tourist Information Service was used as the example case.

The purpose of this report is to present the business case, the functionality and structure of the prototype, and the benchmark results and conclusions drawn from them.

Chapter 2 describes the Tourist Information Service business case on a non-technical level. The case is not meant to be entirely realistic, but to bring up and demonstrate relevant points in the Pilarcos model

Chapter 3 discusses the design of the prototype. The chapter starts with an overview that should give a fair comprehension of the main features and overall structure of the prototype. Following that, central concepts, functionality and implementation are discussed in detail.

Chapter 4 introduces the benchmark arrangements and explains the results from the measurements. Benchmarks were conducted separately for the business service broker and the whole Tourist Information system. In addition, some predictions of scalability are made based on the measured values and basic queueing theory.

Chapter 5 concludes the report with a short discussion of the prototype and the benchmark results.

Chapter 2

Business case: Tourist information service

This chapter presents the business case around which the prototype implementation has been built. Only a basic model of the case is presented here; technical details are discussed in later chapters.

2.1 Business case requirements

In order to provide a concrete and sufficiently realistic context for evaluating and developing the Pilarcos infrastructure [3] a business case was specified.

The goal of the business case was not to specify an innovative business, nor to demonstrate the intended uses of Pilarcos middleware. Instead, the intention was to facilitate prototyping and to point out the differences of the Pilarcos infrastructure services compared to traditional middleware.

Requirements for the business case are:

- Provide enough complexities to enable evaluation of the real problems involved.
- Include only truly relevant complexities and leave out everything else.
- Relevant complexities include at least
  - specification of several business roles representing arbitrary clients and service providers,
  - possibility for a service provider to produce a federable service by combining other federable services, and
  - support for complex cooperation involving trust establishment and interactions between at least three autonomous organisations.

2.2 Business case description: "Tourist Information Service"

We consider a business case called "Tourist Information Service" to be suitable for our purposes. This section provides an overview of the case.

It is assumed that there exist several autonomous service providers providing diverse tourist services including travel information, hotel bookings, and weather services. The case is based on a portal service, the “Tourist Info”, providing travellers with combined service packages matching their needs. These service packages combine services from existing tourist service providers.

In addition to service integration, the portal service can also provide additional value in the form of service localisation and personalisation. It is assumed that there exist several competing tourist info services providing information about different travel locations. Usually neither the portal service nor the vertical services are free, and the tourist has some kind of electronic payment instrument available.

2.2.1 Business roles

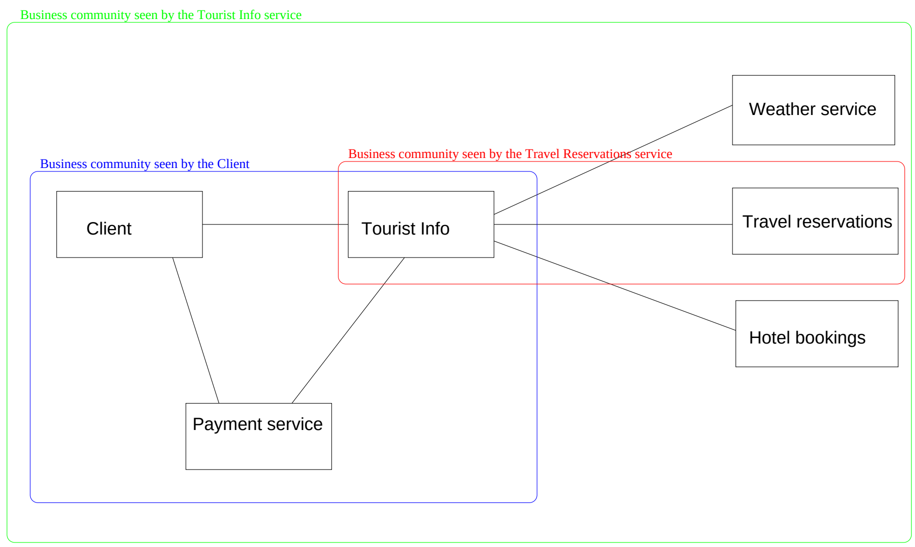

Figure 2.1: Business roles in the “Tourist Information Service”-case.

Figure 2.1 introduces the business roles and shows a few exemplary business communities visible to participants. A short description of each role follows:

- Client represents an arbitrary tourist needing (localised) services when travelling abroad. The client usually has pre-made payment contracts with one or several payment service providers. A payment contract may be a credit card or a more sophisticated electronic payment device.

- Payment Service represents an organisation providing a variety of payment services for its customers. It also acts as a trusted 3rd party in business transactions taking place.

- Tourist Info represents an organisation providing integrated tourist information services to arbitrary clients. The provider of the service may have existing relationships to external service providers or it may dynamically search and integrate globally available services.

- Vertical service providers (Weather service provider, Travel service provider, Hotel bookings provider) represent organisations providing vertical tourist information services.

The tourist uses the tourist info portal service through a client program, which may also provide planning assistance. The focus of the case is on the discovery and integration of services, where the Pilarcos middleware platform is directly involved. Actual service usage, such as booking a hotel, takes place only after the federation between the client and the service providers has been established; therefore, it is not modelled here.

2.2.2 Payment

The business case is not amenable to a traditional client-server architecture, because there are more than two roles that have mutual dependencies. These dependencies arise from the payment arrangements. We assume that a prepayment must be made by the client for the tourist information portal service, and the payment must be made via a trusted third party, the payment service. Figure 2.2 shows the interactions during the payment process.

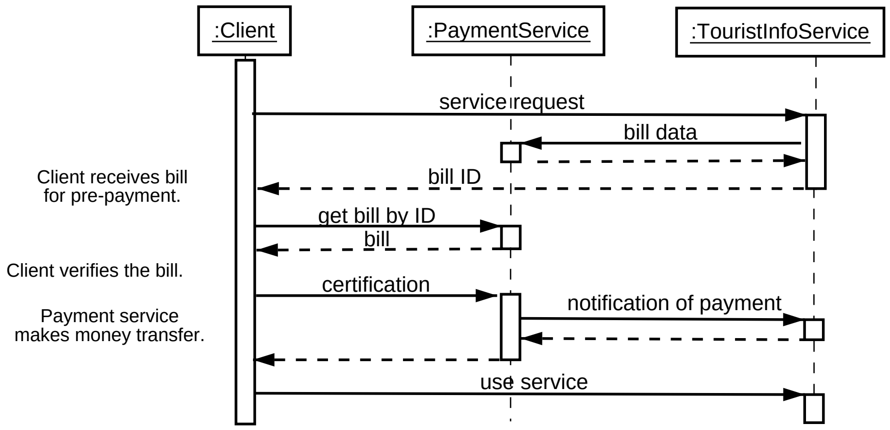

Figure 2.2: The payment procedure as a sequence diagram.

A chain of trust must be established between the three primary parties. Therefore, the bill from the tourist info service is sent to the client via the payment service. Having accepted the bill, the client sends a certificate – including, for example, a password – back to the payment service, which then proceeds with the money transfer. In this procedure, all three parties must agree on the payment method. It is thus not reducible to simple one-to-one client-server relationships.

The tourist info service may mediate payments between the client and the vertical service providers, or it may have a long-term contract with the vertical service providers and charge directly for only its own service. In either case, from the client's point of view the payment is made to the tourist info service provider.

Business service brokerage

From the client's point of view, the business architecture where tourist info service is used consists of three roles: the client itself, the Tourist Info portal, and the Payment Service. It is not visible to the client that the Tourist Info actually combines services from other service providers. The role of Pilarcos middleware from the point of view of the client is to help it find suitable tourist information and payment services, and to establish a federation between them.

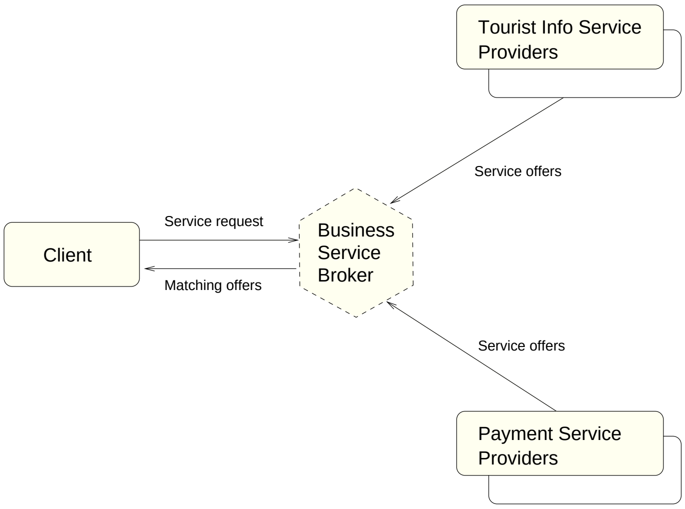

Figure 2.3: Business service brokerage in the business case.

The Pilarcos middleware service that mediates business services is the business service broker. Its role in the business case is illustrated in figure 2.3. The client, the Tourist Info service, and the Payment Service are mutually dependent, because they must agree on a payment method, as was explained above. Because of this, the business service broker must ensure that the candidates it offers for the business architecture roles are compatible with each other.

The Tourist Info may well again utilize Pilarcos middleware to contact the required subservices, or have them always available by some prior arrangement.

2.3 Usage scenarios

Two example usage scenarios for the Tourist Info service are given below.

2.3.1 Scenario I: Making hotel reservations from home

In the first scenario a virtual tourist Ville Virtanen is at home planning a weekend trip to Paris. He has already acquired a flight ticket and is now searching for a moderately priced hotel. Mr. Virtanen has three credit cards to use: VISA, MasterCard, and AmericanExpress. To do the search he opens up the Tourist Info Client software on his workstation and instructs the Client to look for hotel bookings in Paris. The Client then contacts a (remote) Tourist Info service that mediates hotel booking services in Paris area.

Examples of required service attributes are listed below.

Attribute	*Example value*
Area (city)	Paris
Needed subservices	HotelInfo
Maximum service cost	10 (euros)
Type of credit card	VISA, MasterCard, AmEx
Terminal capabilities	PC-SVGA

2.3.2 Scenario II: Using local information services

In the second scenario Mr. Virtanen has arrived to Paris and, having heard of the marvels of the Château, plans a day trip to Versailles the next day. To ensure that the trip will run as smoothly as possible Mr. Virtanen checks the weather-forecast and consults the local train schedule. In order to do this Mr. Virtanen again starts the Tourist Info Client, this time on his PDA, and requests local weather and travel information.

The PDA includes a positioning device, so the Client has the location information readily available and can start the search for TouristInfoService providing WeatherInfoService and TravelInfoService in Paris area. In addition, the limited processing capabilities of the PDA must be considered when selecting the service provider. The electronic wallet on the PDA holds only MasterCard information, which may further limit the selection.

Examples of required service attributes are listed below.

Attribute	*Example value*
Area (city)	Paris
Needed subservices	WeatherInfo, TravelInfo
Maximum service cost	any
Type of credit card	MasterCard
Terminal capabilities	PDA

Chapter 3

Prototype design

This chapter presents the structure and design of the prototype implementation developed during the Pilarcos I project. Section 3.1 presents an overview of the prototype, including development and usage scenarios. Section 3.2 describes the concepts central to the Pilarcos architecture in the context of the prototype. The functionality and implementation of the prototype components are presented in section 3.3. Section 3.4 describes how federation establishment proceeds on the prototype platform.

3.1 Overview

The current Pilarcos prototype is a proof-of-concept implementation of the run-time Pilarcos middleware components. To test the middleware functionality, skeleton implementations of the Tourist Info application components have also been written. Focus has been on the straightforwardness of implementation, not robustness, completeness or efficiency.

The prototype demonstrates the following Pilarcos features:

- Separation of distributed application architecture from program code to an explicit architecture description.
- Separation of service usage and management policies from program code to a central policy repository.
- Automated discovery of multiple, interdependent services by an advanced trading mechanism that uses the architecture description.
- Transparent and automated federation establishment between service providers and users, possibly involving policy negotiations.
- Middleware services that hide complexity related to federation management from application programmers.

Recent work in Pilarcos has focused on dynamically forming distributed applications from independent services in the network, and the prototype reflects this. For instance, the prototype does not as of yet include mechanisms for management of individual services, only for management of service federations.

The Pilarcos architecture includes software engineering tools that interface with the runtime infrastructure through metadata repositories [3, section 4.7]. However, only run-time infrastructure components have been prototyped so far.

The prototype was implemented in Java on a CORBA Component Model platform, OpenCCM 0.2 and ORBacus for Java 4.0.5. The implementation platform and development experiences are discussed in an accompanying report [2].

