Department of Computer Science
Series of Publications C
Report C-2001-nn

Pilarcos architecture

Lea Kutvonen, Juha Haataja, Egil Silfver, Markku Vähäaho

University of Helsinki
Finland

Department of Computer Science
Series of Publications C
Report C-2001-nn

Pilarcos architecture

Lea Kutvonen, Juha Haataja, Egil Silfver, Markku Vähäaho

University of Helsinki
Finland

Contact information

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

Email address: Lea.Kutvonen@cs.Helsinki.FI (Internet)

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

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

Computing Reviews (1998) Classification: C.2.4., D.2.0
Helsinki 2001

Pilarcos architecture

Lea Kutvonen, Juha Haataja, Egil Silfver, Markku Vähäaho

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

Technical report, Series of Publications C, Report C-2001-nn
Helsinki, March 2001, vi + 66 pages

Abstract

The PILARCOS project develops middleware solutions for automatic management of interorganisational applications. The project works at three fronts:

- architecture development for a federated environment where applications are constructed based on architecture descriptions populated by traded components;
- prototyping in CORBA some essential middleware services required for global negotiation of federations and instantiating architectures; some of this work contributes to the further development of common CORBA services; and
- designing an ODP viewpoint based software engineering process and tool that interfaces with the middleware repositories of the runtime environment.

This report describes the middleware architecture supporting interorganisational applications and their distributed management, and gives an overview of the related software engineering process.

The runtime support for applications and the software engineering process become intertwined through middleware repository services appearing in both environments.

Computing Reviews (1998) Categories and Subject Descriptors:
C.2.4 Computer-communication networks: Distributed Systems
D.2.0 Software engineering: General

General Terms:
Design, Experimentation, Standardisation

Additional Key Words and Phrases:
Software system architectures, federated systems, software engineering, component models

Acknowledgements

This report is based on work performed in Pilarcos project at Department of Computer Science at University of Helsinki. The Pilarcos project is funded by the National Technology Agency TEKES in Finland, together with Nokia, SysOpen and Tellabs. Plenty of the ideas behind this research are encouraged by discussions within and around the ODP standardisation work.

Helsinki, March 2001
Lea Kutvonen

Contents

Chapter 1

Introduction

The current trend in information technology is going towards global systems, inter-organisational cooperation, and markets of independently developed components for improved productivity in IT system development.

The variety of component based global systems are of interest here, as well as multiple aspects of them. For example, open virtual enterprises like

- virtual businesses;
- ad hoc groups between people in the large network environment formed by user request;
- ad hoc platform support for mobile computing units to reach their home units that reside at remote servers; and
- ad hoc platform support for ubiquitous computing units to services local to each visited physical environment

involve series of establishment and management problems. Further, software engineers can put together a new application from purchased components either in local software engineering projects or in projects that span multiple organisations. The interesting themes also include marketing aspects like outsourcing and leasing of business services between companies. Topical terms are ASP (application service provider) and peer-to-peer computing.

Following up this trend requires

- evolution of open service markets both for business services and for computational components; these markets should become an integral part of the global computing infrastructure. Achieving this requires some major development in terms of joint understanding on the reasonable structure of service markets and development of open distributing computing platforms to support these markets. Furthermore, the computational components should be consistent with a shared engineering architecture; the requirements of such architecture are discussed later in this paper.

- evolution of organisation wide management tools that allow business and computing policies to be administered in a consistent and efficient way; the management information must be inherited by all of the organisations applications and infrastructure services and also used for agreements on inter-organisational cooperation. The management tools require that the computing infrastructure and the applications are structured in an appropriate way.
- promotion and further evolution of global infrastructure services that support inter-organisational cooperation routines, like checking semantical and technological capabilities for cooperation, negotiating on decisions dependent on organisational policies, accessing business service markets, and selecting suitable computational components for a task.
- evolution of software architecture that binds together the requirements of service and component markets, infrastructure services and organisation-wide management needs.
- evolution of software engineering tools that produce components with the above mentioned architecture view and insert the components to the global markets.

The Pilarcos project works towards these goals by developing middleware solutions for automatic management of interorganisational applications. The name Pilarcos is an acronym for Production and integration of large component systems. The loose term "large system" is used for systems spanning multiple administrative domains, like organisations; the owners and thus goals of the systems involved differs but still they have mutual interest and are technically interoperable.

The project works at three fronts

- architecture development for a federated environment where applications are constructed based on architecture descriptions populated by traded components;
- prototyping in CORBA some essential middleware services required for global negotiation of federations and instantiating architectures; and
- designing an ODP viewpoint based software engineering process and tool that interfaces with the middleware repositories of the runtime environment.

The middleware architecture for a federated environment exploits three levels of abstraction. On the topmost level, the interorganisational applications are captured by business architecture descriptions spanning across organisational boundaries. The services provided by each participant are independently supported by each partner. The second level captures into a design description how the computing is actually organised. The design is populated by component assemblies that implement the service. The assemblies form the third level. The entities at this third level are technology and platform dependent whereas all other levels are technology independent. The topmost population process of the business architecture description manages dynamic federations between autonomous organisations with differing evolution cycles and policies; the design population process manages technology domain boundaries and efficient reuse of existing components.

The infrastructure ideology behind this project is that of ODP-RM (open distributed processing reference model) [10, 11, 12, 13, 17]. The middleware services exploited include traders, type repositories, federated binding mechanisms. Traders are used to mediate metainformation about available service providers in the global network, grouped by service type that indicates common basic behaviour of the services. Type repositories manage information about known service types and thus help traders in their tasks. Type and type relationship information is also used for bindings between objects in the global system, for runtime check of type conformance of partner's interfaces for communication. Federated binding mechanism produces a communication channel between objects in separate, autonomously administered domains. This requires interworking of the negotiation protocols that select suitable components at each domain for the communication.

Also, the software engineering tools to be tried out are based on the ODP viewpoint languages. The first design considers only tools for entering enterprise and computational viewpoints into the repositories and exploiting those descriptions within the runtime system. Special interest is focused on ODP enterprise language that gives some guidelines for the two architecture description levels, and the two population processes mentioned above.

Essential for Pilarcos architecture is that the runtime environment and the software engineering tools use shared middleware repositories. Thus, the two environments become intertwined and give strong support for the service and component marketplaces aspired.

The practical pilots to be used for evaluating the feasibility of the architecture and promoting the ideas are based on OMG technologies, especially CORBA [32], CCM (CORBA Component Model) [37], UML [34], and MOF [35].

This report describes the open federation framework in Section 2, the related software engineering process in Section 3 and explains how the framework mixes the design, deployment and exploitation epochs of a service life-cycle together in Section 2.9. These sections show how the Pilarcos framework supports federation (dynamic cooperation establishment between organisations), exploits component markets, and support automated and adaptive service deployment. As a more concrete representation, the infrastructure repositories and middleware services exploiting them are briefly reviewed in Section 4.

Chapter 2

Open system framework

The goal of the open system framework is to support automated deployment of business services and automated federation establishment governed by the policies of each involved sovereign system.

This section discusses the basic principles of the open system framework, in terms of portals for accessing business services, the deployment process of a business service, and the concepts of federation required for interorganisational cooperation.

The framework is based on the idea of separating out role based architecture descriptions and objects that can be plugged into the roles through a selection process with meta-information available on the objects.

The framework heavily builds on logical, widely understood types (predicates on objects) to negotiate contracts and map the contracts to real implementations separately and independently on each platform. Interoperation is guaranteed by the shared logical contract.

Here, *object* is a very scalable term in the ODP reference model sense, capturing anything ranging from a single variable to a full featured IT system. Interesting examples of objects are CORBA components and CORBA component assemblies.

2.1 Sovereign systems and federation

Federated systems are composed of sovereign systems that include middleware for interoperability support.

A system is considered to be sovereign when it has an autonomous administration. In a sovereign system, decisions – for example on system architecture, services supported, programming and interface definition languages, remuneration, authorisation policies, and communication protocols – can be done independently from any other systems.

Interoperability support includes mechanisms that negotiate about the shared capabilities of the systems and establish an interoperation relationship at system run-time. Such mechanisms are needed in a multi-organisational environment because the interacting objects cannot inherit properties that would ensure interoperability.

Cooperation ability between objects located to separate sovereign systems becomes separated to two concepts: interoperation and interworking. Interoperation between (application) objects means that the supporting infrastructure manages aspects arising from system sovereignty. In an interworking relationship the objects need to manage those aspects internally.

The characterising goal of federated systems is to enable world-wide information service systems, but still allow each member system to offer a localised computing environment for the end-users. A shared goal with the traditional distributed system evolution is the inclusion of new technology, like mobility, multi-media streams, and multi-party communication relationships.

We consider a federated system to be a community of federable systems that dynamically enter and leave federations.

Federation is a state of agreement between two or more systems about interoperation. Federations can be established and terminated during the operation of the systems. The federation agreement covers general communication related aspects, such as protocols and locations of interfaces, and service specific aspects, such as quality of service contracts.

A federable system is a sovereign system that contains middleware services that support federation establishment and management.

The goal of the federated system model is to improve the facilities for accessing services (not servers) from computing systems belonging to other organisations. Federation can be established between such systems that include similar facilities and that are allowed to federate by their owners. There is no predefined shared goal for the joint operation nor a shared control for the joint operation, but clients at any federable system can request services that are eventually performed at another federable system.