3.1.1 Prototype components

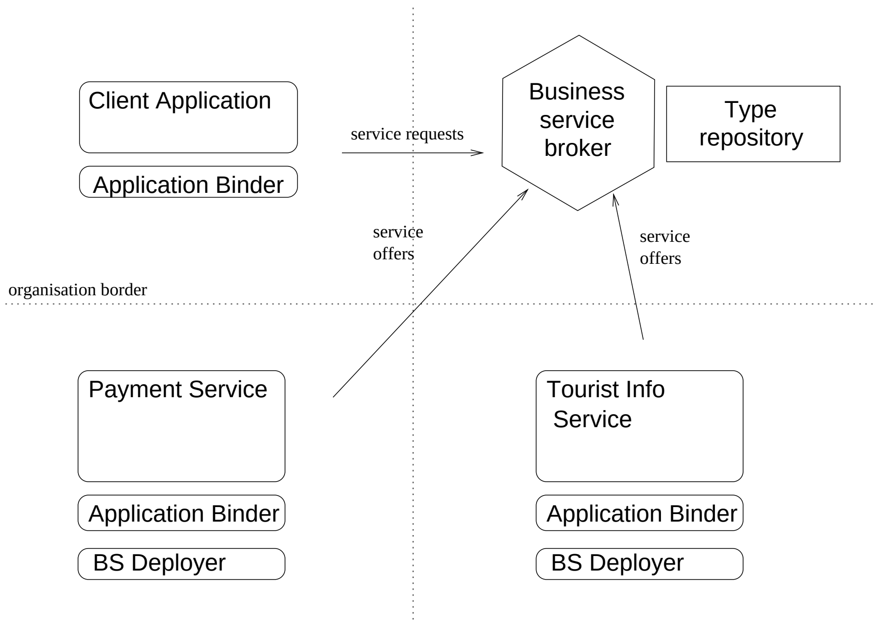

Figure 3.1: Overview of prototype components.

Figure 3.1 shows an overview of the components in the prototype. They are briefly explained below, starting with the infrastructure components.

Type repository. This is the globally accessible business type repository. Service type descriptions and business architecture descriptions are stored here. Local type repositories have not been implemented.

Business service broker. The business service broker is implemented as an advanced trading service. It uses the business architecture description to find compatible offers for all roles in the architecture in one step.

Application binder. Application binders establish and manage federations, hiding the complexity from the application components. In the prototype, they also function as simple policy repositories.

Business service deployer. Business service deployers instantiate server components when first used (late instantiation). The current implementation is very simple.

Client, Payment Service, Tourist Info. These are skeleton implementations of the application components from the Tourist Information Service business case (chapter 2). They are functional enough for testing the infrastructure components. Note that thus far, subservices from the business case (weather service, hotel booking, etc.) have not been implemented in the prototype. The focus has been exclusively on the business community seen by the client (figure 2.1).

Organisational separation has been implemented in the prototype by deploying the components within an 'organisation' (figure 3.1) in an independent component server and having them communicate over the network with the IIOP protocol.

3.1.2 Development process

Figure 3.2 depicts the main actions in developing a federated application that runs on the Pilarcos prototype platform. Although the Pilarcos model is for the most part symmetric for clients and servers, the application making the initiative to establish a federation is here called the *client application*.

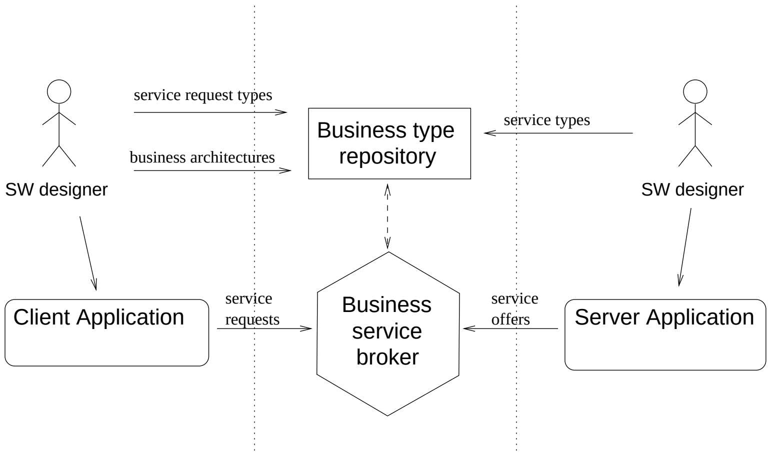

Figure 3.2: Overview of development process.

For service providers, main development steps are:

1. Choosing or developing a business service type.

   A business service type describes the policy framework and dependencies of a class of federable business services. Service types are stored in a globally accessible business type repository. If a suitable type does not exist, the service provider can create one and make it available in the repository.

2. Developing and deploying server components.

   Server application components must be programmed so that they implement the business service type. The Pilarcos middleware provides services designed to make programming federable applications easy.

3. Exporting service offers to the business service broker.

   A service offer describes a specific business service of some service type. It lists the interfaces and policies of the service. Service offers can be exported manually or by the server application via the Pilarcos middleware services.

For developers of client applications, main development steps are:

1. Choosing or developing a business architecture.

A business architecture can be seen as a configuration description of a federated system; however, instead of fixed components, it describes roles and their interconnections. The developer of the client application may choose an existing business architecture or compose a new one using available service types. For the role of the client, a service request type needs to be created. Business architecture descriptions are stored in the business type repository.

2. Developing the client application.

   When using federated services, the client application must specify the business architecture it wants to be established. The client application must therefore be programmed to function correctly in the architecture.

3. Deploying and running the application.

   Federation establishment is handled on behalf of the client by its application binder. The roles in the business architecture are populated by the business service broker upon an import request from the application binder.

The above description is quite simplified. Software development in the Pilarcos context has been discussed more thoroughly in an earlier report [3, chapter 3].

3.1.3 Run-time scenario

In this section, we present an overview of the sequence of events that take place when the prototype is started.

1. Startup and initialisation.

   Initial state of the system when the prototype is started is shown in figure 3.3. The business service broker and the type repository are started first and registered with the CORBA Naming Service. Business service deployers of the tourist information and payment services are also started; at startup, they export service offers to the broker. The client application and its application binder are deployed and started.

2. Importing service offers.

   Through the application binder, the client sends an import request to the business service broker. The request identifies the business architecture to be populated and includes the policies of the client for the federation.

   In response, the business service broker returns a compatible combination of service offers for all roles.

3. Federation establishment.

   The application binder of the client initiates the distributed federation protocol (figure 3.12). On receiving the prepare-service request, the business service deployers deploy and start server components. When the protocol ends, contracts have been negotiated and bindings established between the client and the servers (figure 3.4).

4. Payment for the tourist service.

   As described in chapter 2, the client makes a pre-payment for the tourist information service through the payment service. At both ends, method calls are intercepted and routed by the application binders.

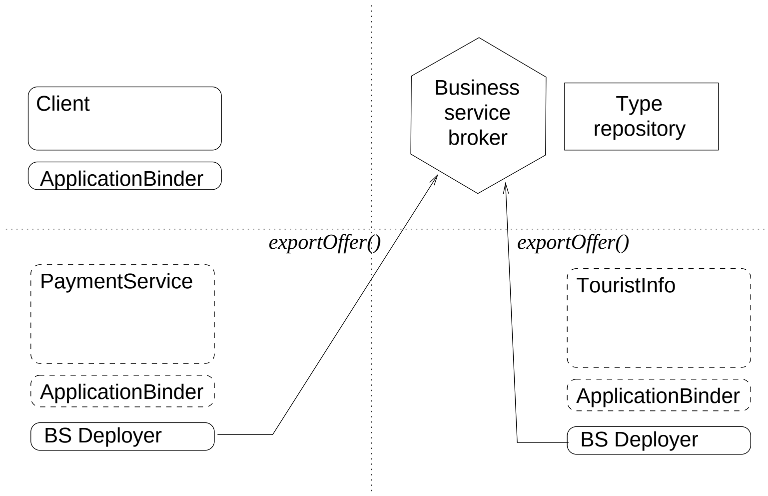

Figure 3.3: Initial state of prototype.

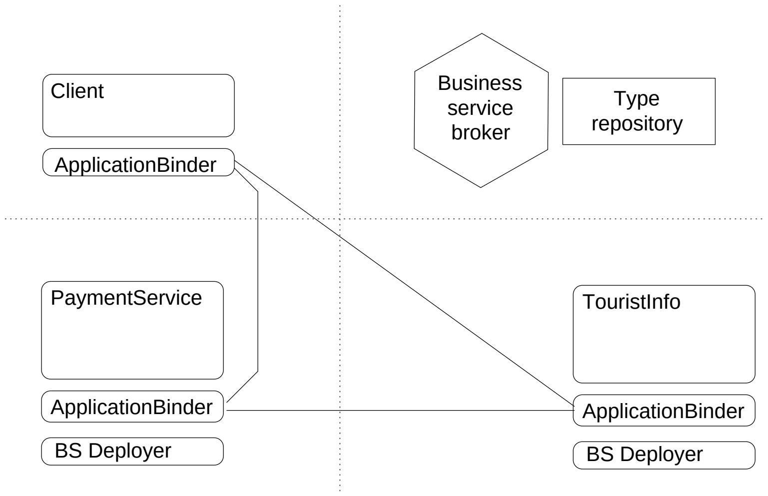

Figure 3.4: Federation established.

5. Usage of the tourist service.

   The client simulates using the tourist information service with a simple method call.

6. Federation tear-down.

   When the client has finished using the tourist information service, it orders the application binder to terminate the federation contract. The application binder initiates the termination protocol, which is similar to the distributed federation establishment protocol.

3.2 Central concepts

In the traditional client-server model, there are logically only two parties: the service provider and the client. Pilarcos extends the client-server model with a role-based architecture description, allowing more complex relations between service providers and clients.

This section aims to explain the *business service* and *business architecture* concepts as used in the Pilarcos prototype. After these fundamental concepts, the implementation of *policies* and *application interfaces* in the prototype are discussed.

3.2.1 Services and business architectures

In the Tourist Information Service business case, the tourist information service and the payment service are modelled as business services. A Pilarcos business service can require another service to function; in our example, the tourist information service needs a separate payment service for secure money transfer.

In Pilarcos, an explicit architecture description models the systems providing and using services as roles. The structure of the federated system that they form together is defined by the interconnections between roles. The architecture description is used at run-time by the Pilarcos middleware for automated discovery and federation of services.

In the prototype only business level – that is, interorganisational – architectures have been implemented. We speak of *business services* and *business architectures*. The same concepts could be applied within an organisation on the computational level [3, section 3.2.3].

3.2.2 Service types

Services are categorised by *service type*. From a developer's point of view, a service type is a template for service offers and service requests. When a service provider wishes to offer a service, it must either use a service type that is already registered to the type repository or create a new one and register it.

Because a service can require another service to function, a service type definition includes both provided and required services.

A *service type definition* consists of

- a unique name (within the naming context) for the service type;
- the interfaces that this service provides;
- the services and interfaces that this service requires; and
- the policy framework for this service.

The definition in figure 3.5 is directly taken from an initialisation file of the prototype implementation. It defines the service type TouristInfoService, which requires PaymentService to function. The client program can use the tourist information service through TouristInfoInterface. This is an abstract name for an interface whose technical definition may vary; see section 3.2.5 for more information. The prototype implementation uses IDL3 keywords for denoting CCM port types (facets, receptacles, event sources and sinks). In the above example, TouristInfoInterface is provided as a facet ("provides" in IDL3).

The payment service is used through BillingInterface. Another service type, PaymentService, provides this interface. As this demonstrates, a service can have different

```
service type TouristInfoService {
    provides TouristInfoService {
        TouristInfoInterface: provides
    }

    requires PaymentService {
        BillingInterface: uses
    }

    policy framework {
        Area: string
        Price: double
        Available_subservices: set
        Payment_methods: set
        Client_terminal_type: set
    }
}
```

Figure 3.5: Service type definition of TouristInfoService.

interfaces to different parties. The parties are modelled as separate roles in the architecture description.

The policy framework determines the policy parameters related to the service. Policy frameworks and policies are discussed in section 3.2.4. In Pilarcos, all service properties are modelled as service-specific policies.

Service requests must conform to a service request type. Service request types are defined as service types with one exception: whereas a service type must provide exactly one service, a service request type may only require other services. For an example, refer to appendix B, where all service type definitions for the tourist info case are listed.

Service types can be organised in an inheritance hierarchy, similarly to the CORBA trader service types [4]. However, the functionality to support this has not yet been implemented in the prototype.

3.2.3 Business architectures

From a run-time point of view, a business architecture is a template model of a federated system. It consists of a set of roles and their interconnections. The roles are placeholders for systems providing and using business services.

The architecture description is used at run-time for discovering services and establishing federations. First, the business service broker uses it to find compatible service offers for all empty roles in the architecture. After this, the federation protocol uses it for establishing bindings between parties. This workflow is explained in section 3.4.

The business architecture to be used is dictated by the client – that is, the party initiating federation establishment. The interfaces of a business service are completely described by its service type; services are thus independent of business architectures, and can be re-used in several different architectures. This arrangement resembles that of the Wright architecture description language [1], which separates components and their port descriptions from complete configurations.

A *business architecture description* consists of

- role definitions that state the service types related to each role;
- connections between the roles, including interface and policy bindings; and
- optionally, policies related to the architecture.

```
business architecture TouristInfoArch {
    role Client {
        service type: TouristInfoUser
    }

    role Server {
        service type: TouristInfoService
    }

    role PaymentSystem {
         service type: PaymentService
    }

    binding Client.TouristInfoUser - Server.TouristInfoService {
        policy bindings {
            Area = Area
            Price = Price
            Needed_subservices = Available_subservices
            Payment_methods = Payment_methods
            Terminal_type = Client_terminal_type
        }
    }

    binding Client.TouristInfoUser - PaymentSystem.PaymentService {
        policy bindings {
            Payment_methods = Payment_methods
        }
    }

    binding Server.TouristInfoService - PaymentSystem.PaymentService {
        policy bindings {
            Payment_methods = Payment_methods
        }
    }
}
```

Figure 3.6: Business architecture definition for TouristInfoArch.