Composition of a world-wide system involves interconnection of the independent systems. The scope of potential federations is determined by the amount of common communication facilities and the amount of applications that are suitable for cooperation. A fully connected communication network between the federable systems is not necessary, as all services are not used at all federable systems and in some cases an intermediating system can be used. Furthermore, reachability of individual servers from all locations is not an objective. Therefore, practical limitations of interconnectivity do not invalidate the model. However, the communication services across the federated system should be able to join various communication technologies. For example, the communication services should be able to adapt the failure detection and recovery mechanisms to fixed and mobile networks.

In the world-wide federated system, it is not reasonable to expect that all organisations would offer a similar computing platform or application repertoire for their users. Instead, each organisation should be allowed to offer localised interfaces. We can adopt the concept of personalisation (from the context of micro-kernels [22, 41]) as a design pattern that separates the actual execution of services from the mechanism through which the users exploit the service. In the federated system, we allow transformation of generic service concepts to a local representation format at each member system. Such an architectural opportunity is beneficial for vendor competition as well: The vendors of federable system platforms and applications are granted a possibility of commercial competition with their products within the federated systems.

The federation management mechanisms are easier to build if federations are not established between whole systems but per individual services. Each service federation can be established independently and the federations can be reconstructed whenever changes at the services or service implementations are introduced. The service-wide federations require standardised contract schemata. Such work has already been initiated, for example within the business object modelling special interest group (BOMSIG) of OMG [30].

We introduce a federated system application architecture that contains sovereign objects establishing liaisons, i.e. contractual relationships, among themselves. The liaisons ensure that the interacting objects are technically able to exchange information and perform services for each other in a semantically consistent way. The technical capabilities are controlled using meta-information about the objects. The meta-information is maintained by platform level services.

Figure 2.1 illustrates the global system view where each service is essentially formed by application level objects and platform level objects. The properties of platform level objects effect essentially the interconnectivity potential of an application object, and thus platform objects must be included in the global system view. In distributed systems, the network of platforms is homogeneous and fully connected, and the global system view can be abstracted to contain only the application level objects. In federated system model, the interconnectivity and heterogeneity of the platforms must be considered. The federated system global infrastructure may consist of a set of distributed systems. The network of platforms is not fully connected. Instead, it obtains a shape depending on the shared protocols, the awareness of other systems, and the authorisation of remote users at each individual system. We can consider each federable system as an independent management domain [39].

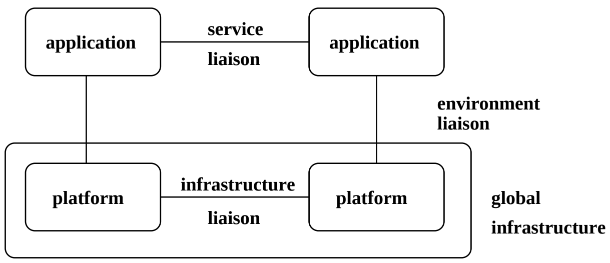

Figure 2.1: Global system view of federated system.

Examples of typical federated system applications include services for electronic commerce, i.e., authentication, billing, payment, and retail. Some services common in distributed systems, like distributed file systems or global naming services, do not appear as joint services of the federated system. Instead, such services can be supported by the federated systems and federated at will.

Typical federated system platform services include those of distributed system platforms, but especially additional services like trading, type repositories, federated binding management, and federated naming services. These services represent meta-information services fundamental for federated systems, and services that exploit meta-information for creating object federations. Some platform services are federated by nature themselves. These services are further discussed in Chapter 4.

Currently, there are not yet any federated system implementations. However, examples of federated system characteristics can be found in TINA, CORBA, and ANSA systems.

The open system framework is supported by a set of new infrastructure services, i.e., middleware functions and repositories. The essential benefits of these services become understandable only through the understanding of the enhanced basic concepts induced by the ODP reference model.

2.2 Federable system architecture

From the point of view of the run-time structure, the architecture of a federable system in Pilarcos has three levels. In this section, we introduce these levels and present their the run-time function along with their connection to the ODP software engineering viewpoints (subject of Chapter 3).

The levels of architecture, illustrated in Figure 2.2, are

- the *business architecture*, which describes combinations of services possibly spanning multiple organisations;
- the *design* of a service, that is, its *computational architecture*; and
- the *component assemblies* that are used to implement the design.

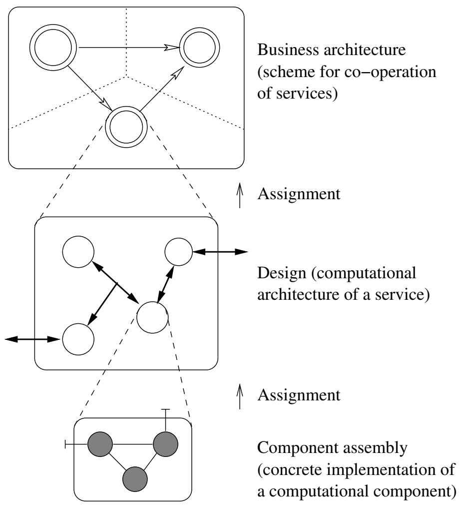

Figure 2.2: Levels of architecture in Pilarcos systems.

The topmost level, the business architecture, is an ODP enterprise viewpoint description of a possible cooperation scheme in which federable systems may participate.

In ODP enterprise language terms, a business architecture is a community specification. The essential content of such a specification is a contract about the cobehaviour of the community:

- the community structure as a set of roles;
- interactions between the roles; and
- assignment rules for objects into the roles.

Each role in the community specification denotes a possible behaviour. The behaviour descriptions are refined with policy statements indicating which parts of the behaviour are prohibited, permitted or obliged to take place and under what conditions. Roles themselves can represent further communities, enabling compositionality.

The ideas of roles as placeholders and assignment rules for constraining the population of roles are fundamental to the *role-based architecture model* used in Pilarcos. The choice of objects to fulfil the roles can be done at design time, but can equally well be deferred until run time. The choice of objects to fulfil the roles can be done by the designer with assistance of a tool or automatically by the infrastructure services at run time; in either case, the assignment rules guide the process. This role-based architecture model enables the objects to be developed and re-used independently, a model naturally suitable for federable systems.

In business architectures, the roles are placeholders for business services. These business services should accomplish significant business tasks and they should be technology independent. Upon a service request, suitable services for fulfilling the roles are found from the dynamic service market supported by business service traders or brokers. This is discussed further in Section 2.5.

The business architecture is used for ensuring that the service providing parties agree on the task to be performed. It can be seen as denoting workflows (for example, [36]) within and between organisations. Since the business architecture is used as the common semantic basis for negotiating about business services, the negotiating parties must have a shared or mappable view of it. This is achieved by storing the architecture descriptions in a business type repository (Section 4.7.6).

The business architecture related to a service should be defined so that it only includes the roles and interactions that all participants must know of. For instance, the implementation of a service within an organisation can involve an internal business architecture, but this internal architecture should be separate from the publically used architecture. Note that a client requesting a service occupies a role in the business architecture just like the servers; the only difference between a client and a server is that the client makes the initiative for the particular interaction.

Designs describe the platform-independent computational architectures of the services that are visible in the business architecture. While a business architecture potentially spans multiple organisations, a design represents the implementation strategy of a service within an organisation. A design is an ODP computational viewpoint description, and is related to the business architecture by the correspondence rule stating that exactly one design must be chosen to fulfil a business role. The chosen design matches by its externally visible interfaces to the business role that it implements.

The role-based architecture model is used in Pilarcos not only on the business architecture level, but also on the design level. The placeholders for component assemblies in the design are analogous to business roles. Each placeholder must be filled by one component assembly. Using component assemblies instead of individual components as atomic building blocks makes it possible to integrate legacy components in the system by embedding them within the assemblies. As physical implementations, the component assemblies are platform-dependent, unlike designs and business architectures. Like designs, the component assemblies are also selected and instantiated at run time by the infrastructure services.

The interactions between component assemblies are captured as bindings (Section 2.4.3). Bindings are also instantiated at run time. Designs can contain both local and federated bindings, of which the latter are used for interorganisational interactions. In business architectures, the interactions between roles set requirements for the corresponding bindings on the design level.

2.3 Application programming interface

The federated system architecture is too demanding for programmers to tackle without proper support. We need to raise the level of abstraction seen by users and providers of services in a federated environment, and free programmers from having to explicitly deal with the technical issues involved in federation.

A central concept in the federated environment is that of service. Here, the term is used of both business level and design level services. They differ only by their level of abstraction; the underlying concepts and the supporting mechanisms are the same. In the Pilarcos environment, services are identified by type, not by name. Thus, the service implementations that an application uses are not fixed, but can instead be selected at run time with support from the infrastructure services.

In addition, the client and server views of the service interface need not be identical or based on a common inheritance hierarchy; the infrastructure services are able to automatically match the interfaces using appropriate interceptors. This is discussed in Section 2.4.1. These ideas facilitate the independence of clients and service providers, a key theme in Pilarcos. Combined with proper encapsulation of components, this makes it possible for parts of an application to evolve independently, bringing a level of flexibility also to the software engineering process within an organisation.

To the application programmer, the Pilarcos framework provides a generic programming interface through which services can be invoked. The exact form of the generic interface depends on the language and technology used. Federation transparency is achieved by using the same interface for both design and business level services, whether local or federated.

When making a service request, the client specifies the service type, criteria for the service such as QoS properties, the business architecture or design involved, and the roles of the client and the service within it. The *service type* is a classifier for services. In addition to the functionality, it also specifies the framework of quality of service parameters related to the service. A service type is not tied to any one role in an architecture nor to a specific implementation, but can be used in several role specifications and have multiple implementations.

Service requests are received and handled by an application binder, a local supporting infrastructure object. The process is described in Section 2.8. All component assemblies have an application binder, which is their single point of access to the outside world and can thus control and enforce contracts and policies (Section 4.6.1).