Figure 3.6 shows an example of a business architecture description. Again, it is a verbatim copy from an initialisation file of the prototype implementation.

The example defines three roles, familiar from the business case description in chapter 2, and associates service types with them. (In the prototype, just one service type per role can be defined.) In effect, the interfaces defined by a service type become the interfaces of the role. The relation of roles to service types is illustrated in figure 3.7.

The bindings define which services in which roles should be connected together. Implicitly, this also binds the interfaces of the services to each other. Only one-to-one bindings have been implemented in the prototype thus far.

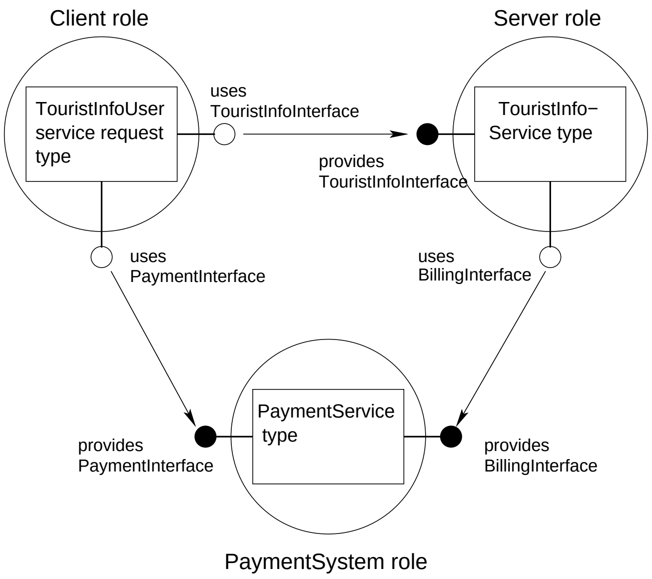

Figure 3.7: Business architecture illustrated.

An important part of the binding definition is binding of service-specific policies. These bindings have the effect that a federation cannot be established unless the parties agree on the values of these policies. For example, in figure 3.6, the client policy named 'Needed_subservices' is bound to the tourist info service policy named 'Available_subservices'. Their constraints for these policies thus have to be compatible for federation to be possible.

See also how the 'Payment_methods' policies of all three roles are bound together. This ensures that the services chosen for the roles are able to agree on a payment method (e.g., a specific credit card). The business service broker uses this information to rule out offers that are not compatible with each other. In effect, the prototype generalises the properties and constraints of traditional trading services as policies.

Policies constraining roles, bindings and whole architectures could also be defined in the business architecture description. However, these kind of policies have not been implemented in the prototype.

3.2.4 Application policies

The Pilarcos architecture aims to ease administration of large distributed systems. This is done in part by collecting all administrative policies to a central place, the policy repository of the organisation. Any kind of behaviour restricting decision that can be expressed as a constraint can be formulated as a policy.

Policies can affect entities ranging from whole organisations to individual applications and network channels. In Pilarcos, all policies are part of a policy hierarchy tree, where lower levels inherit policy decisions from upper levels. The policy hierarchy resides in the policy repository of the organisation. Pilarcos middleware components and application components interact with the policy repository to uphold the policies.

Establishing a federation between systems in independent organisations essentially means negotiating a contract between them. When a contract has been negotiated, each organisation stores it as a new policy context in its policy repository [3, section 4.7.2].

The prototype implementation currently does not have a policy repository. Policies are stored locally by the application binder of each application component. The policies also do not form a hierarchy; only application-level policies are currently used. Policies are implemented as constraint expressions for policy variables. The names and types of policy variables are defined by a *policy framework* (for an example, see figure 3.5). Three types of variables are available: double precision real numbers, strings, and string sets.

Policy constraints are expressed in OMG Constraint Language [4, appendix B]. Figure 3.8 shows an example of policy constraints. The substring matching operator (~) is used for testing containment in a string set.

```
policies {
    Area: Area == 'Paris'
    Price: Price < 10
    Needed_subservices: 'HOTEL_INFO' ~ Needed_subservices
    Payment_methods: 'VISA' ~ Payment_methods or 'MASTERCARD' ~ Payment_methods
    Terminal_type: 'PC' ~ Terminal_type
}
```

Figure 3.8: Example of policy constraints.

3.2.5 Application interfaces

The Pilarcos federable system architecture separates abstract interfaces from their concrete technical implementations [3, section 2.4.1]. Since the prototype does not yet have a complete type repository implementation, this is done in a very simple way. Every interface has a name that represents the interface as an abstract entity; there can be several different OMG IDL definitions of the interface. As defined in section 3.2.2, service types only list abstract interfaces; service offers and service requests define the concrete interfaces used by each party. Concrete interfaces are recognised by their OMG Interface Repository identifiers.

Interceptor components that convert between different technical versions of an interface can be stored in the type repository. They are meant to deployed automatically by the Pilarcos middleware services when federations are formed; this functionality will be added in a future version.

Bindings between application interfaces are implemented as simple CORBA object references. In effect, the ORB is used as the channel controller [3, section 4.6.2].

3.3 Prototype components

In this section, the prototype implementations of Pilarcos middleware components are discussed in some technical detail. It is recommended that the reader takes a look at the OMG IDL3 definitions of these components, found in appendix A, while reading these descriptions. The IDL3 definitions have been written with simplicity and readability as primary objectives.

In the prototype, all components within an 'organisation' (figure 3.1) run in an independent component server. The IIOP protocol is used for communication between them. Broker and type repository references are resolved using the CORBA Naming Service [5].

3.3.1 Type repository

The business type repository implementation in the prototype is very simple. It is essentially a storage for registering and retrieving service types, business architecture descriptions and interceptors for differing versions of interfaces. It is currently used by the business service broker and the application binders.

Because the business service broker is built on the CORBA Trading Service, the type repository keeps business service types in the Service Type Repository of the CORBA Trading Service. The type repository component resides in the same component server as the business service broker. Interfaces of the type repository and broker components are illustrated in figure 3.9.

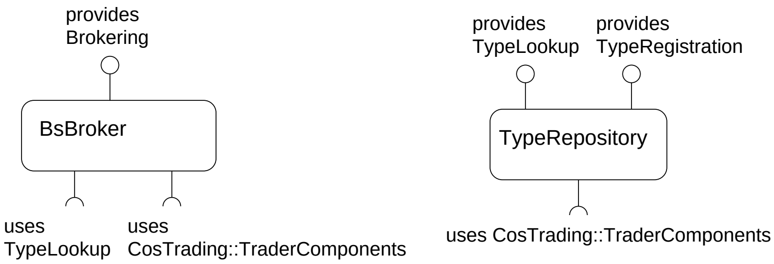

Figure 3.9: Interfaces of business service broker and type repository.

3.3.2 Business service broker

The business service broker implementation can be described as an advanced trading service that is able to find compatible candidates for several roles in one step. It offers two operations: importing and exporting business service offers.

Functionality

Service offers are exported one at a time by service providers. In the prototype, the business service deployers of the tourist info service and the payment service do this when they are first started.

Import requests are made by application binders on behalf of the application components when federable business services need to be found from the network. The service request from the application binder contains the business architecture whose roles are to be populated and the role of the requesting application in it.

On receiving an import request, the broker searches its offer database for suitable offers for each role in the business architecture. The policy constraints of the request and the offers are checked for compatibility according to the policy bindings (section 3.2.3). As a result, tuples of service offers containing one offer for each role is returned. The offers within a tuple are always compatible with each other and with the original request.

In addition, to make federation negotiations easier and quicker, the offers in a tuple are 'narrowed', that is, an intersection is made of the policy constraints by the broker. For example, if there is only one common payment method in the policy constraints of the offers, all other payment methods are deleted from the offers before they are returned. The original service request is also narrowed and included in the resulting tuple.

Because initial matching is done by the broker, the resulting tuples are called *service contract drafts*. They form a basis for further negotiations. It is up to the requesting application to choose the best one of the service contract drafts suggested by the business service broker and to start the federation establishment protocol.

Implementation

The current broker implementation is based on the CORBA Trading Service. Because the CORBA trader can only find offers for one service at a time, the broker uses a depth-first search algorithm in which offers are imported for one role at a time. Relevant policy constraints from the service request and from the offers imported for previous roles are concatenated together in the import request made to the CORBA trader. The process is simplified somewhat by the decision to express policy constraints directly in the OMG Constraint Language. Still, string sets are not directly supported by the CORBA trader and need special handling.

An unfortunate consequence of using the CORBA Trading Service is that service offers cannot contain arbitrary policy constraints; only fixed values for policy variables are possible.

The current broker prototype is limited in that the full depth-first search is not implemented, and at most one tuple of offers is returned per an import request. In general, the broker is a critical component in the architecture, and it will be redesigned and reimplemented in a better form in the future.

3.3.3 Application binder

An application binder component is associated with every Pilarcos application. The application binder establishes and manages federations and routes federated service requests, hiding the complexity from the application component.

Appendix C contains an example of a client program (the tourist information client) using the application binder to establish a federation and make federated method calls. Interfaces of the client and its application binder component are illustrated in figure 3.10. Application-specific interfaces are marked with thicker line in the figure.

Functionality

For client application programmers, the application binder offers a binding interface through which it is easy to make broker queries and to establish federations. Only the business architecture to be populated is given by the application program when making a broker query; policies for the request, which include service selection criteria, are automatically extracted from the current policy context of the application.

The service contract drafts returned by the business service broker can be examined in the application program. Once the most suitable service contract draft has been chosen, federation between the application and the offered services can be established with a single bind operation. When this operation is performed, the application binder starts the

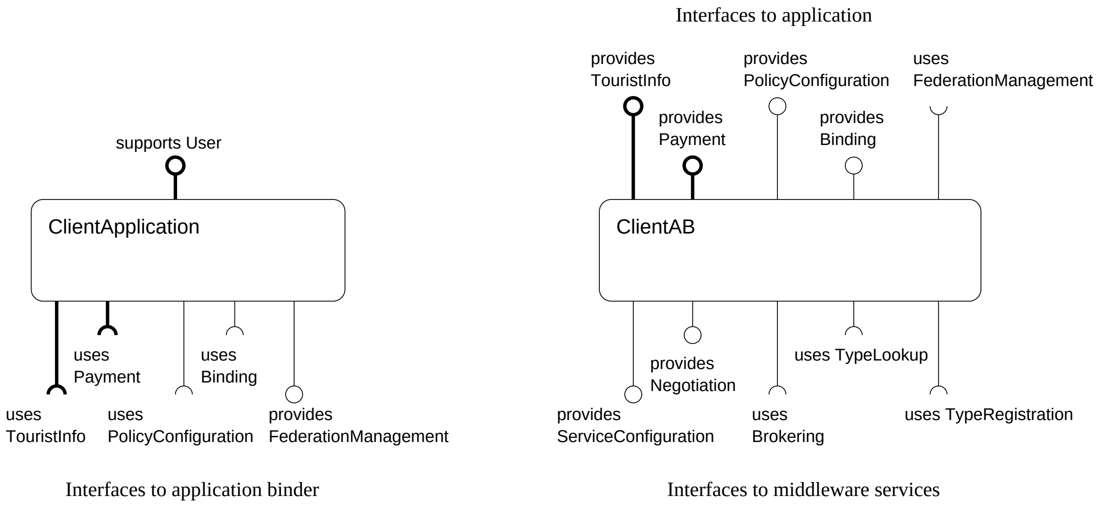

Figure 3.10: Interfaces of tourist information client and application binder.

federation establishment protocol. During this protocol, it distributes the service contract draft to all participants, negotiates a final service contract, and establishes bindings. The protocol is described in more detail in section 3.4.

For server application programmers, the application binder offers an even more transparent interface. Establishing a federation is completely automated; again, the policies to be used for the federation are looked up in the current policy context. Method invocations from federated clients are routed by the application binder to the corresponding server skeleton methods.

The negotiated contract governing all parties in the federation is called the *service contract*. Service contracts – and thus federations – are distinguished in the application by service contract identifiers. In addition to the service contract, bindings within the federation are governed by *binding contracts*. If necessary, the contents of both contracts can be examined by the application with the operations provided by the application binder.

In the prototype, the application binder itself functions as a policy repository, offering an interface for creating policy frameworks and setting and querying policies. Currently, only a single layer of application-specific policies is used.

Implementation

The prototype implementation of the application binder connects to the application component with CORBA Component Model facet-receptacle-bindings, and resides in the same component server as the application component.

Service interfaces – for example, the tourist information service interface – are also available to both the client and the server application through facets and receptacles. Federated bindings are implemented as simple CORBA object references between the application binders, and federated method calls pass through the application binder on both sides: the application binder intercepts all calls. In a future version, the application binder can use this facility to enforce policies.

The facet-receptacle-connection approach makes implementation simple, but has the drawback that the application binder must be statically configured for the services that the application component above it provides or uses. Application binders are thus not generic and have some application-specific code. Although it is probably not practical to try to make application binders completely generic, the application-specific parts could be separated and perhaps generated automatically. This is an area for future research.

In the prototype, federations cannot be renegotiated and must be terminated by an explicit operation by the client application.

3.3.4 Business service deployer

The purpose of the business service deployer is to deploy the components needed to provide the business service. This is done when the business service deployer receives a prepare-service request from an application binder in the first step of the federation establishment protocol.

The prototype implementation is very simple. When the prototype is initially run, only the business service deployers of the tourist info service and the payment service are deployed and started. At startup, they read their associated service type and service offers from a text file and register them with the type repository and the business service broker. When a business service deployer receives the first prepare-service request, it deploys the server component and the associated application binder component. After this, control is passed to the application binder.

The business service deployer, application component and the application binder are all installed in the same CCM component server. By necessity, the late instantiation causes a noticeable delay when the services are first used.

Interfaces of the tourist information server, its application binder and business service deployer are illustrated in figure 3.11. Application-specific interfaces are marked with thicker line in the figure.

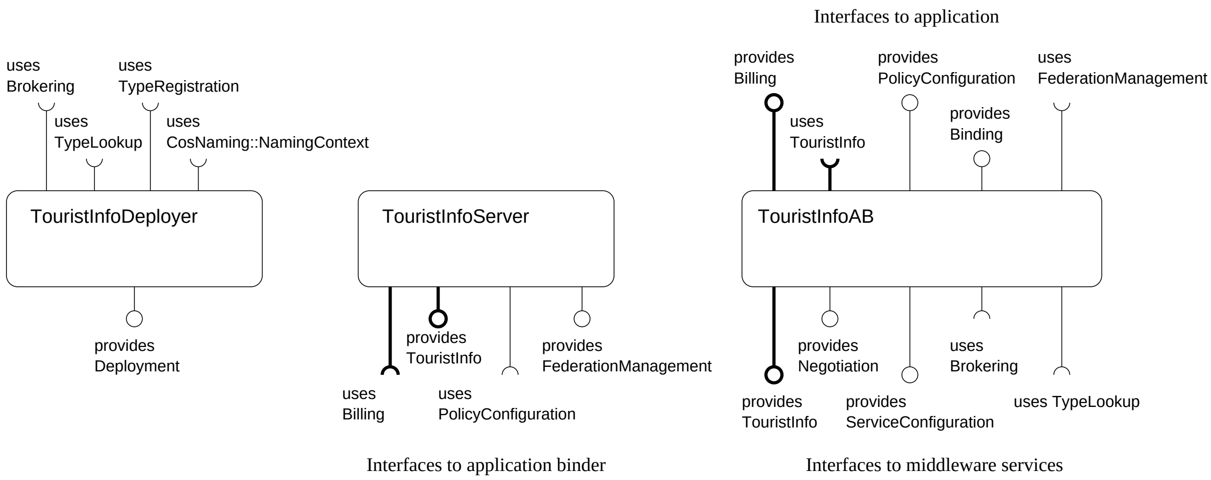

Figure 3.11: Interfaces of tourist information server components.

3.3.5 Application components

The application components implemented in the prototype are the tourist information client program, the tourist information server, and the payment server. All of these are skeleton versions with just enough functionality for testing the middleware components.

The role of the business service broker is to select matching offers from a database that includes a multitude of them. In the prototype there is only one implementation of the tourist information server and the payment server. To have more offers in the broker, either the servers must export more than one offer, or the servers must be deployed multiple times with different policies. The former approach has been taken in the tests conducted with the prototype.

Client program

The client program first changes policies in the current policy context, does a broker query, and establishes a federation with the tourist information and payment servers.

Pre-payment for the tourist information service is simulated following the sequence in figure 2.2. Usage of the tourist information service is simulated by a method call that returns a block of raw data from the tourist information server. After use, the client terminates the federation by an explicit unbind operation. The application code of the client program, stripped of debug operations, is listed in appendix C.

For benchmarking, the client program is able to fork threads that repeatedly perform the query-bind-use-unbind cycle. Ongoing user activity can be simulated by consecutive requests to the tourist information service with a configurable frequency.

Tourist information server

The current implementation of the tourist information server does very little. It participates in the payment sequence (figure 2.2) when a new client arrives, and simulates the workload caused by service requests by predetermined delays. During a service request, subservice and database access are simulated by idle waiting (sleeping). Post-processing of the data is simulated by running in a busy loop. Both delays are configurable in milliseconds for benchmarking purposes.

The implementation is thread-safe and one server can, in theory, handle an arbitrary number of concurrent clients.

Payment server

The payment server simulates payment service by keeping a table of bills and payments. Like the tourist information server, the payment server simulates workload by an idle wait and a busy loop for each payment operation. Both delays are configurable.

The implementation is thread-safe.

3.4 Federation establishment

This section describes how federations are established in the prototype implementation. An overview of the procedure is given first, followed by a more detailed explanation of the distributed federation protocol. Iterative and concurrent versions of the protocol are discussed along with an example from the tourist information case.

3.4.1 Overview

The complete federation establishment procedure can be divided into four phases:

1. business architecture population;
2. preparation of business services;
3. binding contract negotiation; and
4. binding contract realisation.

The business architecture population phase is centralised and performed by the business service broker. It has been discussed in section 3.3.2. The remaining phases are carried out by a distributed protocol called the *federation protocol*.

This discussion focuses on the distributed federation protocol. We assume that the business architecture has been populated: the client application has received a set of service contract drafts from the business service broker, and chosen one of them as the basis for forming a federation. Note that although the Pilarcos model is for the most part symmetric for clients and servers, we call the application that makes the initiative to establish a federation the *client application*.

The distributed federation protocol has four purposes:

- to pass the service contract draft to all involved parties;
- to verify that the services chosen for the roles are running and that the offers are still valid;
- to negotiate exact values for both federation-wide and one-to-one policies, forming contracts; and
- to distribute interface references for communication channel establishment.

The fact that policies of the offers in a service contract draft have been narrowed to a compatible set by the broker makes the following distributed binding contract negotiation more robust, easier to implement, and more scalable.

3.4.2 Phases of federation protocol

Service preparation. In the service preparation phase, the business services chosen for the business roles are notified of the federation that should be established. The application binder of the client application initiates the phase by sending a service preparation request with the service contract draft to the parties it has to establish a direct binding to. The bindings to be established are dictated by the business architecture description.

The server-side middleware service that handles the service preparation request is the business service deployer. On receiving a service preparation request, it checks whether the service is already running; if not, it instantiates the needed server components and the corresponding application binder. After this, it hands the service contract draft to the (server-side) application binder, which takes over the protocol from there. The application binder again sends a service preparation request to those parties it has to establish a binding to.

The service preparation process goes on, following the topology of the business architecture in a tree-traversal fashion, until an application binder that has no more parties to contact is reached. When this happens, the federation protocol moves on to the binding contract negotiation phase.

Contract negotiation. As a response to the prepare service request, the sending application binder receives an interface reference to the application binder in the other domain. It then starts the negotiation phase with the other application binder. The negotiating application binders try to find exact values for all of their common policies; this is simplified by having only compatible sets of values in the service contract draft. The negotiation may require one or more rounds depending, for example, on the types of the policies and whether the parties revealed all policy rules in their service request or offer. In normal conditions, the negotiations should not fail. The result of the negotiation is a binding contract between the parties.

The binding contract negotiation phase proceeds backwards on the service preparation tree, until all binding contracts have been negotiated or the negotiation procedure gets aborted for some unexpected reason.

Contract realisation. After the binding contracts have been negotiated, they can be stored and the concrete communication channels can be instantiated. The channels may also be torn down and re-instantiated according to the previously negotiated binding contracts. In addition, some or all of the binding contracts can be renegotiated at any time by any of the parties.

3.4.3 Iterative and concurrent protocol versions

In the *iterative* version of the protocol, the application binder sends a service preparation request to each connected role sequentially, waiting an answer to the previous request before sending the next one. The iterative protocol in the tourist information service case is illustrated in figure 3.12.

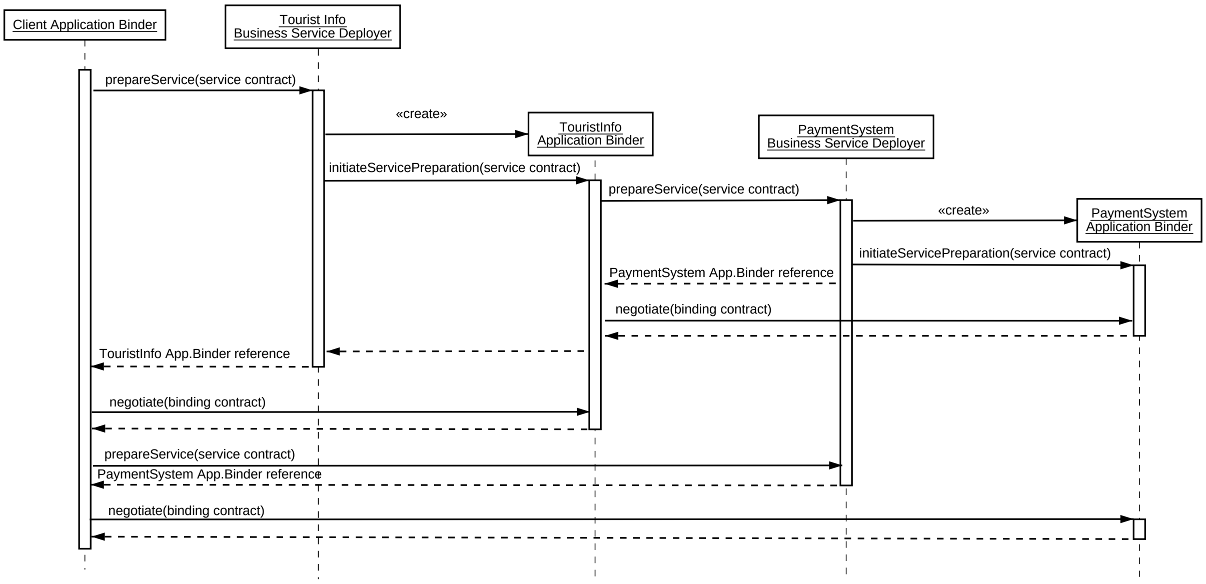

Figure 3.12: Federation establishment in the example case.

Since the iterative protocol proceeds in a depth-first manner, the first parties to negotiate are the ones 'farthest' from the initiator in the business architecture. If policies of several roles are bound together in a chain-like way, the results of the first negotiations will restrict the options available for subsequent negotiations in the chain. However, because compatibility of policies has been verified by the broker, the negotiations should not fail in normal conditions.

Concurrency can be exploited in the protocol by sending service preparation requests to all parties concurrently, without waiting for replies. Thus we have the *concurrent* version of the federation protocol. However, concurrent distributed negotiation protocols are difficult to construct and verify; moreover, it is not clear that the performance benefit obtained this way is significant except in some specific cases.

3.4.4 Implementation

In the prototype, both the iterative and concurrent versions of the federation protocol have been implemented. The latter works in the tourist information case, but is not guaranteed to work in any other architecture. At present, binding contract negotiations always succeed in one round (there is no iteration).

Federations can also be torn down. In the current implementation, the client initiates the tear-down procedure, and the servers always consent to it. The tear-down protocol traverses the business architecture graph in the same way as the federation protocol, but is simpler because no negotiations need to be performed; propagating a simple termination request is sufficient.

Chapter 4

Performance benchmarks

The prototype implementation of the Pilarcos runtime discussed in chapter 3 was benchmarked to assess the feasibility of the architecture.

More specifically, benchmarking of the Pilarcos prototype aimed at:

- Evaluating prototype behaviour in a realistic scenario.
- Measuring prototype scalability.
- Locating possible bottlenecks in the system.

The benchmarking process also helped in locating and fixing several bugs and design flaws in the Pilarcos middleware.

Because business service broker usage is physically separate from and independent of federation establishment and federated service usage, benchmarking was done in two separate parts. In the first part, performance of the business service broker in different conditions was measured. Results are presented in section 4.1. In the second part, federation establishment and service usage was benchmarked, with broker queries simulated by a fixed delay derived from the broker measurements. These results are presented in section 4.2.

Section 4.3 presents a brief analysis of the performance and scalability of the architecture based on the benchmark results.

4.1 Broker performance

In a typical usage scenario, importing offers from the business service broker is a far more frequent action than exporting offers. In addition, only import request service times affect the performance experienced by clients. Therefore, only import requests were benchmarked.

Because the business service broker is a centralised server, response times under different loads can be calculated using queuing theory from service times. For this reason, using a single measurement client was deemed sufficient.

Note that because the prototype implementation of the business service broker internally uses the CORBA Trading Service, the benchmarks directly reflect performance of the trader implementation.

The prototype implementation of the broker is limited, as explained in section 3.3.2. At most one service contract draft (offer tuple) is returned per query. The broker also does not have the ability to take backsteps in searching.

4.1.1 Benchmarking environment

Two identical workstations were used for broker benchmarks:

- Workstation 1 was running measurement client software within a single component server.
- Workstation 2 was running the business service broker and the business type repository within a single component server, plus the ORBacus Trading Service used by the broker as a separate process.

The workstations were attached to an isolated network segment without any external traffic.

Each workstation had:

- Linux 2.2.19
- Pentium III, 1 GHz, 512MB RAM
- 100 Mbps Ethernet connection

The software platform used on the workstation was:

- Java SDK 1.2.2, green threads, no JIT, 200MB heap size
- ORBacus 4.0.5 for Java, `blocking` clients, `threaded` servers
- ORBacus Trader 2.0.0 for Java
- OpenCCM 0.2

4.1.2 Benchmark setup details

The business architecture description from the tourist information service case (appendix B) was used in all measurements. In this architecture, there are three business roles in all, with two (tourist information service and payment service) to be populated by the broker. Where not stated otherwise, the policy frameworks of TouristInfoService and PaymentService types were also those of the business case examples.