The bindings between communicating parties are implicit: the programmer does not need to construct any part of the communication channel explicitly. Thus far, only primitive bindings (direct communication between two parties using a simple transfer protocol) have been provided in implicit form. Complex bindings, for example those for group communication or involving QoS control, have required explicit programming within the application. In the Pilarcos framework, even complex bindings are implicit.

Although the service usage model presented here may make the construction of simple applications somewhat more complicated than before, it greatly eases the task of developing and maintaining complex applications, making federated systems practicable to build.

2.4 Extensions to some basic concepts

The federated system model requires a more rigorous set of concepts for maintaining interactions between objects than traditional distributed systems. In a federated environment, objects cannot have inherited information about the behaviour or communication capabilities of each other. Instead, each object of a sovereign system must explicate its properties and negotiate about the federations to be entered.

2.4.1 Services and interfaces

Federated systems are basically founded on the concept of service. In order to facilitate federation negotiations, the views of the same service are separated based on whether a service is supported for others to use, or whether a service is requested.

A *service* denotes a behaviour that can be invoked through an object interface. For a client, the service concept denotes a functionality that can be performed by its system environment. For a set of service providers, the service concept denotes a potential sequence of interactions that may be performed as a result of a defined signal from the client. The sequence of actions is governed by agreed policies on the joint behaviour of interacting object and the state of system environment at the time the service is performed.

For traditional distributed systems service is usually considered to cover only a description of interface structure. The service definition above captures the interface description as one of the items involved, but enhances the concept further to include aspects that are interesting for the organisations. The organisations consider the services also from the point of view of merchandises. Thus, quality of service, remuneration, and monitoring of the actual behaviour against the liaison become interesting. Furthermore, the definition requires that we promote interfaces to capture object behaviour in addition to the interface structure. Behaviour can have either operational or stream-exchanging semantics, as defined in ODP reference model.

Federation establishment mechanisms also require the refinement of interface concept. In the object models characteristic to distributed systems, the interface is the abstraction of functionality that is shared between the communicating objects. In a federated environment, the existence of such shared knowledge cannot be assumed. Instead, all objects have their private views to any interfaces and services. A commonly used interoperation scenario is that of a client and a server object. For those objects we can define a client-role interface and a server-role interface. When one of these interfaces is expected to receive a signal (e.g., operation invocation delivery), the other one should include statements of sending the corresponding signal. If the interfaces do not match perfectly, but are reasonably similar, the objects can still interoperate, but via an interceptor that transforms the signals while they are transported between the interfaces.

Thus a *client role interface* denotes a requester view to a service. A client role interface definition expresses the client's assumptions and restrictions on the service type and the access technology for a requested service. Similarly, a *server role interface* denotes a supporter view to a service. A server role interface definition expresses an implemented service type and required access technologies for the supported service.

In order for the sovereign systems to negotiate about cooperation, the views to offered and requested services have to be captured. Therefore, a *service offer* defines

- the type of server interface supported by a set of objects,
- the service properties in terms of quality of service, and
- the conditions under which the advertised properties are expected to be valid.

Likewise, a *service request* defines

- the type of client interface supported by the requesting object,
- the service properties expected in terms of quality of service, and
- the requirements for the validity of the advertised properties in an acceptable service offer.

For communication to take place through the client's and server's interfaces, the type conformance of these two need to be tested. Even, when the two are not identical, there can be ways of intercepting the two to match to each other for successful communication. A mechanism called type repository is used for checking the type conformance, and to suggest the required interceptors.

These service offers and service requests are mediated through trading mechanisms between organisations and between technology domains within organisations. The matching offers and requests form service contracts that are used for construction of liaisons shown in Figure 2.1. How this happens in practice is described in Chapter 4.

Although the Pilarcos project concentrates on the effects of federation on operations only, the contract model is equally valid and directly usable to streams too. Existing work on quality of service and stream binding are considered and consistent with this work.

2.4.2 Type and template

All the above definitions use the term 'type' when referring to interfaces or services.

According to ODP reference model, a type is a predicate over an object [11, clause 9.7]. An object is of a given type if it fulfils the predicate, irrespective of how the object was created. The concept of type is very general and can be specialised in many ways to form various type hierarchies. Types are used for reasoning and verifying properties of objects, i.e., in the trading and binding processes. Suitable examples of types include "requires CORBA platform", and "supports banking operations for opening and closing accounts".

Types are not sufficient for execution of services, though. Templates [11, clause 9.11] are more constructive by nature and are used for instantiation. For example, program code can be considered to form an object template.

The concepts of templates and types are independent of each other: having a common template does not induce being of the same type, and especially, being of the same type does not induce a shared template. A template is a type that is suitable for object creation in a given environment.

In a federated environment, the service liaisons must be based on the type conformance of their interfaces. For the creation of objects templates are used. Object templates are private for each platform architecture.

Types need to be mapped to templates during service deployment and binding establishment. Type repositories are used for storing the relationships between types, between templates, and between types and corresponding templates. At system design time, or at the time of introducing new building blocks for systems, templates are bound to existing type definitions, or even, new type definitions are introduced. At system usage time, types can be used for global negotiations and the result can then be locally mapped via the type repository to templates. The templates are in this way independently selected for each participating system, and heterogeneous configurations can be automatically managed.

The relationship between objects, types, and templates has been interpreted slightly differently in different contexts [15, 33, 19]. Most systems do not separate the concepts and furthermore trust on inheritance hierarchy as the only means for subtyping.

2.4.3 Binding liaisons and channels

A *binding process* facilitates signalling between object interfaces. In a centralised environment, the result of binding is that the objects are able to address each other. In a distributed environment, binding of object interfaces makes not only the objects aware of each other's interface locations, but also creates a communication channel between the interfaces. In other words, a configuration of marshalling and transmission software objects is created and the required resources for the communication are reserved.

We can abstract the binding process a step further. We can define the result of the binding process, from the point of view of the communicating partners, to be a binding liaison, that ensures that a communication channel can be created between the communicating objects [17]. Thus, the time for configuring the objects involved and the resource reservation can be optimised based on communication load.

A *binding liaison* is a context where the shared facilities supporting the client and server role interfaces have been selected and will be deployed.

The binding liaison has two states; one in which the binding contract has been agreed, and one in which the agreement has taken a concrete, resource consuming format and a communication channel has been established.

From the point of view of the system that supports the binding liaison, a concept of binding type is of interest.

Binding type [17, 5, 4] defines the communication partners, the roles of each participating object and the number of roles in the group. Binding type also specifies the behaviour rules for the group. For example, it defines what transparencies are assumed, how a failure is determined and what kind of security functionality is included. From these concepts enough technical detail can be generated within middleware to create a full blown communication channel between a group of application objects. The supporting services of the communication relationship must use the binding type as a specification of the necessary channels between the involved objects. Also, the end-points of the channels must conform to the interfaces of the involved application objects.

The binding is realised by a *channel* formed by a set of independent *channel sections* and a *channel controller* that carries the binding liaison information even when the channel sections are not present. The number of channel sections involved and the interface types supported at each channel end-point is dependent on the binding type.

A channel is a configuration of intermediate objects that are able to route signals (operation invocations, terminations, flow signals) from one application object to another. A channel is composed of traditional objects, like stubs for marshaling, binder objects for controlling the channel connectivity, and protocol objects for data transfer.

When the location of the channel objects within administrative domains is considered, the channel structure as a set of independent channel sections becomes visible. Each section has an endpoint at an application object, and another endpoint presenting an external low-level access point for communication with objects in other systems. Each section is managed within a single administrative domain. The channel controllers interwork in order to gain control over the whole channel.

The actual channel structure varies depending on which distribution transparencies are selected by the user, and which communication protocols are in use. The actual channel structure varies also depending on the platform architecture and administrative rules on the systems that support partial channels.

The channel objects can be selected at run-time, instead of compilation time as in many RPC implementations. Also, several stubs can be active concurrently, using the same protocol link underneath. Separate concurrent protocols for channel objects are, for instance, group management [38], and QoS management [42]. In a general case, the stubs are not self-sufficient, but require services from management functions like authentication services [21].

The end-points of the channel are determined either by the binding initiator or the binding process itself, i.e., the interfaces to be interconnected can be either identified or searched based on their properties. This also means, that the computational interfaces are bound together, instead of creating a channel between the technical addresses at which the interfaces are initially located. A channel does not necessarily form a static circuit through the network; a channel can be based on connectionless protocols.

The role of the channel controller is dependent on the binding type selected by the object requesting to be bound to other objects. For instance, the channel controller may monitor the membership of the binding liaison and act as a leader in failure detection protocols.

The channel controller allows the channel configuration and parameters to be modified during the service liaison duration. Changes can involve, for example, multi-cast group members or timeout values when a fixed network line is switched to a mobile network line. A channel controller is a direct client to all of the channel object's management interfaces, and therefore it offers a combined control interface to all of them. The channel controller has a specific object at each domain, and those objects may cooperate in order to offer a joint binding liaison management service.

Channel management actions can occur because initiated by management requests arriving to the channel controller interfaces or because of internal events of the channel. For example, if a breach of the binding liaison occurs, the channel may be reconfigured. It should be noticed that the breach may be related either to the federated application or to the virtual platform supporting the application federation. In the latter case, the application federation can be retained and reconfigure only the lower service layer. This means that the channel type is retained but the instances for it are replaced.