Service offers and service requests were generated in advance by a script, and stored into files. Offers were exported to the broker and requests read into client memory at startup, before starting measurement.

A measurement round consisted of 3000 consecutive import requests made and timed by the measurement client. Because there was only one client and network delay was negligible, query response times measured by the client are equal to service times in the broker.

Timing was done using `System.currentTimeMillis()` method of the Java SDK API that returns the current system time with a precision of one millisecond. Times were logged to a file on the local hard disk; no other disk activity occurred during the measurements.

All measurements were repeated for three rounds, with all components restarted at the beginning of the round.

4.1.3 Measurement parameters

Service times of import requests were measured in respect to the following parameters.

- Number of service offers in the broker.

  Varying numbers of unique service offers for the TouristInfoService service type were generated and exported to the broker. Number of offers for PaymentService was kept constant at five.

- Policy complexity.

  Service offers and requests were generated with two levels of complexity. In *simple* offers and requests, there is one constraint in each policy. In *complex* requests there are two constraints in each set-type policy, and in complex offers there are three constraints in each set-type policy, joined with boolean operators `and` and `or`.

- Number of policies in requests and offers.

  Seven extra policies were added to the policy frameworks of TouristInfoService and PaymentService types.

- Number of different requests.

  The default number of unique requests used was five. A comparative test was made with 500 unique requests. This was done to find out whether the ORBacus Trader caches results and is important for the interpretation of the other results.

- Java virtual machine version.

  An additional benchmark was run with a newer Java virtual machine, HotSpot Client VM version 1.4.0beta2, running in mixed (interpreted/compiled) mode.

Requests and offers were matched so that the hit rate of import requests was about 50% for all tests.

4.1.4 Results: service time vs. number of offers

Measured service times per import request versus number of offers in the broker are shown in figure 4.1.

It is clear from the figure that service time is directly proportional to number of offers. The ORBacus trader probably performs a linear search when constraint expressions are used in requests. However, this is not as alarming as it seems, because the relationship concerns only offers of the same service type. We presume that in a typical business service broker, there are less than hundreds of offers per service type.

Complexity of policy constraints seems to affect service time only by a constant factor. This is to be expected of a linear matching algorithm when hit rate is kept the same. However, using a CORBA trader in the prototype limits the complexity of offers significantly; the final Pilarcos broker may exhibit different behaviour.

Number of different service requests does not affect service time. (The discrepancy in the figure to the baseline case is due to the fact that the hit rate is not exactly the same.) Neither the ORBacus Trader nor the Pilarcos broker that uses it cache query results. This means that benchmarking can be done with just a representative sample of requests.

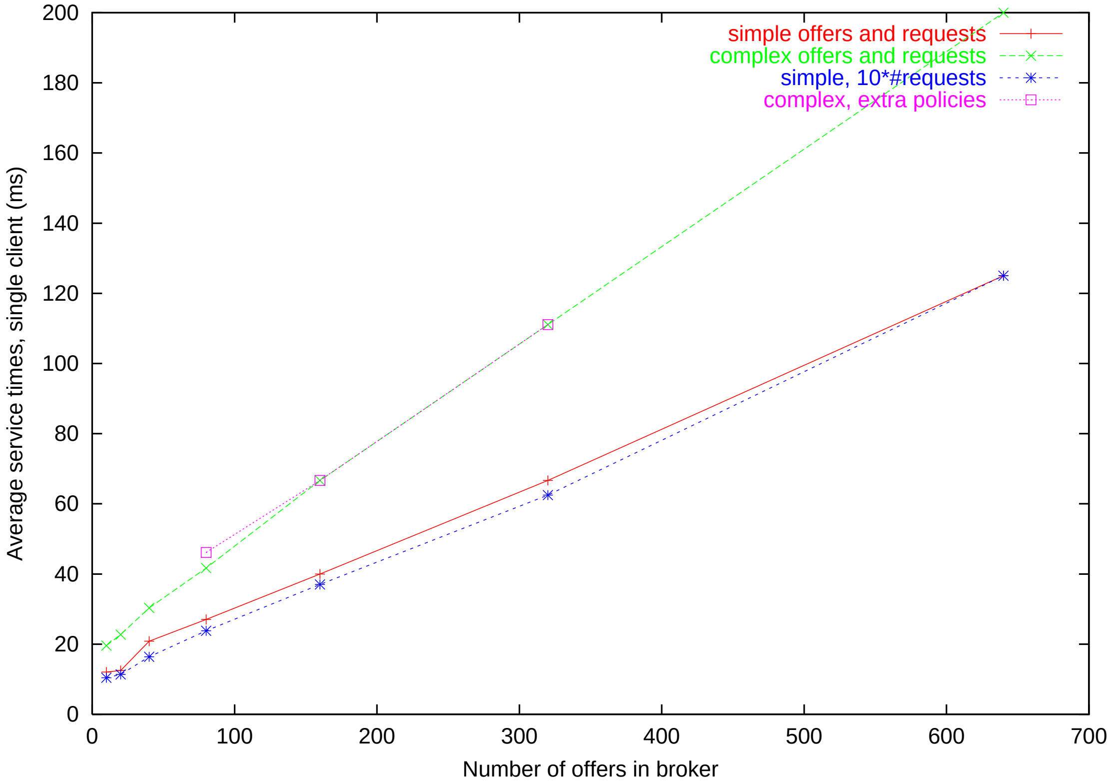

Figure 4.1: Broker service times with different parameters.

Increasing the number policies does not seem to significantly affect service time. This is probably true as long as the number of policies is relatively small, as was the case here. Again, using a CORBA trader in the prototype means that the results do not necessarily reflect the behaviour of a complete Pilarcos broker.

4.1.5 Results: effect of JVM version

The results presented above were all obtained with the default Java version 1.2.2, running in interpreted mode. The set of tests with complex offers and requests was also run with the new Java HotSpot Client virtual machine version 1.4.0beta2. The comparison is presented in figure 4.2.

The better performance of the new HotSpot virtual machine is due mainly to two factors: adaptive compilation together with rapid memory allocation and garbage collection [6]. The difference is approximately of a constant factor but is great enough to impact scalability.

The effect of adaptive compilation is visible in figure 4.3 which shows service times of 5000 consecutive import requests in averages of 100 requests. When the broker is started, the HotSpot virtual machine runs the program code in interpreted mode, resulting in performance comparable to the older virtual machine. As more requests are made, the HotSpot compiles code on the fly beginning with the most performance-critical parts. Only after 2000 requests does the service time stabilise.

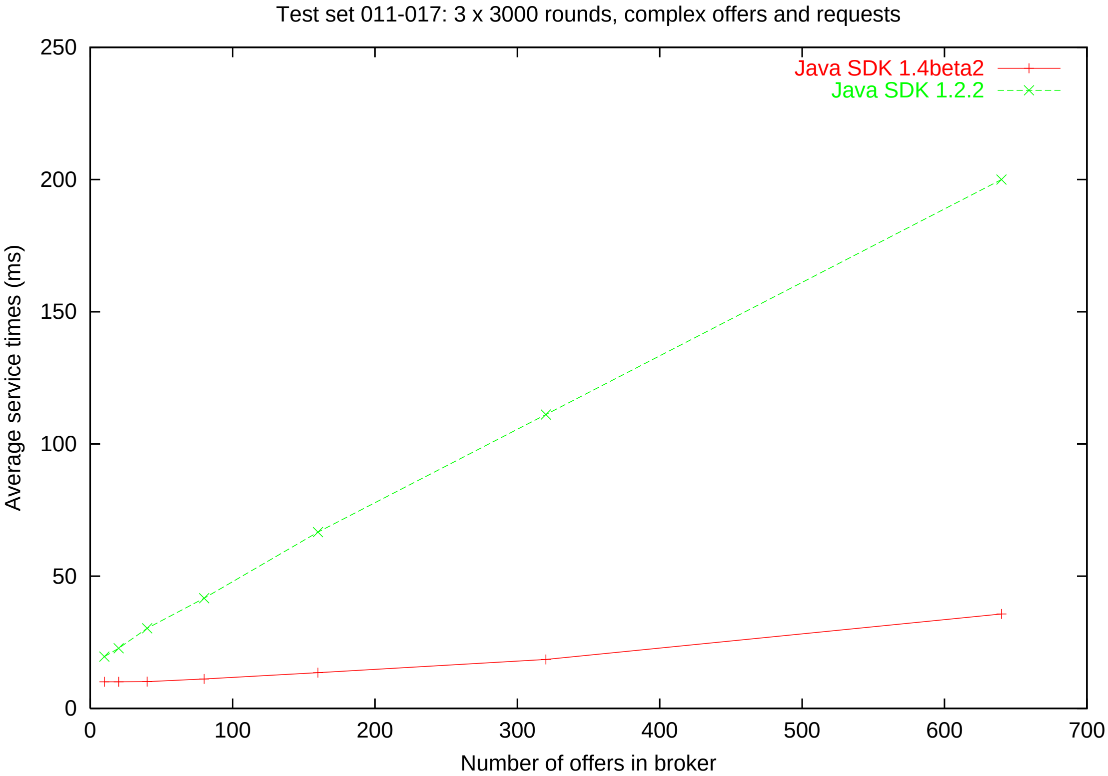

Figure 4.2: Broker performance with two Java virtual machines.

A more detailed analysis of individual requests reveals that compilation does not cause large spikes in service times, because it is done in the background. Further experiments indicated that the HotSpot VM compiles code based on relative frequency of use; the time between requests does not affect results.

The benchmarked system is especially amenable to adaptive compilation and optimisation, because the same operations (code paths) are executed repeatedly. In a larger application, the difference would probably not be as great.

Experimental benchmarks were also run using the Server version of the HotSpot VM that does more aggressive optimisation, but performance was lower than with the HotSpot Client VM. This somewhat surprising result is probably attributable to differences in memory management between the two versions.

Other benchmarks were run with an interpreting virtual machine because of the difficulties that the adaptive behaviour presents to obtaining consistent and relevant results.

4.1.6 Summary of results

The measurements showed that the response times of current Pilarcos broker increase linearly when the number of stored service offers increases. The number of different requests has no effect on broker performance, which indicates that the underlying ORBacus trader does not perform result caching. Increasing complexity of policy statements in offers and requests only increases the service times by a constant factor which is, of course, relative

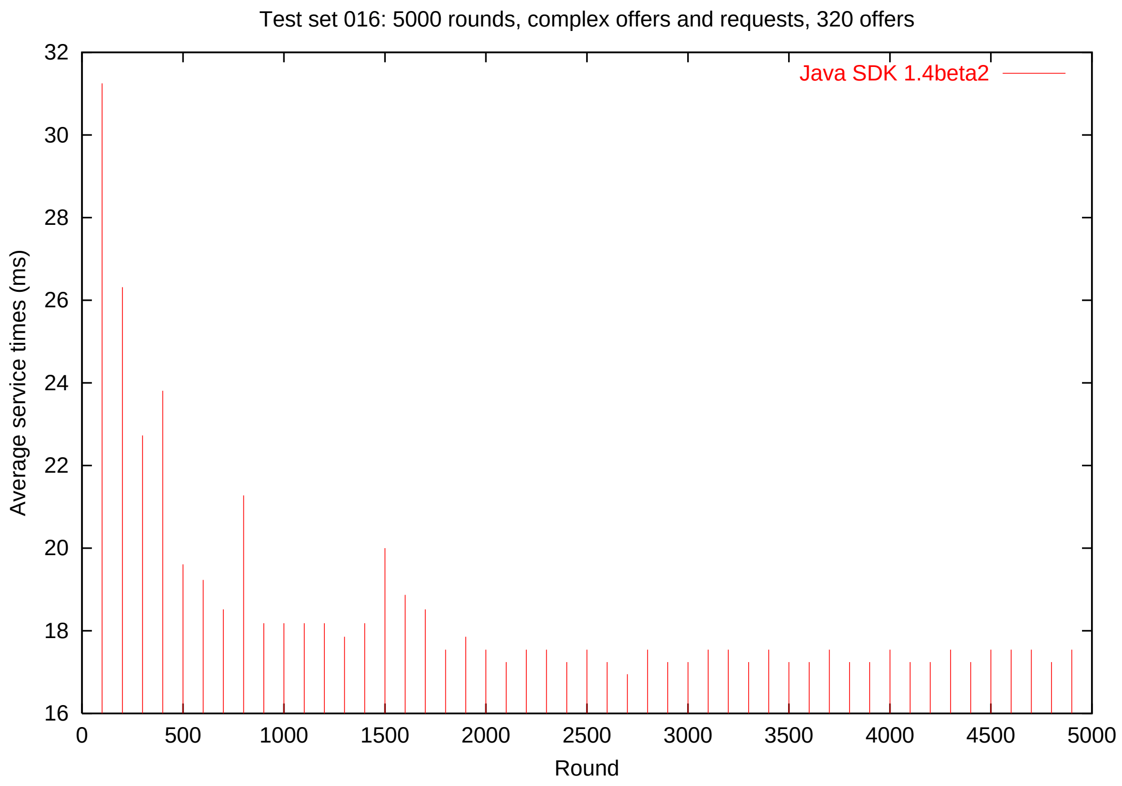

Figure 4.3: Service time decrease due to incremental compilation.

to changes in complexity.