The binding liaison information is captured to a *binding contract* object that is replicated to each of the sovereign systems involved. The binding contract object is encapsulated into the channel controller and thus managed through the channel controller interfaces. The binding contract supports the following information for controlling the architecture and design level aspects:

- a service type identifier (for each type system involved),
- service type specific QoS agreements,
- a business architecture identifier,
- a role identifier within that business architecture,
- the selected design for that business role, and
- the role (component placeholder) identifier within that design. Reference to the policy context becomes identified with the last four items.

Further, the binding contract supports the following information for controlling the channel:

- a binding type identifier (for each type system involved),
- binding type specific failure detection and recovery protocols, failure defined as not being able to meet the QoS agreement,
- remuneration protocol,
- technical descriptions for interface signature expected by the client (can be differently selected at each sovereign system),

- communication protocol,
- channel type identifier (for each type system involved), and
- interface reference for the channel controller (separately for each sovereign system).

The binding contract information is replicated in each of the objects involved in the liaison. Because the interfaces can reside at separate systems, the data representing the contract information may have different local formats and coding. Each object can use the local copy of the binding contract as policy information or parameters to its internal activities. This mechanism can be used to provisioning of the binding contract as part of the object behaviour (see [27] for an example).

The ODP reference model discusses various forms of bindings, i.e., *simple and complex bindings* and *implicit and explicit bindings.*

The complex bindings differ from simple bindings in several key points. A simple binding is a direct one-to-one communication channel between two operational interfaces where no management operations on the communication channel are supported. A complex binding involves more complex mechanisms. First of all, complex bindings cover not only operational interfaces but also stream interfaces (continuous media), and allows multiple participants to be bound together. Furthermore, a complex binding supports interfaces for managing the communication channel during its life time.

Traditionally, a simple binding has been supported by implicit binding mechanism - the communication channel is established as part of the operation call performance. However, there has not been implicit support for complex bindings; applications need to create the communication channels themselves and take care of the management operations as well.

2.5 Scenario for accessing business services

Let us consider a scenario where an organisation supports a portal. Through the portal, clients can access – in a tailored way – a set of business services. Behind the portal interface there is an infrastructure that dynamically deploys the objects comprising the requested service and tailors it according to the client preferences. The portal can be represented by a business service trader or a business service broker.

The parties in this scenario may be complex; the organisation behind the portal can be a virtual company comprising of various autonomous organisations, likewise the client can be a requester on behalf of a large software setup performing duties of a business service.

The services fulfilling the business architecture are found via the service markets supported by business service traders or brokers. The main difference between traders and brokers is in the amount of responsibility over the mediated services taken, level of automation in the selection process, and the capabilities of marketing combinations of services with independent origins.

Business service brokerage is a business of its own right; location and number of such services is guided by market forces. The broker has responsibilities over the quality of the mediated services, and the negotiation between clients and brokers is manual. The negotiation involves a business contract that does not participate any computing tasks, but in text form sets obligations for the involved companies about the computing task in question. The broker also provides technical information about the contracted task.

Trading implements a simple, automated business contract negotiation. The protocol involves only a trading request parametrised by a contract draft describing the needs and properties of the client, and a reply with a set of potential contracts filled with service provider information. The trader does not need to know which one of the contracts the client picks - that is part of the federated binding process explained later. The business contract captures the business architecture, which is required to force a matching ontology to be used by both service provider and user. Additional services can be involved into the contract - like billing for the service or checking for authorisation of using the service. For checking the technical suitability of a business service, the contract must include also technical interface information.

Matchmaking between the client request and the services offered through the portal is based on

- a business service description that expresses what kind of service is requested and how the negotiators see the workflow related to that service to involve the communicating parties;
- a technical interface through which the client is prepared to communicate in its role within the business service;
- business level requirements set for the service, such as policies governing the service behaviour, quality of service aspects.

The match making must result into sufficient information at the service providing system for automatic service deployment to take place. The deployed service is a configuration of computational objects and channels between them, and the configuration also contains a communicating channel reaching towards the client outside of the system. The deployment process itself is described in the next section.

2.6 Business service deployment process

While business architectures are globally understood, each role in the architecture can be implemented with a different design in each organisation supporting that service. The design can be produced within the software engineering process (see Section 3) or can be brought to the system as an outsourced design. In both cases the design is retrievable from the local design repository, using the business service as a key. If multiple designs are available, also multiple offers can be available through the business service broker or there is a local heuristic to choose the design to be applied.

The design description introduces a set of computational (logical) objects, their interfaces, bindings between the interfaces, QoS agreements and policy rules governing the shared behaviour of these objects. It may also set requirements for allocating objects, for sharing resources, and for performance optimising strategies. The design should also be reusable for other business architecture descriptions. The computational objects needed to populate the architecture description are found via local component assembly repository, or via a trader.

The computational design shall be further refined for solving engineering problems, like data partitioning or replication, data transmission modes, and failure recovery. However, the result of this refinement identifies too small units for global markets - the expense of maintaining and retrieving information would be too high because of the huge number of required objects. Therefore, the Pilarcos project places emphasis on the computational objects and services, and shows engineering refinements as internal structures and requirements for the marketed objects.

The computational objects are representations of assemblies where objects and simple bindings between them are established. More complex configurations are then formed based on these functional groups and complex bindings can be established between the groups.

The component assemblies are platform dependent implementations of an object. The designs are platform independent and thus allow heterogeneous, distributed implementations to be set up, as long the whole implementation stays within the same administrative domain.

The component assemblies include implementations of the objects involved, interfaces for them, simple bindings between the interfaces, listing of interfaces that are left for external objects to bind to, structures for instantiation related data like interface locations.

Naturally, it is expected that component assemblies are highly reusable for various computational designs, and thus, become part of numerous business architecture implementations.

The component assemblies are also the unit of managerial control within the organisation or within the system. This solution causes engineering architecture requirements for the computational objects also in form of compulsory management elements associated to each computational object. The management elements can be either be required to be implemented by each marketed object, or, there can be platform services that offer these elements for the objects to use. In Pilarcos, the goal is to find a design by which generic infrastructure services can be exploited; these include application binders, in cooperation with the policy repository.

2.7 Management aspects

Management issues in the Pilarcos architecture include management of application objects, federations between them, infrastructure service management. Shared special factor in these are contract based management and organisation-wide policy management.

A contract is an agreement between two or more objects on their shared behaviour, i.e., interaction sequence and the goal of that behaviour. As an example, refer to binding contracts and their management in Section 2.4.3.

A policy is a decision that has to restrict object behaviour within the system. For example, a policy "opening times are 9-18" should restrict operations for creating new bank accounts to these times.

The organisational management tools are based on a policy repository where a hierarchy of enterprise, information, computational, engineering and federation instance policies and contracts are stored. This information is inherited by the additional management elements introduced above to be included into each object that provides business services. In this way, the organisational and federation negotiation decisions can have effect on the newly started or already running services.

An organisation can represent a single user as well. For example, the user of a ubicomp device has policies about interacting with system and exploiting surrounding services.

The policy repository provides means to check the consistency of all policies within the organisation.

In the case of a federation, the established federation contract becomes the governing policy authority for the partners involved for the duration of that federation liaison. As described earlier, an organisation can simultaneously participate multiple federation liaisons even if they are contradictory, as long the organisation's internal policies are not violated. For security reasons, a federation contract must restrict the further usage of information originating from a federation.

The federation contracts are negotiated during the remote service deployment process and binding establishment between the communicating partners in all involved systems. The federation contract needs to be introduced temporarily into the local policy repositories, so that the policy rules agreed will take effect on all the objects participating the federation.

2.8 Summarising the federated service acquisition process

In a federated environment, acquiring a service from an other organisation and binding to the related interfaces is a complex process. For application programmers, the situation should be made as simple as possible. The Pilarcos framework supports simple concepts for that purpose: service type and binding type. With these concepts the application programmer is able to express the application needs for service, and the infrastructure services can have enough instructions to take care of the federation negotiations needed. Essential improvement here moves the responsibility of establishing and maintaining federated bindings from application programmers to the infrastructure services.

Figure 2.3 summarises the overall system architecture. Section 4 further discusses each middleware service and repository involved. In the following paragraphs the basic flow of control is briefed.

The client makes a request to a local infrastructure object supporting it, client application binder, expressing the business service description and the call interface it uses. The request gets forwarded to the portal that matches the request against the offers it has stored from the service providing systems.

The client application binder then sends a prepare service request to the address given by the portal. That request invokes a deployment process at the service providing system.

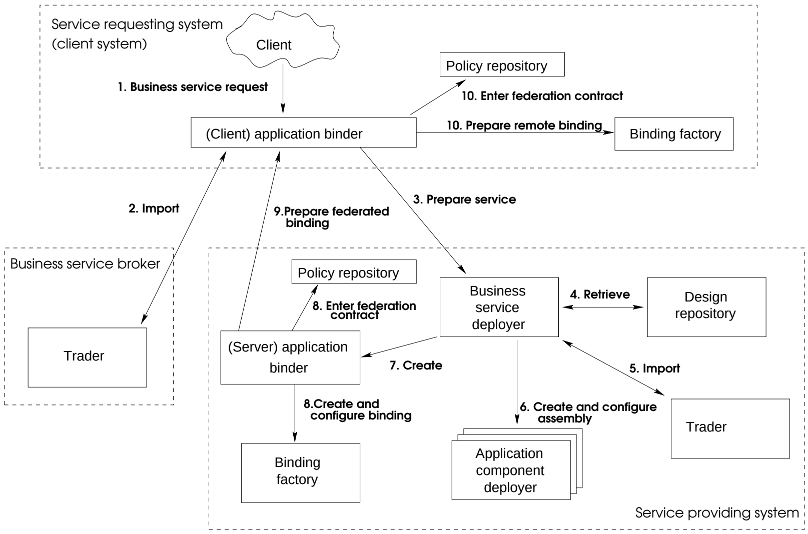

Figure 2.3: Overview of the system.

The business service deployer of the selected organisation or selected system retrieves a corresponding design, i.e., computational architecture description. Essential building blocks in that description are service types and binding types. For each service or binding in the description, the deployer needs to choose and instantiate (if not already running) a suitable implementing module. The services can be found via an organisational trader that supports knowledge about application objects running or instantiatable. For the binding objects, the business service deployer creates an application binder for each binding endpoint. It also instructs groups (or most often pairs) of application binders to connect to each other. The application binders create mutual connections by the help of binding factory - which is able to create both local and federated bindings. The binding factory uses the help of a type repository for knowledge about how binding types are instantiated at each platform.

The instantiation process is guided by application object type and binding object type information found in a design description that matches the business service in question. The instantiation process is naturally platform dependent, unlike other parts of the deployment process.

As a response to the prepare service request, the client application binder receives a request to prepare a federation contract in the local policy repository and a binding object towards the newly created binding object for the service. For the federation contract, the request contains suggested agreed values for all business, computational and engineering aspects of the federation. The client has initially given suggestions at the initial service request, so in a simple case this response completes the negotiation. In more complicated cases, for example when security reasons have forced keeping back parts of the contract information, further messages may be exchanged between the service deployers. For the binding, the request contains binding type and address information. After this process the client's original service request can be forwarded to the service as a operation invocation through the transport connection that has been created between the outermost binding objects.

The overall system architecture view spans multiple administrative domains as follows. The business service broker represents a global market of business services. The brokerage service can be distributed, but essentially provides a separate service for fining out from which organisations business services are available. Each organisation has their private business service deployers. The deployment process is governed by organisational decisions on technologies, remuneration, evolution of provided services, etc. Heterogeneity does not appear only between organisations, but also within each organisational IT system. Thus, the business service deployers are platform independent and exploit the services of a group of platform dependent application component deployers. The design repositories and component assembly traders are organisational repositories. Design repositories and organisational component assembly traders do not automatically federate but give a local service. Possible federations amongst design repositories, for example, need human intervention via software engineering tools (see Chapter 3). Organisations can also federate between the assembly repositories, but only as means of design time loading or publishing of assemblies. For use in the execution time environment, the assemblies must be locally available for performance reasons.

2.9 Interaction between service design, deployment and exploitation aspects

The open system framework deliberately integrates together design and run time information. The tools for this intertwining are provided by the dynamic middleware repositories:

- business service brokers and traders offer an up-to-date view to offered services world wide;
- the business services on markets are categorised based on the business architecture they are able to participate in; furthermore, the set of business architectures can be evolved over time as the descriptions are stored on-line;
- in each organisation, the selected set of business service elements can be supported by a set of designs, i.e., computational viewpoint descriptions of the service element; like the business services also the computational services are available on-line, and categorised with a dynamically evolvable type system;
- finally, in each organisation, the computational services can be implemented in a distributed manner; the mappings between platform independent designs and platform specific implementations are retrievable from local type repositories and traders; again, the repositories are at place to ensure system evolution.

The essential programmer concepts of service type and binding type are supported by the deployment process. The key is to use type concepts for applications to set requirements for the services they need, and map these declarations to detailed instructions, i.e., templates for instantiation, later in the process. A type is just a predicate over an object's externally visible properties, while a template denotes all necessary details for creating an object on a given platform. Thus, types are interpretable globally, while templates are restricted to a given platform environment.

The software engineering process can roughly be divided in projections where

- business architecture is specified,
- computational designs for a business service are defined, and
- component assemblies for implementing a computational objects is constructed.

Correspondingly, the products of these projections are stored into

- global business service broker's type repository,
- organisational design repositories, and
- component assembly repositories and organisational component assembly traders.

From these repositories the same elements are used for installation of appropriate component assemblies and further for deployment or run time instantiation of the parts needed to provide a business service.

Chapter 3

Software engineering with ODP viewpoints

As already discussed in Section 2.9, the ODP viewpoint languages directly support architectural descriptions necessary for the service acquisition processes and federation. This chapter will further describe the viewpoint languages involved, and the software engineering process suitable for evolving new, open systems.

3.1 Overview of ODP viewpoints

The ODP viewpoints define projections of a system under description. The viewpoints discuss various aspects of the system as follows [12].

- Enterprise viewpoint [18] discusses the system under specification and its behaviour in relation to its environment. Basic concepts for description are those of community, roles, interactions between roles, policies governing the collective behaviour of the roles, and rules about assigning components to each of the roles.
- Information viewpoint discusses the scope and processing rules of information needed for and by the system. The logical content of the information storages involved, and the logical structure of information are described. Required rules for consistency are expressed as invariant and static schemata, modifications to the information as dynamic schemata. Each schema is a logical statement about the values, not programmatic expression.
- Computational viewpoint shows the system as a collection of logical components together with their interfaces and behaviour.
- Engineering viewpoint is concerned of showing how the system is supported by the open infrastructure services, like processing on some nodes and communicating across the network, and how distribution is managed.
- Technology viewpoint concerns selected standards and solutions for actual implementations, and also on additional information for testing.

The relationships between ODP viewpoint languages are not commonly accepted. The effects of changes in one viewpoint onto another are not established by the standard, and there seems to be very little to force on all specifications, as the viewpoints are rather like projections than overlapping views.

The Pilarcos method of using the viewpoints takes the engineering view as the complete system description and uses the other viewpoints to give restrictions and requirements for the construction of that view.

The four viewpoints show projections of the engineering view of the system, thus they do not affect each other. They are interrelated, but changes in one viewpoint are not propagated to other viewpoints than engineering. Violations against other viewpoint specifications are flagged for the designer, and the corrections can be done at any chosen one of the viewpoints. Therefore, it is also the designer's choice how much effect reuse of existing components have on the architecture.

Notations for the various viewpoint languages have not been fixed yet. The ODP viewpoint languages are abstract, they provide the concepts and relationships between concepts, for notational languages to present. For a specification tool a simple set of concrete notations, preferably graphical notations, need to be selected.

For enterprise, information and engineering viewpoints, UML has been considered and tested as a notation. However, it does not appear fully suitable to these tasks.

3.2 Brief tutorial on ODP viewpoints

The ODP viewpoints allow object systems to be specified in an organised, guided manner. However, the viewpoint rules do not instruct on the level of detail or completeness of the specifications. Those aspects must arise from the software engineering process in which the viewpoint specifications are involved. Because of this multi-purpose nature, the specification rules are very general and should have their specific interpretations in each use-case. We can still claim that the goal of the ODP reference model is not to cover the full software engineering process, but only to support system specification and specification conformance analysis. For implementation specifications the ODP viewpoints would be too hefty.

The ODP reference model defines five viewpoints – enterprise, information, computational, engineering and technology viewpoints [10, 12]. A system must be specified from each of these viewpoints. Each viewpoint specification is a consistent and complete specification on its own, but it only considers those aspects of the system that are valid on its point of view. The viewpoint specifications do not overlap totally, but they may show different level of detail in the areas where they discuss same or related features. The engineering viewpoint specifications are tightly related with the ODP infrastructure model that is specified as part of the engineering viewpoint specification rules. The viewpoint specifications can be considered as projections of a system.

3.2.1 Enterprise viewpoint

The enterprise viewpoint description of a system specifies the activities and the responsibilities of the system [12, 18, clause 6]. Activity means any information exchange sequence and it is a high-level abstraction of the operations within the system. The system itself can have any granularity that is interesting. The system can be as wide as a global information network with all applications or as small as a memory cache in a processor. The enterprise specification identifies the system, its environment, and the required communication of the system and its environment. The specification answers to the questions “What is the purpose of the system?” and “What services the system is responsible to provide?” and “Who needs the services?”.

The objective of a enterprise viewpoint specification is to define the purpose and scope of the system under specification. This is done by describing the system structure, the evolution rules for that structure, the responsibilities the system has, and the functionalities the system is required to implement.

An enterprise specification consists of a group of communities. Each community is constructed for a given objective, i.e., is a logical collection of performers and functionality that is required for achieving the community goal.

A community is a configuration of enterprise objects with a contract on their collective behaviour. The contract structure is defined by the community specification.

A community is specified in terms of

- roles that is a placeholder for an enterprise object that performs given actions in the community; the role specification gives requirements and restrictions for the behaviour of an object;
- rules for assigning enterprise objects to roles; the policy rules can include both behavioural and non-behavioural properties of the potential objects or properties over relationships between objects in the roles of a community;
- policies that apply to roles; policies govern the cobehaviour in the community;
- behaviour that changes the structure or the members of the community during its lifetime.

Communities have relationships to each other. Especially, one community can populate a role in another community. Thus, more complex systems can be constructed by the population process alone. However, the community specifications can also force some interleaving for the communities: some roles can be defined to be used in multiple communities, or some roles can be required to be populated by the same object in order to cause information or control flow between the communities.

A community is also capable of communicating with its environment. The environment is not specified in detail, but only the requirements for it that are relevant for understanding the behaviour of the specified system.

A community specification gives the rules for the behaviour of the community and the responsibilities for the community. During the lifetime of the community, the members may leave or join the community; however, even when a role is temporarily without assigned actor, the community still carries all responsibilities related to that role. The degree of dynamism in the community structure is dependent on the community specification – no changes are allowed unless there are specified behaviour rules for that change.

Communities can also have specified capabilities for creating additional communities within the enterprise specification: Federation is a special case of community establishment. Some of the specified communities are able to dynamically create a new community specification that is then populated.