The most astonishing results were received when two Java versions were compared. The newer 1.4.0beta2 version was by far more effective than the initially used 1.2.2 version. The new features of the Java virtual machine, including adaptive compilation of bytecode and enhanced memory handling, show remarkable increase in Java performance.

With reasonable numbers of service offers the broker service times stay below 100 milliseconds, well below 50 milliseconds when the new JVM is used – even with several hundreds of offers.

4.2 Whole system performance

4.2.1 Measurement scenario

The behavioural pattern of the measurement system simulates Tourist Info Service (see Chapter 2) usage in a largish city. The main characteristics of the baseline case are:

- The city has 10 tourist info services.
- 100 000 clients visit the services per hour.
- Each client session lasts for 10 minutes and includes 10 “queries”.

From these parameters it follows that

- Each tourist info service has 10 000 clients per hour.
- Each tourist info receives 100 000 “queries” every hour.

Further, each “query” was specified to require a fixed amount of processing time (initially 20 ms) to simulate service usage and wait time (200 ms) to simulate time spent waiting for subservices. In addition, each “query” returned 1024 bytes of result data to the client. Based on earlier measurements with broker (Section 4.1) each broker query was assumed to consume 50 milliseconds time.

4.2.2 Measurement environment

Because of practical limitations, the measurements were made with one tourist info service, one payment service, and suitable number of background client threads to simulate the behaviour of the scenario. A separate measurement client was used for gathering performance metrics as perceived by a client. To simplify the interpretation of results the client arrival rate and client “query” interval were kept constant using a timing mechanism.

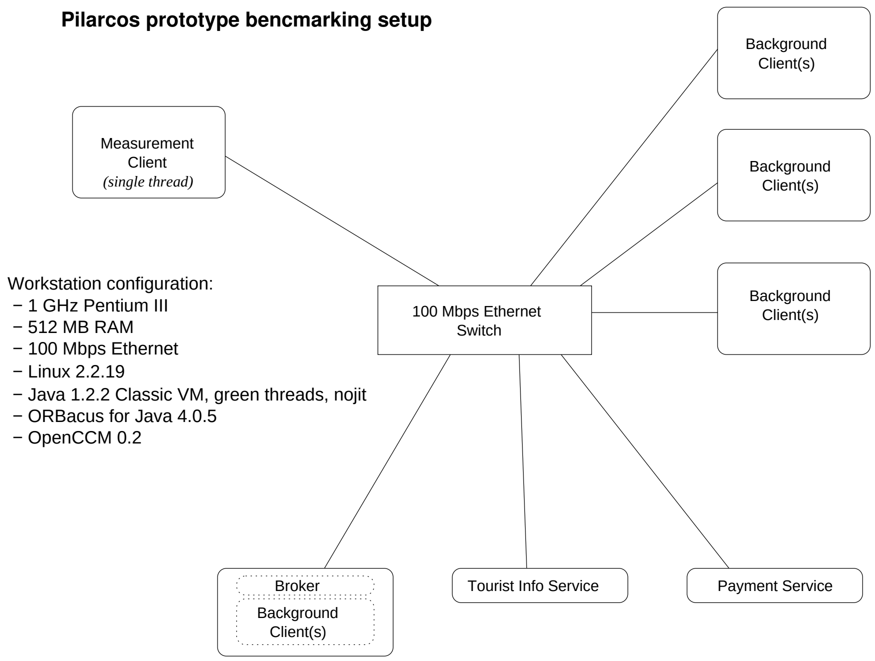

Figure 4.4: Prototype measurement environment.

Measurements were performed using 7 identical workstations attached to an isolated network segment without any disturbing external traffic. Each workstation has

- Linux 2.2.19
- Pentium III, 1 GHz, 512MB RAM

- 100Mbps Ethernet connection

The Java virtual machine used in measurements was version 1.4.0-beta2 running in interpreted mode with native threads.

To enable the running of large number of concurrent clients threads, the ORB parameters were set to

```
ooc.orb.conc_model=threaded
ooc.orb.oa.conc_model=thread_per_request
```

also at the client side. This instructs ORBacus to spawn a new thread (in the ORB) for each CORBA request, thus considerably decreasing wait times.

Figure 4.4 shows the constellation of services. Tourist info service and payment service were running on dedicated hosts. Broker host, which had negligible amount of actual service usage as broker queries were disabled, was also used to run background clients.

During initial measurements it turned out that the Java virtual machine threading mechanism was unable to keep up the pace when running with intended number of concurrent threads on a background client. To circumvent this, the background clients were made to run less threads with respectively reduced session times. Furthermore, the Java version in use was changed from 1.2.2 to 1.4.0-beta2 due to noticeably better thread handling in the later version. By this means the client arrival rate on the servers as well as the overall workload was kept at requested level.

Another limitation comes from Linux kernel parameters: the test system configuration prevented spawning more than about 300 client threads per host when running Java in native threads mode. This could easily be fixed by recompiling the kernel.

4.2.3 Client behaviour

The measurements were committed using special client program that simulates the behaviour of a client as specified in Section 4.2.1. A single client session consists of:

- Federation establishment (binding).
- Execution of 10 service requests at fixed intervals.
- Termination of federation.

The measurement client executes one client session during which beginning and end times of each operation to be monitored are logged.

To generate the required background load on the system a set of background clients was used. A background client operates in loop where it runs the specified number of client threads concurrently. The background client starts by spawning, at intervals that correspond to requested client arrival rate, the specified number of client threads. Each client thread then runs consecutive client sessions in a loop. Thus, from server components view the clients arrive at (approximately) uniform intervals and the number of simultaneous clients soon stabilises to a constant level.

Before the actual measurement client operation starts the system is allowed to warm up running only background clients for time that is approximately 1.1 times the duration of a client session.

4.2.4 Results: effect of the number of simultaneous clients

In the first test set the system response times were measured with respect to varying number of clients. The background client systems were unable to handle as many client threads as initially planned, so the number of simultaneous clients was divided by 2 and the number of "queries" per hour was multiplied by two (effectively halving the session times) to keep the number of threads manageable to Java. Hence, the following client/session combinations were used:

- 556 clients with 5 minutes session time (to simulate 1112 clients with 10 minutes sessions).
- 836 clients with 5 minutes session time (to simulate 1672 clients with 10 minute sessions, the baseline case).
- 836 clients with 3.3 minutes session time (to roughly simulate client increase by 50%).
- 836 clients with 2.5 minutes session time (to roughly simulate client increase by 100%).

Results presented here are mean values of three consecutive test runs.

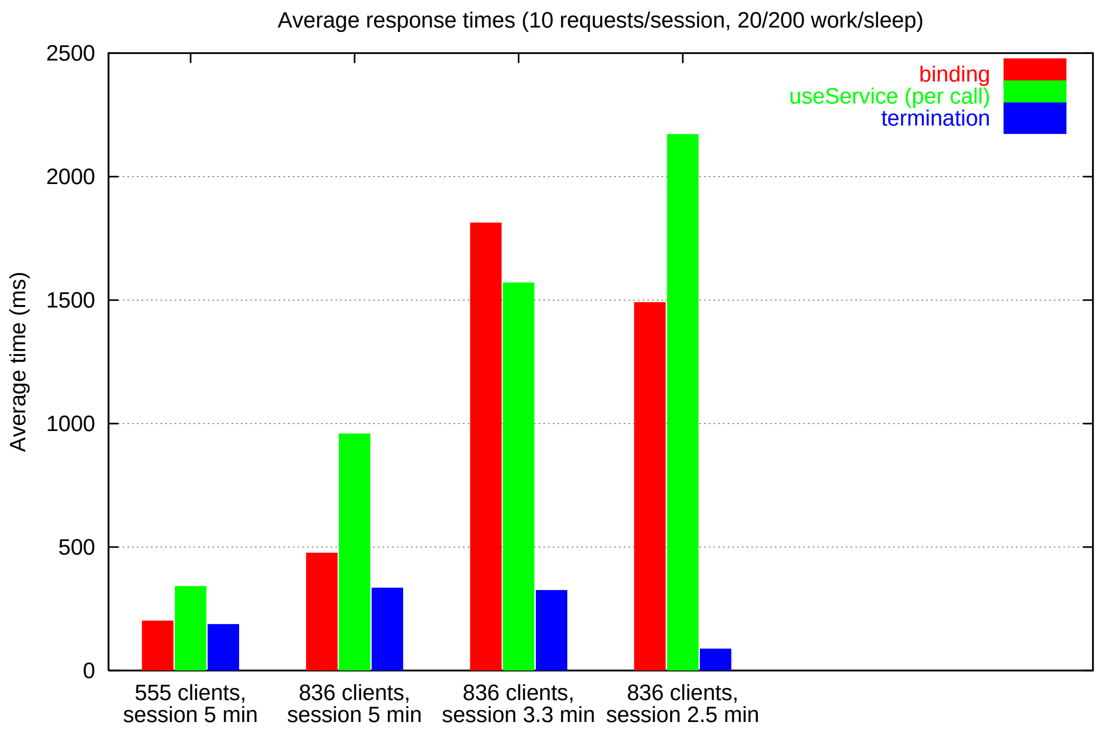

Figure 4.5: Response times vs. number of clients.

Figure 4.5 shows the mean response times when:

- Establishing federated bindings.

- Using service (per “query”).
- Terminating federation.

As can be seen in the figure, response times grow linearly when the number of simultaneous clients increases. The service response times stay at acceptable levels with 836 clients and 5 minutes session time, which corresponds to the initial scenario.

It is worth noting, that time consumed in federation establishment is rather minimal compared to actual service usage. In Figure 4.5 the columns indicating service usage are per call and the measurement scenario includes 10 service calls per session. Each session includes only one federation establishment and termination which in total consume just 5% of the total service time.

The apparent decrease in response times with the highest load is due to the fact that the Java virtual machine in background client hosts is unable to handle the specified number of client threads with short session times. This leads to a situation where background load is lower than expected, which gives better response times to the measurement client.

The same behaviour can be seen in Figure 4.6 which shows the average CPU load in different nodes during measurement client operation.

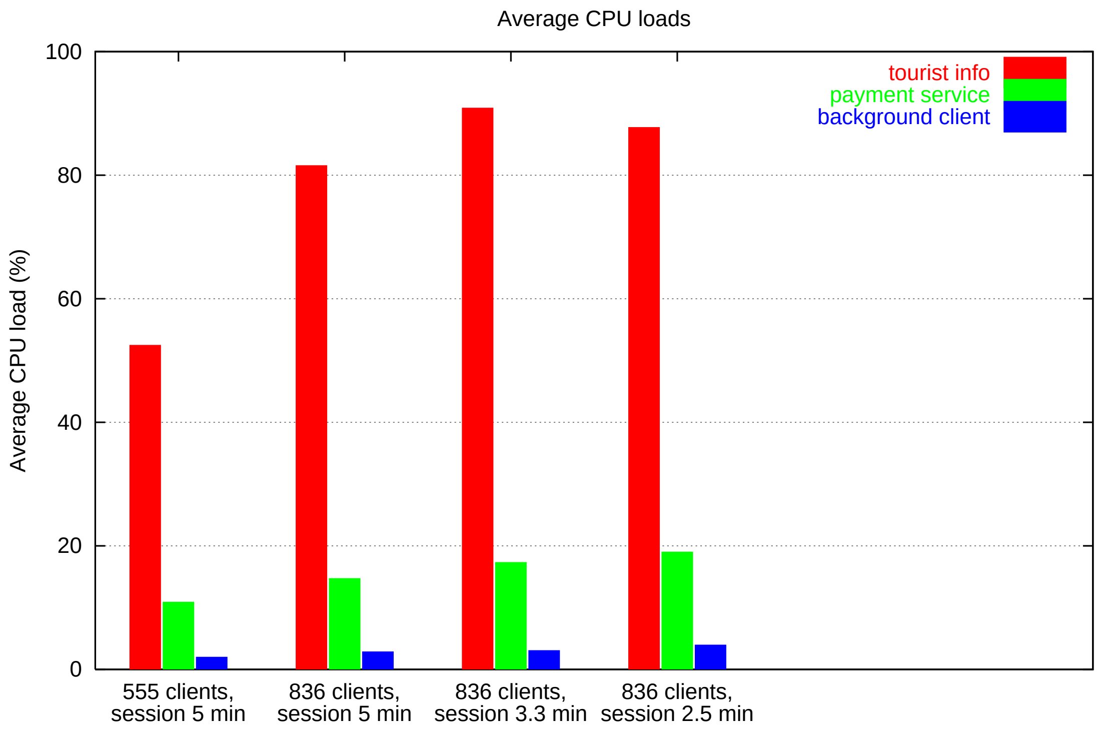

Figure 4.6: CPU load vs. number of clients.

With 836 simultaneous clients and 3.3 minutes session time (50% increase compared to the baseline case) the tourist info service CPU load goes over 90% which clearly indicates that tourist info service is becoming a bottleneck. This corresponds to over 20 000 clients arriving per hour, in another words to 5.57 new clients per second.