A policy is a decision that restricts the potential for actions and interactions implemented in an object. Policies can be associated to enterprise objects, roles or communities can be restricted by policies.

Alternative way for describing cobehaviour in the community is to model with process. In process modelling the focus is on the whole community moving from one joint state to another because of a chain of actions with a goal. A process is a chain of actions that can be started by an external or internal trigger, and that terminates when a target state is reached.

3.2.2 Information viewpoint

The information viewpoint description of a system identifies logical information entities, their logical contents, their repositories and the objects that are responsible for the information flow in the systems [12, clause 7]. Questions for information viewpoint specification are "What information is needed to support the system's services?", "Where does the information come from and go to?", and "Is it necessary to store the information somewhere?". The information viewpoint specifications should not describe data structures, but only the semantics of the information. Also, the technique of storing information is irrelevant in this viewpoint (as the logical infrastructure supports storage services).

3.2.3 Computational viewpoint

The computational viewpoint specification captures the behaviour of the system [12, clause 8]. Behaviour is an abstraction of how things are done, in contrast to the notion of what things are characteristic in the enterprise viewpoint activities. An activity identified in enterprise viewpoint may involve several objects to perform a sequence of operations in the computational viewpoint. The computational viewpoint shows the system as a composition of logical objects. For each object its interfaces are described. If the interface involves operations, each operation gets logical parameter descriptions (logical information components, not technical data structures) – if the interface involves streams, each data flow component of a stream gets logical protocol descriptions instead. This is the viewpoint that usually explicitly shows potential for distribution. Neither the enterprise viewpoint nor the information viewpoint specifications need to express any distribution concerns. The computational viewpoint answers to questions like "Which operations are available?", and "Who (which logical entity) performs the operation?".

3.2.4 Engineering viewpoint

The engineering viewpoint specification identifies the infrastructure services needed for the system to operate [12, clause 9]. The RM-ODP engineering viewpoint defines the set of available infrastructure services, and all other engineering viewpoint specifications should show how the specified system utilises these services. The engineering specification, therefore, answers the question "By which services are the computational objects supported?". The ODP infrastructure model identifies a set of global, distributed basic services that should be available at each node in the global system. These include invocation of operations, transfer of continuous data as streams, trading, type repository functions, etc. [14, 16, 15]. These services facilitate selective transparency of communication between objects.

The ODP reference model defines an abstract infrastructure to offer basic services like distribution transparent communication primitives. The infrastructure is described using some supporting concepts, that give an internal engineering view of the middleware. However, these concepts are only used for description purposes, not as technology requirements. The abstract infrastructure model is described in Section 4.1.

3.2.5 Technology viewpoint

The technology viewpoint specification shows in a concrete hardware and software configuration how the system services and other required components are realised [12, clause 10]. The specification answers the question "How are the infrastructure services realised?".

3.3 Interpreting viewpoint correspondences

The defined correspondences between viewpoint specifications of a system are of specific importance for the system design process.

The current committee draft for the ODP enterprise language [18] requires the system specifier to declare the relationships between the enterprise specification structures and the structures in other viewpoints. The Pilarcos toolset has built-in some basic assumptions as follows:

- For each role in the enterprise specification there shall be a list of the computational designs with interfaces that correspond to the enterprise action.
- For each enterprise interaction type there shall be a list of binding types capable of representing the enterprise interaction type.
- For each role affected by a policy in the enterprise specification there shall be a list of the computational object types that exhibit choices in the computational behaviour that are modified by the policy.
- For each computational object there shall be a configuration of engineering objects able to perform the tasks of the computational object.
- For each computational binding there shall be a list of engineering channel types and stub, binding or protocol objects that are constrained by the binding and the corresponding enterprise interaction.
- For each computational object there shall be a list of those engineering nodes that support some or all of its behaviour.

- For each community and role in the enterprise specification there shall be an engineering viewpoint controlling object governing the behaviour of those engineering objects that correspond that community or that role in question. The responsible controlling object is the appropriate application binder together with the associated policy context within the policy repository.

These compulsory mappings give structure for the Pilarcos software engineering process. They force a top down refinement between enterprise viewpoint and computational viewpoint, still keeping the design independent of technical detail. Both viewpoints provide an architectural description, but at different levels of abstraction. They also force a refinement between computational and engineering viewpoint, where the engineering view is dependent on the platform and shows the full rigour of distribution. Finally, they force an enterprise viewpoint structure to be automatically imposed on the engineering viewpoint.

In addition to these, the following consequences are visible between enterprise, information, and computational viewpoints and the structure of the engineering viewpoint specifications.

Corresponding to the enterprise view, explicit contract management objects represent the organisations, communities and roles. A contract management object stores policy information inherited from the policy repository and provides a service interface for changing that. Changes can occur because of organisational management operations or because of establishing a federation. A federation takes place as an application level phenomenon and a federation contract only involves a specific instance of using services; it is not an organisation to organisation agreement for all future cooperation cases around that application.

Information viewpoint must define the information storages and relate them to enterprise roles. It must also define artifacts that are stored within the storages and transferred between enterprise objects. For interoperability, the artifacts should be as generally exploitable as possible. Artifacts can be standardised at two levels: within an enterprise for federation purposes or externally, for example predefined DTDs for an application domain can be standardised for example by W3C.

Invariant and static schemata become visible in the engineering view as policies controlling the data consistency. Various triggers and interceptors can be used for this. An information object defines a control domain for the object introduced to the engineering view.

The dynamic schemata interrelates enterprise viewpoint activities and computational viewpoint operations. In the engineering view, the dynamic schemata become visible as pre- and postconditions of operations and triggers derived from the enterprise specification.

In Pilarcos, decisions done for the whole enterprise in technology viewpoint are also stored to the policy repository and inherited down to the engineering level via the policy repository.

3.4 Software engineering processes for the Pilarcos architecture

The foreseen Pilarcos tools support the following production processes

- process of specifying and designing a new system;
- process of implementing, storing and reusing independent components; and
- process of defining metainformation to be used in the process of integrating components into a system that provides a feasible service

and the following integration processes

- process of analysing the federation of multiple existing systems;
- process of executing a global cooperation scheme based on a shared business architecture that gets populated by appropriate service providers and service components by demand; and
- process of integrating components into a service providing system.

The Pilarcos architecture model operates with three independent levels of descriptions and implementations. Each of these levels is manipulated independently from the other, with the corresponding viewpoint related tools: business architectures with enterprise viewpoint editor, designs with computational viewpoint editor, component assemblies with engineering viewpoint editor and programming tools. The system will be brought to life by defining correspondences between specifications via the viewpoint correspondence tool, and by registering implementations to the assembly repository and traders.

If a new system is to be developed totally from the beginning, the tools need to be repeatedly used for supporting each phase of the development cycle. Although the enterprise viewpoint stresses the same aspects as system analysis, there are aspects that need to be captured in other viewpoints as well. The system design affects the enterprise viewpoint specification as well, although the majority of work in design phase probably go to computational viewpoint designs.

However, the development of a standalone new system is not what the Pilarcos architecture is designed for. In the contrary, gradual system evolution is assumed. The system is already running with a set of business architectures, a set of designs and a set of component assemblies. In this environment, the system developers, application programmers and system engineers create new constellations of existing services and introduce additional services.

The enterprise specification provides a business architecture that is used for global negotiation about suitable service types to participate a cross-organisation work flow. Such an architecture description is stored into the type repository that supports the business service trader/broker.

3.5 The toolset

The first cut of Pilarcos toolset provides only tools for entering enterprise and computational viewpoints and analysing the engineering view.

The Pilarcos toolset offers a graphical editor for each viewpoint for designing new aspects of the systems.

The enterprise specification provides a rough architecture description for the system under design and need to be refined with a suitable computational architecture description. The computational design is stored to an organisational design repository. The repository can be equally used at system development time as a refinement step and at service usage time for the service deployment process.

For the construction of designs in the computational viewpoint, the concepts of service type and binding type are offered. For application programmers, both of these can under normal circumstances be readily available from the organisational type repository. The application programmer can implement a new component assembly that implements a service type in a new way. The application programmer can also exploit other service types directly, trusting on the already implemented component assemblies available through the organisational trader. It is the task of system designers, platform developers, and system programmers to introduce new service types and binding types. The type descriptions and relationships between types are stored to type repositories using directly or via a graphical user interface to the repository.

In the engineering viewpoint, component assemblies are constructed with platform specific programming tools, for example the programming tools for EJB components or CORBA components. The assembly packages contain a required set of metainformation. The Pilarcos tools package contains a repository into which these packages can be stored. As part of the process of storing the assembly, the metainformation is used to provide the required export operations to the assembly trader and checks on the availability of required type definitions are performed.

Consistency checkers and analysers are needed for each viewpoint specification separately. This can be done based on extended state transition model analysis tools. However, in the early phases of the project, the toolset is only designed so that verification tools are easy to integrate later on.

In addition, conformance with the restrictions and requirements of other viewpoints need to be checked. As the output of each viewpoint editor is stored in form of its engineering viewpoint consequences and into the policy repository, these two collections of information can be searched for inconsistencies after modifications in any of the viewpoints. However, much of this work requires further development of analysis methods on action refinements and service equivalence classes. That in itself is a topic for a whole research project, and thus the Pilarcos tool only lays a requirement for such work in future.

For performance considerations a separate analyser can be provided to support design choices [20], if UML is used as a common notation.

In the following, each tool is briefly characterized.

3.5.1 Enterprise viewpoint editor and analyser

The purpose of the Pilarcos enterprise viewpoint editor is to produce business architecture descriptions to be stored into the type repository of business service brokers.

Enterprise specification editor supports a graphical method for defining

- communities, in terms of their structure (roles and their relationships), behaviour and policies;
- domain of control of the specifying organisation;
- roles within communities, in terms of behaviour, policies, and assignment rules;
- rules for changing the structure of the communities; and
- enterprise structure in terms of overlapping and federation rules of communities.

The result of the enterprise viewpoint specification process is a set of community descriptions. The enterprise view does not match to an organisation. An enterprise specification naturally covers cooperation across multiple organisations.

The enterprise specification can be divided to a set of community specifications. The editor should support divisions based on an enterprise level structure, like federation between domains within the enterprise, or on a functional split, for example pulling accounting and marketing out as separate community descriptions. The enterprise viewpoint analyser checks separately the consistency of each community and the interactions between the communities.

A technique to connect the functional communities together to form the full enterprise specification is to require certain roles in these communities to be combined. The process of combination must involve consistency check for the policies governing objects in those roles. In the specification tool, each community is stored as a set of finite extended state transition machines, also any consistency checked combination role. The enterprise viewpoint analyser checks whether there are contradictions in the policies for a role or for a combined role over multiple communities.

The domains controlled by an organisation can be made visible in the enterprise specification. When this is done, the enterprise viewpoint specification can be used as a design and run-time tool for managing computing within the organisation, as an essential aspect of the enterprise specification is defining business and computing policies.

A role defines what interactions an entity in that role is expected to be capable of participating. The definition of the behaviour in this way requires a shared ontology on abstract behaviours - such an ontology exists in the set of business service types commonly agreed on, somewhat standardised, and supported by the type repository serving business service brokers.

The behaviour associated to a role or a community is described by an extended state transition machine. A transition denotes an activity into which the role or the community may participate. The transitions are further guarded by policy statements. A policy is an additional constraint for a transition and denotes whether in a given situation the state machine is permitted, prohibited, or obliged to initiate or participate a transition.

The state transition model is further extended by alternative strategies, that means, some of the behaviour descriptions can be exchanged to others. Strategies can be considered as epochs in the lifetime of a role or a community, an epoch change possibly causing changes in the behaviour or the objective.

A policy choice denotes enabling and disabling various strategies. The policy choice can be inherited from the enterprise specification, be entered though organisational policy repository or be triggered by some state internal or external to the system.

Policies can be denoted as additional constraints (prohibitions, obligations, permissions) over system state to be used in addition to the pre- and postconditions on activities in the system. In this form, policies are stored into policy repository, form which each activity inherits information that affects its engineering level behaviour.

The roles are populated by actual service providing solutions either at design, installation or run time. The population process is guided by assignment rules.

The assignment rules for roles form a bridge to the computational viewpoint. The computational object selection must conform to the assignment rules. These rules form policies for the computational components selected and give guidance when either an identified component is needed, or related components are needed for the community roles (e.g., same component must populate several roles in the enterprise).

The entities populating a business architecture can join and leave the formed community according to some joining and withdrawal rules, dependent on the application logic. An empty role can be repopulated by using the same assignment rules again.

During the execution of the system, if obliged things do not or prohibited things do after all happen, sanctions or corrective actions occur. The arbitration policy need to be presented in the enterprise specification.

It is not necessary for the business architecture descriptions to be static in nature. There can be rules changing the structure of the community at its life time, i.e., roles can be added and removed.

The stable phases are called *epochs*. Each epoch represent a fixed set of services made available from the community. A change in the community structure that changes the set of available services, is an epoch change. All other changes in the number and type of members in the community are considered to be expressions of different *cardinalities*. A cardinality expresses the allowed number of simultaneous players of a given role in the community. A cardinality of zero denotes, that a role is allowed to be empty at times.

3.5.2 Computational viewpoint editor and analyser

The result of computational viewpoint specification is a set of designs representing the generic structure of an application that provides a computational service. The design essentially contains a set of computational objects (comparable to roles in the enterprise viewpoint, i.e. placeholders for implementing constructs), their interfaces, and communication relationships between the objects.

The resulting designs from this viewpoint are stored to design repositories. The repositories can also be used for retrieving already defined designs for reuse and modification. There can also be federation relationships between design repositories, thus making it possible to support reuse of designs across organisations. This sharing does not, however, appear in the execution environment, as the process is too heavy for run time usage.

The computational objects are represented as extended state transition models, in the similar way as enterprise viewpoint roles, only at a more detailed level.

In the software engineering process, transition from enterprise viewpoint to computational viewpoint is done by replacing each abstract interaction by a computational interaction sequence. Pre-designed interaction sequence models can be reused in this process.

The correspondence between a business architecture role and a design is left for the designer to decide. There is no automated way of determining the relationship that would adequately capture semantics of the relationship. The relationship is entered to the system through the viewpoint correspondence tool once both participants of the relationship are defined. There can be multiple designs corresponding to one business architecture role, and multiple business architectures in which a design is applicable to fulfil a role. The concept used for expressing role related requirements and design related offers is that of *business service type*.

The interaction sequences require certain interfaces. The list of required interfaces is split into suitable groups to form computational objects. A suitable split can be found for example by taking a required information storage as a starting point and associating information registration and retrieval interfaces to it.

Computational protocols need to be refined for all permitted and obliged activities, triggers for violation arbitration should be provided for all prohibited activities. Additionally, triggers for all activities initiated by the role under consideration should be captured.

The interactions in the sequences should be generic and move around artifacts. The artifacts can then be related to the information viewpoint artifacts. The protocol may have additional requirements for data that is transferred. That is collected separately but is not reflected back to the information viewpoint. However, it is passed down to binding requirements in the contract as an computational management artifact related to the chosen computational methodology.

For computational bindings the engineering view provides a binding controller. If there is a breach of a contract, it reports to the enterprise arbitrator that starts a recovery process selected based on the type of the breach and the type of the contract. The channel controller also controls QoS aspects.

Computational viewpoint specification provides an architecture view that can be populated by existing components. Components are chosen based on meta-data that covers the behaviour, binding type, technical requirements, server/client profile understood, policy information supported.

A computational object may be related to an information viewpoint repository or artifact, and must be related to at least one role in the enterprise specification as a member populating a design.

On the verification tool area, the computational viewpoint analyser should check whether the computational specification for a community is deadlock free, has liveness properties, etc. However, the Pilarcos tool offers an interface for such a service, but does not include a verifier.

3.5.3 Engineering viewpoint editor and system composer

For the production of independent computational components, the engineering viewpoint editor allows refinement of computational objects in a similar way as enterprise viewpoint roles are refined into computational objects.

In practice, the engineering viewpoint editor provides two views to the system under construction. First, a view where components and their configuration is shown. Second, a closer view for each component separately, for actual programming. The first view essentially has two kinds of components inherited from the computational viewpoint: application elements and binding elements. Both of these can be constructed in the same way by programming a component assembly package. These packages are stored (and can be reused from) component assembly repositories that are local for each organisation.

A complete engineering view of a system essentially consists of two things:

- the computational architecture description and population rules for each computational object; this information is adequate for instantiating the system for execution and managing the bindings between computational components;
- management objects derived from all other viewpoints; and
- Pilarcos infrastructure services.

3.5.4 Viewpoint correspondence tool

During the specification and design of a new system or application, the viewpoint editors give projections of individual elements of the system. The viewpoint correspondence tool is needed to show the relationships between the business communities, service types, and component assemblies.

The relationships include

- designs can be appropriately used in one or more business architectures; to support this the business service broker uses the business service types available at its type repository;
- component assemblies can be appropriately used in multiple designs; to support this the component assembly traders use computational service types available at the organisation's type repository; and
- binding objects are applicable to multiple designs; to support this binding types each represent a group of technology dependent biding templates which are represented as references to corresponding implementations stored in the assembly repository.

The viewpoint correspondence tool therefore is a combination of an editor for creating type descriptions and storing relationships between these descriptions.

3.5.5 Management tool

The management tool provides means to

- populate the traders with appropriate offers for business services and computational services;
- install component assemblies; and
- instantiate business services.

These services require the support of new infrastructure services as described in Clause 4.

Chapter 4

Infrastructure services

This chapter discusses the services and repositories present in the Pilarcos infrastructure. The infrastructure is an enhancement of the ODP infrastructure model and introduces two new features. First, federation transparent bindings between objects. Second, implicit binding process resulting to complex bindings.

4.1 ODP infrastructure model from engineering viewpoint

The ODP reference model defines an abstract infrastructure to offer basic services like distribution transparent communication primitives. The infrastructure is described using some supporting concepts, that give an internal engineering view of the middleware. However, these concepts are only used for description purposes, not as technology requirements. In the following, we consider the abstract infrastructure from two points of view, functions supported and the organisation of the supporting, hypothetical objects.

The infrastructure model is an abstract computing engine and does not intend to prescribe the implementation technique of the platform involved. Therefore, the infrastructure model allows implementation independent discussions on various services.

Functions

The ODP reference model supports distribution transparent communication with the four fundamental function classes: management, coordination, security, and repository functions [12, clause 12].

The *management functions* include

- node management function that controls processing, storage and communication within a node, i.e., nodes support time services, creation of channels between objects, location of interfaces, and management of processing threads;
- object management function that checkpoints and deletes objects;

- capsule management function that instantiates, recovers, reactivates, and deactivates clusters and deletes capsules; and
- cluster management function that checkpoints, recovers, migrates, deactivates or deletes clusters.