4.2.5 Results: effect of service workload

The second test set consists of simulations where the amount of processing time required per service request was varied. Other parameters were the same as in the baseline case with 836 clients and 5 minutes session time. Again results are presented as mean values of three test runs.

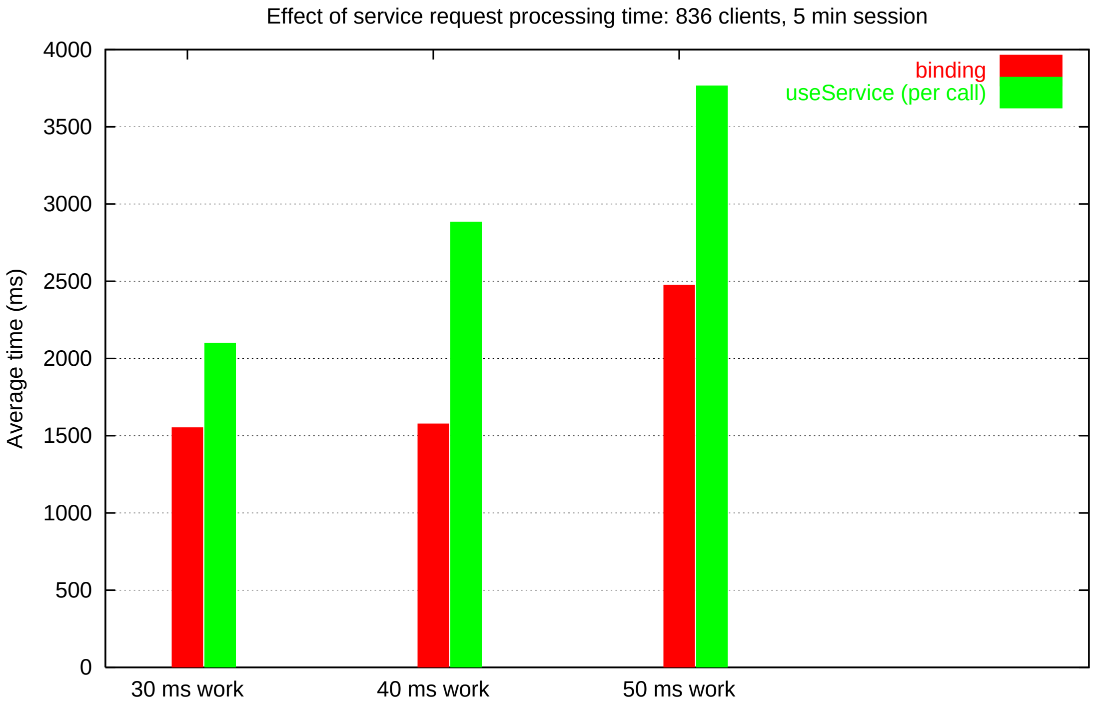

Figure 4.7: Response times vs. tourist info processing time.

It can be seen from Figure 4.7 that the overall response behaves linearly when the amount of work performed at tourist info service increases, as expected.

4.2.6 Results: effect of federation method

The third test set compares response times when using the two different versions of the federation protocol: iterative method and concurrent method.

Figure 4.8 shows that the proposed concurrent federation protocol reduces binding time by approximately 30% and, due to simultaneous decrease in overall load, also service response time by 10%.

4.2.7 Summary of results

The measurements show that the current Pilarcos middleware can easily handle the load specified in the measurement scenario in Section 4.2.1. The additional load from federation establishment and termination is only 5% of the time used in actual service.

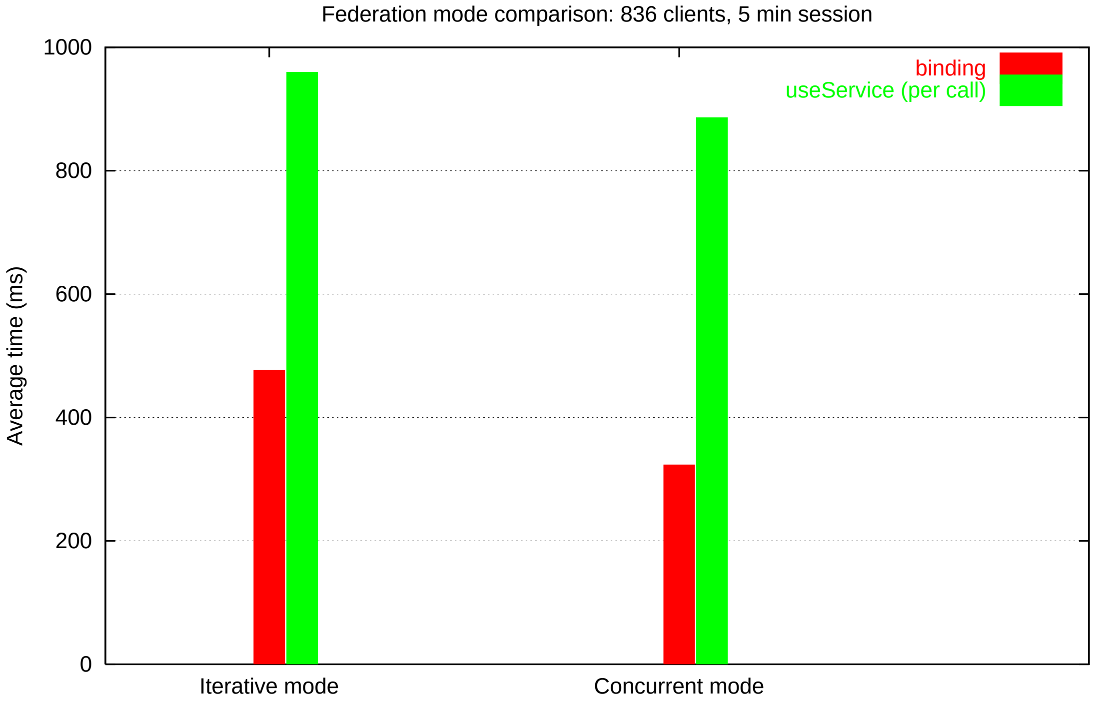

Figure 4.8: Comparison of federation methods.

Before extrapolating the results it is best to remember, however, that Pilarcos middleware is designed to remain active during service usage e.g. guarding the behaviour of the federation. This functionality is absent in the current prototype and will without question affect the performance of a real system.

The measurement results become less reliable when the session time is decreased from the baseline case. Indeed, as will be seen from the calculations in Section 4.3.2, the baseline case happens to be such that the number of simultaneous clients could not be much higher without significant decrease in performance.

4.3 Performance analysis

4.3.1 Business service broker performance

We calculate throughput of the business service broker in some example cases using an open single-server queue model.

Terms used in calculations are:	
arrival frequency of clients	X
service time	S
server utilisation rate	U
response time	R

All terms represent average values. From the broker benchmarks in section 4.1, we can estimate the service time of an import request to be 20-50 ms in this case.

From the queue model we have

$$R = S/(1-U),$$

and further

$$U = 1 - S/R.$$

Arrival frequency of clients (import requests) can now be calculated as

$$X = U/S.$$

We estimate response time requirements for the broker to be in the 100-1000 ms range in the present case. Results for a few example cases are given below.

S (ms)	R (ms)	X (queries/s)
50	500	18
30	300	30
30	100	23

From these results, we can see that with these assumptions made the business service broker cannot be expected to serve more than about 30 queries per second.

4.3.2 Analysis of tourist information service case

For analysing the tourist information server scalability, a more complex model is used. Clients make several consecutive queries to the tourist information server, which in addition to its own processing waits for subservice queries. This is the model that was used in the benchmarks.

A simple model

Federation establishment and termination is not included in the initial calculations. Similar to above, the following terms are used in calculations:

tourist info (maximum acceptable) response time	R_{ts}
tourist info processing time per query	S_{ts}
tourist info server utilisation rate	U_{ts}
client arrival rate on tourist info server	X
query rate on tourist info server	X_{ts}

Moreover, the following additional terms are used:

client think time	Z
number of queries within session	n
client session time	$T = n \times (Z + R_{ts})$
tourist info subservice wait time	D_{ss}

As in Section 4.2, it is assumed that each client session lasts for 10 minutes during which the client executes 10 queries (1 query/minute). Thus, the following initial values will be used:

$$\begin{aligned} R_{ts} &= 1 \text{ s} \\ Z &= 59 \text{ s} \\ n &= 10 \\ T &= 10 \text{ min} \end{aligned}$$

Each query requires 20 milliseconds processing and 200 milliseconds wait for subservices on the tourist info server, so:

$$\begin{aligned} S_{ts} &= 0.02 \text{ s} \\ D_{ss} &= 0.2 \text{ s.} \end{aligned}$$

Let T_{ts} denote the query execution time on tourist info server. The tourist info response time consists of prosessing and wait for subservices:

$$R_{ts} = T_{ts} + D_{ss}.$$

From the selected inital values follows now that:

$$T_{ts} = 0.8 \text{ s.}$$

On the other hand, because

$$T_{ts} = S_{ts}/(1 - U_{ts}),$$

the maximum tourist info server utilisation rate that allows the requested response time is

$$U_{ts} = 1 - S_{ts}/T_{ts} = 0.975.$$

The maximum sustainable query rate – assuming that the use of subservices does not include queueing – is then:

$$X_{ts} = U_{ts}/S_{ts} = 0.975/0.02s = 48.75 \text{ 1/s.}$$

The response time is also related to the number of simultaneous clients by $R_{ts} = N/X_{ts} - Z$. From this it follows for (maximum) number of simultaneous clients that

$$N = X_{ts} \times (R_{ts} + Z) = 48.75 \times 60 \approx 2900$$

In other words, if the tourist info service has at most 2900 simultaneous clients, response time remains below 1 second.

With these values there are on average

$$N_{ts} = X_{ts} \times R_{ts} \approx 49$$

concurrent queries on the server.

Having, on average, 2900 simultaneous clients sessions that last for 10 minutes the tourist info service receives $X = N/T = 2900/600 \approx 4.8$ new clients per second corresponding to 17 000 new clients per hour.

The measurements in Section 4.2 indicate, however, that the 1 second response time is already reached with approximately 1700 clients. To analyse the divergence it is necessary to consider differences in assumptions. First, the calculations above do not take into account the processing time consumed in federation establishment and termination. Second, the use of payment service was not considered in calculations.

A slightly refined model

From the measurement results it can be approximated that the federation operations require together about 21 milliseconds processing time. Amortising this evenly to queries will add roughly 2 milliseconds processing time to each query resulting in 22 milliseconds. In addition, each payment operation requires 10 milliseconds processing time and 10 milliseconds sleep time. In order to avoid making the model too complex, lets estimate this by adding a 30 milliseconds wait to each query, noting that the measurements showed that in this scenario the payment service has a very moderate load.

Repeating the calculations with the above mentioned corrections gives the following results for different response time requirements:

R_{ts} (ms)	X_{ts} (queries/s)	N
300	33.3	2000
500	43.9	2634
1000	46.3	2779
1500	46.8	2810
2000	47.1	2823
2500	47.2	2830

Thus, the tourist info server should be able to handle about 2800 simultaneous clients with 1 second response time. The deviation between calculations and measured values is still about 30% in the baseline case. Assuming that the actual processing time is 33 milliseconds per query results in quite accurate approximations, which seems to indicate that there is quite substantial overhead in thread handling and context switching when running a large number of threads.

4.3.3 Conclusion

The usage scenario presented in Section 4.2.1 defined the number of competing tourist info services to be 10. With 10 tourist info services and service usage pattern defined in the same scenario the broker may receive as many as 50 queries per second in contrast to the calculated maximum of 30 queries per second. It is thus evident that in a real life situation – where the broker most probably mediates other types of services in addition to tourist info services – the broker will become a bottleneck, unless replicated or federated.

When analysing the tourist info service behaviour in the usage scenario of Section 4.2.1 by simple queue modelling techniques, it seems that the service could handle nearly 3000 simultaneous clients. The measurements showed, however, degradation of performance with somewhat smaller number of concurrent clients. Proper understanding of this issue would require further research, but at this time the most probable cause lies in the cobehaviour of the Java virtual machine and the underlying operating system, more specifically in thread handling.

Chapter 5

Conclusion

The Pilarcos project has developed a new model for large distributed applications that is based on separating architecture descriptions from application components. The model extends the traditional client-server paradigm to multi-party relationships, and enables federated systems to be formed of autonomous services on the basis of a contract negotiated between them.

Prototype implementations of the central Pilarcos middleware services have been constructed on the OpenCCM platform. Of them, the business service broker provides support for discovering combinations of interoperable services. Application binders handle management of federations and communication between peers in the federated system, hiding the complexity from application programmers.

To assess the feasibility of the Pilarcos model, benchmarks were conducted using the prototype implementation. The tourist information service was used as the example case. Results indicated that the Pilarcos middleware services do not carry a significant overhead in this type of application. Some caution should be exercised when interpreting the results, however, because the prototype is incomplete. The business service broker may become a bottleneck, and should therefore be replicated or federated. In addition, interesting results on Java virtual machines were obtained, showing a marked increase in performance from interpreted Java 1.2.2 to adaptively compiled Java 1.4.

The prototype is currently in a proof of concept state, and not all Pilarcos middleware services have been implemented yet. A separate policy repository is needed for more flexible management of policies within each autonomous organisation. Interoperability features, including bridges and proxies for communication between technically differing domains need to be developed. For real-world use, tools for application programmers would also have to be built.

In light of the benchmark results, the Pilarcos architecture can be considered feasible but for very time-critical systems. The prototype provides a good starting point for further development of both the Pilarcos concepts and the implementation.

Bibliography

[1] ALLEN, R. J. *A Formal Approach to Software Architecture.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[2] HAATAJA, J., SILFVER, E., VÄHÄAHO, M., AND KUTVONEN, L. CORBA Component Model – status and experiences. Tech. rep., Dec. 2001. C-2001-65.

[3] KUTVONEN, L., HAATAJA, J., SILFVER, E., AND VÄHÄAHO, M. Pilarcos architecture. Tech. rep., Mar. 2001. C-2001-10.

[4] OBJECT MANAGEMENT GROUP. *OMG Trading Object Service Specification*, June 2000. OMG Document formal/2000-06-27. Also http://www.omg.org/cgi-bin/doc?formal/2000-06-27.

[5] OBJECT MANAGEMENT GROUP. *OMG Naming Service Specification*, Feb. 2001. OMG Document No. formal/2001-02-65. Also http://www.omg.org/cgi-bin/doc?formal/2001-02-65.

[6] SUN MICROSYSTEMS. The Java HotSpot Virtual Machine. White paper. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html, 2001.

Appendix A. Pilarcos prototype IDL3 definition

```
module Prototype {

    typedef sequence<octet> OctetSeq;
    typedef sequence<string> StringArray;

    //
    // ---------- Policy definitions ----------
    //

    struct PolicyType {
        string name;            // "Price"
        string type;            // "double"
    };

    typedef sequence<PolicyType> PolicyFramework;
    typedef sequence<string> PolicyFrameworkNames;

    struct Policy {
        string name;            // "Price"
        string constraint;      // "Price < 10"
    };

    typedef sequence<Policy> Policies;

    //
    // ---------- Interface definitions ----------
    //

    struct InterfaceType {
        string name;            // "TouristInfoInterface"
        string bindingModel;    // "provides" (CCM)
    };

    struct InterfaceDef {
        string name;            // "TouristInfoInterface"
        string technicalDef;    // reference to IDL definition
    };

    //
    // ---------- Service type related definitions ----------
    //

    struct ServiceDef {
        string name;            // "TouristInfoService"
        sequence<InterfaceType> interfaceTypes;
    };

    struct ServiceType {
        string name;            // "TouristInfoService(Type?)"
        sequence<ServiceDef> providedServices;  // 0 for request, 1 for offer
        sequence<ServiceDef> requiredServices;  // 0..*
        PolicyFramework policyFramework;
```

```
};

interface Deployment;

struct ServiceOffer {
    string serviceTypeName;
    Policies policies;
    sequence<InterfaceDef> interfaceDefs;
    Deployment deploymentInterface;
};

struct ServiceRequest {
    string serviceTypeName;
    Policies policies;
    sequence<InterfaceDef> interfaceDefs;
    string businessArchName;
    string roleName;
};

//
// ---------- Business architecture definitions ----------
//

struct Role {
    string name;
    sequence<string> serviceTypeNames;
    Policies policies;              // framework from service types
};

struct PolicyBinding {
    string requirerPolicyName;
    string providerPolicyName;
};

struct RoleBinding {
    string requiringRoleName;
    string requiringServiceTypeName;
    string providingRoleName;
    string providingServiceTypeName;
    sequence<PolicyBinding> policyBindings;
    Policies bindingPolicies;       // framework: generic
};

struct BusinessArchitecture {
    string name;
    sequence<Role> roles;
    sequence<RoleBinding> roleBindings;
    Policies populationPolicies;    // framework: generic
};

//
// ---------- Offer and binding contract definitions ----------
//
```

```
struct OfferForRole {
    string roleName;
    ServiceOffer offer;
};

struct ServiceContract {
    string id;
    ServiceRequest servReq;
    sequence<OfferForRole> offers;
};

typedef sequence<ServiceContract> ServiceContractOffers;

struct InterfaceRef {
    InterfaceDef interfaceDef;
    any reference;                  // reference to providing object
};

struct BindingContract {
    string contractId;
    string servContId;
    Policies policies;
    sequence<InterfaceRef> interfaceRefs;
};

//
// ---------- Infrastructure interface definitions ----------
//

interface TypeRegistration {
    void addServiceType(in ServiceType serviceType);
    void removeServiceType(in string serviceTypeName);

    void addBusinessArch(in BusinessArchitecture businessArch);
    void removeBusinessArch(in string businessArchName);

    void addInterceptor(in InterfaceDef iDef1, in InterfaceDef iDef2,
        in string interceptorId);
    void removeInterceptor(in string interceptorId);
};

interface TypeLookup {
    ServiceType describeServiceType(in string servName);
    BusinessArchitecture describeBusinessArch(in string businessArchName);

    boolean isCompatible(in InterfaceDef iDef1, in InterfaceDef iDef2);
    string getInterceptorId(in InterfaceDef iDef1, in InterfaceDef iDef2);
};

interface Brokering {
    void exportOffer(in ServiceOffer servOffer);
    ServiceContractOffers importOffers(in ServiceRequest servReq);
};
```

```
interface Negotiation {
    boolean negotiate(inout BindingContract bindContr);
    void terminateContract(in string servContId);
};

interface Deployment {
    Negotiation prepareService(in ServiceContract servCont);
};

interface ServiceConfiguration {
    void initiateServicePreparation(in ServiceContract servCont);
};

interface PolicyConfiguration {
    void setPolicyFramework(in string frameworkName,
        in PolicyFramework framework);
    void removePolicyFramework(in string frameworkName);
    PolicyFrameworkNames getPolicyFrameworkNames();
    PolicyFramework getPolicyFramework(in string frameworkName);
    void configure(in string frameworkName, in Policies policies);
    string query(in string frameworkName, in string policyName);
};

interface Binding {
    string importSingleOffer(in string businessArchName); // servContId
    ServiceContractOffers importOffers(in string businessArchName);

    ServiceContract describeOffer(in string servContId);
    void bind(in string servContId);
    void unbind(in string servContId);

    // for benchmarking only
    void duplicateServiceContract(in string fromServContId,
        in string toServContId);
};

interface FederationManagement {
    void contractTerminated(in string servContId);
};

//
// ---------- Infrastructure component definitions ----------
//

component TypeRepository {
    provides TypeRegistration typeRegistration;
    provides TypeLookup typeLookup;
    uses CosTrading::TraderComponents trader;

    attribute long debugLevel;
};
home TypeRepositoryHome manages TypeRepository {
};
```

```
component BsBroker {
    provides Brokering brokering;
    uses TypeLookup typeLookup;
    uses CosTrading::TraderComponents trader;

    attribute long debugLevel;
};
home BsBrokerHome manages BsBroker {
};

//
// ---------- TouristInfo interface definitions ----------
//

struct Bill {
    string biller;
    double amount;
    string description;
};

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

interface Payment {
    Bill getBill(in string billId, in string servContId);
    void payBill(in string billId, in string account,
        in string certificate, in string servContId);
    void refuseBill(in string billId, in string servContId);
};

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

interface User {
    void run();
    void benchmark(in long concurrentClients,
        in long queries, in long queriesPerHour,
        in long rounds, in long brokerQueryMs);
};

//
// ---------- TouristInfo component definitions ----------
//

component ClientApplication supports User {
    uses TouristInfo touristInfo;
    uses Payment payment;
```

```
    uses PolicyConfiguration policyConf;
    uses Binding binding;
    provides FederationManagement federationManagement;

    attribute long debugLevel;
};
home ClientApplicationHome manages ClientApplication {
};

component ClientAB {
    provides TouristInfo touristInfo;
    provides Payment payment;

    provides PolicyConfiguration policyConf;
    provides Binding binding;
    uses FederationManagement federationManagement;

    provides ServiceConfiguration serviceConf;
    provides Negotiation negotiation;

    uses Brokering brokering;
    uses TypeLookup typeLookup;
    uses TypeRegistration typeRegistration;

    attribute long debugLevel;
    attribute boolean concurrencyModel;
};
home ClientABHome manages ClientAB {
};

component PaymentServer {
    provides Payment payment;
    provides Billing billing;

    uses PolicyConfiguration policyConf;
    provides FederationManagement federationManagement;

    attribute long workMillis;
    attribute long sleepMillis;
    attribute long debugLevel;
};
home PaymentServerHome manages PaymentServer {
};

component PaymentServerAB {
    uses Payment payment;
    provides Payment federatedPayment;
    uses Billing billing;
    provides Billing federatedBilling;

    provides PolicyConfiguration policyConf;
    uses FederationManagement federationManagement;
```

```
    provides ServiceConfiguration serviceConf;
    provides Negotiation negotiation;

    uses Brokering brokering;
    uses TypeLookup typeLookup;

    attribute long debugLevel;
    attribute boolean concurrencyModel;
};
home PaymentServerABHome manages PaymentServerAB {
};

component PaymentDeployer {
    provides Deployment deployment;

    uses Brokering brokering;
    uses TypeLookup typeLookup;
    uses TypeRegistration typeRegistration;
    uses CosNaming::NamingContext namingContext;

    attribute long workMillis;
    attribute long sleepMillis;
    attribute long debugLevel;
    attribute boolean concurrencyModel;
    attribute string componentServerName;
};
home PaymentDeployerHome manages PaymentDeployer {
};

component TouristInfoServer {
    uses Billing billing;
    provides TouristInfo touristInfo;

    uses PolicyConfiguration policyConf;
    provides FederationManagement federationManagement;

    attribute long workMillis;
    attribute long sleepMillis;
    attribute long debugLevel;
};
home TouristInfoServerHome manages TouristInfoServer {
};

component TouristInfoAB {
    provides Billing billing;
    uses TouristInfo touristInfo;
    provides TouristInfo federatedTouristInfo;

    provides PolicyConfiguration policyConf;
    uses FederationManagement federationManagement;

    provides ServiceConfiguration serviceConf;
    provides Negotiation negotiation;
```

```
        uses Brokering brokering;
        uses TypeLookup typeLookup;

        attribute long debugLevel;
        attribute boolean concurrencyModel;
    };
    home TouristInfoABHome manages TouristInfoAB {
    };

    component TouristInfoDeployer {
        provides Deployment deployment;

        uses Brokering brokering;
        uses TypeLookup typeLookup;
        uses TypeRegistration typeRegistration;
        uses CosNaming::NamingContext namingContext;

        attribute long workMillis;
        attribute long sleepMillis;
        attribute long debugLevel;
        attribute boolean concurrencyModel;
        attribute string componentServerName;
    };
    home TouristInfoDeployerHome manages TouristInfoDeployer {
    };
};
```

Appendix B. Type definitions from Tourist Info case

```
#
# Business architecture definition
#

business architecture TouristInfoArch {
    role Client {
        service type: TouristInfoUser
    }

    role Server {
        service type: TouristInfoService
    }

    role PaymentSystem {
        service type: PaymentService
    }

    binding Client.TouristInfoUser - Server.TouristInfoService {
        policy bindings {
            Area = Area
            Price = Price
            Needed_subservices = Available_subservices
            Payment_methods = Payment_methods
            Terminal_type = Client_terminal_type
        }
    }

    binding Client.TouristInfoUser - PaymentSystem.PaymentService {
        policy bindings {
            Payment_methods = Payment_methods
        }
    }

    binding Server.TouristInfoService - PaymentSystem.PaymentService {
        policy bindings {
            Payment_methods = Payment_methods
        }
    }
}

#
# Service type definitions
#

service type TouristInfoService {
    provides TouristInfoService {
        TouristInfoInterface: provides
    }

    requires PaymentService {
        BillingInterface: uses
    }
```

```
    policy framework {
        Area: string
        Price: double
        Available_subservices: set
        Payment_methods: set
        Client_terminal_type: set
    }
}

service type PaymentService {
    provides PaymentService {
        BillingInterface: provides
        PaymentInterface: provides
    }

    policy framework {
        Payment_methods: set
    }
}

service type TouristInfoUser {
    requires TouristInfoService {
        TouristInfoInterface: uses
    }

    requires PaymentService {
        PaymentInterface: uses
    }

    policy framework {
        Area: string
        Price: double
        Needed_subservices: set
        Payment_methods: set
        Terminal_type: set
    }
}
```

Appendix C. Example of Pilarcos client code

```
public void run() {
    ///// Policy configuration /////

    PolicyConfiguration policyConf = get_connection_policyConf();
    Policy[] policies = new Policy[] {
        new Policy("Area", "Area == 'Paris'"),
        new Policy("Price", "Price < 10") };

    policyConf.configure("TouristInfoArch", policies);

    ///// Federation establishment /////

    Binding binding = get_connection_binding();

    String servContId = binding.importSingleOffer("TouristInfoArch");
    binding.bind(servContId);

    ///// Service usage /////

    TouristInfo touristInfo = get_connection_touristInfo();

    StringHolder billId = new StringHolder();
    touristInfo.requestService(billId, servContId);

    Payment payment = get_connection_payment();
    Bill bill = payment.getBill(billId.value, servContId);

    // here, should check if Bill is OK
    // ...

    // pay bill
    payment.payBill(billId.value, "<AccountNumber>",
                    "<Certificate>", servContId);

    byte result[] = touristInfo.useService("<Parameters>", servContId);

    ///// Federation tear-down /////

    binding.unbind(servContId);
}
```