The *coordination functions* include

- engineering interface reference tracking function that monitors the transfer of engineering interface references between engineering objects in different clusters
- event notification function that records and makes available event histories (logs),
- checkpointing and recovery function that coordinates creation of cluster checkpoints (time, storage), and coordinates the use of the stored checkpoints in recovery of failed clusters,
- deactivation and reactivation function that coordinates cluster deactivation and reactivation using checkpointing for other reasons than failures,
- migration function that coordinates the migration of a cluster from one capsule to another,
- transaction function that coordinates and controls a set of transactions to achieve a specified level of visibility, recoverability and permanence.

The *security functions* include conventional security related services, i.e., functions for access control, security audit, authentication, integrity, confidentiality, non-repudiation and key-management.

The *repository functions* include

- trading function that mediates advertisement and discovery of interfaces, and
- type repository function that manages a repository of type specifications and type relationships.

The trading function presents an information repository that can be updated by independent information producers. Trading is not a mapping mechanism like name services. Instead, it resembles more directory services that allow attribute values to be used as a search criteria.

Trading activities are described through a trading community that represent the roles of 'importer', 'exporter' and 'trader' [16]. The object in trader role supports a repository of 'offers'. Each offer describes properties of an entity. The offers are produced by objects in exporter roles. When an exporter sends an offer for a trader to be stored, it is said to 'export'. When an exporter requests an offer to be deleted, it is said to 'withdraw' an offer. The objects in importer roles make queries to the offer repository, they 'import'. The import requests have a basic form that is similar to database queries: the request specifies criteria for selecting the offers to be included to the response. The objects that use the traded information need not necessarily be the importers or exporters themselves. The 'clients' and 'servers' that use the information

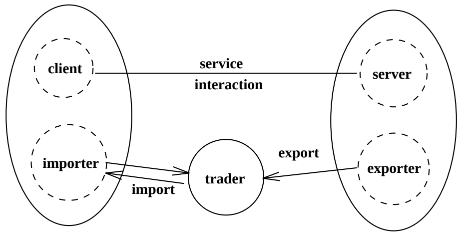

Figure 4.1: The trading community.

are therefore considered as separate roles. The communication between importers and clients, or between servers and exporters is not prescribed (but trading mechanism is recommended).

The ODP trading function is designed to mediate server interface offers. An interface offer contains an abstract service type name, an interface signature, and a set of attributes. The attributes can describe quality of service aspects or they can describe the supporting platform features of interest. The trading function specifies only the mediation mechanism and information structuring rules, not the interpretation of offers.

A service offer is a set of attribute name - value pairs. Attributes can have static values, set at the export time, or dynamic values, evaluated at import time.

The service offers are organised according to the service type offered. The service type also determines the attributes relevant for the offer.

For the binding process, the service type also determines the structure of the binding contract, as the export - import process forms a simple negotiation protocol sufficient for technical compatibility checks.

The attributes of interest for the computational component trader are: interface, location, protocol, QoS, binding type, id.

For binding purposes, the trader needs to carry some realtime properties. To start with the standard interface allows restricting the query processing by limiting resource consumption and span of the search, also, an upper limit for the time spent for the matching process can be given.

The *type repository function* supports operations for

- publishing type descriptions,
- checking conformance of two type descriptions,
- retrieving subtypes and super-types,
- translating types, and
- name management operations for types.

A type description is a tuple of type name, and a set of attribute names with acceptable value data type. A type can be equal to another type, subtype of it or supertype of it. The type hierarchy is not interpreted based on inheritance rules but resemblance of the descriptions.

The basis for the type repository activities is awareness of the target concepts to be supported. The network of target concepts is called the metamodel for that type system. For example, take target concepts object and operational interface. All operational interfaces must have a structure defined for calling and returning results. Also, an interface must be related to an object, but an object can have multiple interfaces. The metamodel naturally includes a target concept for relationships between any two types as well.

Objects

The management and coordination functions are supported by clusters, capsules, nodes, nuclei, channels, and basic engineering objects with the following definitions:

- "Cluster: A configuration of basic engineering objects forming a single unit for the purposes of deactivation, checkpointing, reactivation, recovery and migration" [12, clause 8.1.1].
- "Capsule: A configuration of engineering objects forming a single unit for the purpose of encapsulation of processing and storage" [12, clause 8.1.4].
- "Node: A configuration of engineering objects forming a single unit for the purpose of location is space, and which embodies a set of processing, storage and communication functions" [12, clause 8.1.7].
- "Nucleus: An engineering object which coordinates processing, storage and communication functions for use by other engineering objects within the node to which it belongs" [12, clause 8.1.6].
- "Basic engineering object: An engineering object that requires the support of a distributed infrastructure" [12, clause 8.1.1].

The concept of channel bridges between the ODP binding model and the actual infrastructure objects.

Configuration

The configuration of these objects is illustrated in Figure 4.2. A node contains a set of capsules. One of the capsules has a special role of being a nucleus of that node. All the other capsules are bound to the nucleus capsule to get the basic services. The nucleus is responsible for the node management function. Each capsule contains a set of clusters. One of the clusters has a special role of being a capsule manager. It is responsible for the capsule management function. For these purposes, it may request the nucleus for some services. The capsule manager, or another cluster, may perform the tasks of one or more cluster managers, with the corresponding functions. All objects are instantiated in such a way that they have prebound connections to the objects that offer the fundamental functions for them. This means that all objects are encapsulated into the clusters. Each cluster is bound to a cluster manager, which in turn is able to request the services of the capsule manager. The situation can be made more concrete by using an example: a segment of virtual memory containing data items can be considered to form a cluster, and a process represents a capsule; a computer with operating system and applications can be considered to form a node, and an operating system kernel represents a nucleus.

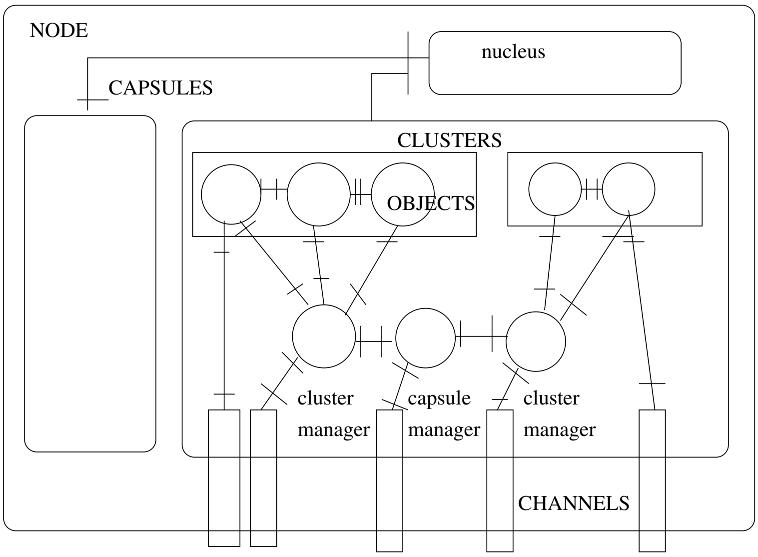

Figure 4.2: RM-ODP infrastructure objects [12].

As each cluster is indivisible, inter-object communication between objects in the same cluster may take place in any suitable method. The cluster is the smallest possible unit of migration and activation, so those actions cannot cause problems. Between objects in different capsules, either in same or different nodes, a channel is required for communication.

The granularity of these structures is arbitrary. When a system is described using these terms, the components that use non-ODP communication methods between each other are encapsulated within a cluster. A node can represent one computer or a set of them.

For the computational behaviour of objects, the channel creation is the most fundamental infrastructure service. Channels are created by nucleus objects in cooperation with each other. The channel supports distribution transparent interaction between engineering objects. This includes, for example,

- operation execution between a client object and a server object,
- a group of objects multi-casting to another group of objects, and
- stream interaction involving multiple producer objects and multiple consumer objects.

4.2 Using and extending the ODP reference model

The ODP infrastructure model gives a group of requirements for management and coordination functions for an open, distributed system. However, for example CORBA does not have such interfaces but in vendor dependent form. The reviewed concepts and functions give a framework for such recommendations. The management standards are essential for large scale interoperation, whether federations are formed manually between organisations, or whether automatic federation establishment is connecting services. The Pilarcos architecture expects such management functions to exist.

The Pilarcos infrastructure services introduce three fundamental extensions to the ODP reference model:

- type based, platform independent factories for service types and binding types;
- support for implicit complex bindings with federation transparency; and
- automated deployment of business services comprised of a configuration of service types and binding types.

The extensions are all related to each other. The third can only be achieved through the second, which requires the first.

The type based factories – application component deployers working with templates together with a type driven trader to select the templates and nodes to run on and binding factories – are based on the type repository capability of translating a type description to a set of equal but implementation-wise different templates separately for each node. The type based factories also support directly the programmer concepts of service type and binding type, thus enhancing the expressive power of programming languages.

The binding factory creates a channel section and the corresponding channel controller, illustrated in Figure 4.3. Channel controller contains the binding contract information that is formed by the business service deployers negotiating with the remote system. This information is used to monitor and manage the stubs, binders, protocol objects and interceptors in the channel.

The channel controller provides means to manage the binding in type abstractions and thus in platform independent form. This allows a generic infrastructure object to be formed for implicit establishment and control of the binding.

In the overall system architecture model, the application binder object is added on top of the channel layers. The application binder is responsible for initiating the business service deployment process, taking part into the federation establishment protocol, and pass on the service invocation when the receivers are deployed - all on behalf of the application object. The application binder also initiates the channel creation. The application programming interface supported should be able to provide an implicit biding model for complex bindings, and the model should be symmetrical for clients and servers. The invocation interface therefore supports a generic call routine for business services. The application binders at client and server side of a binding are symmetrical, i.e., the interfaces support operations needed in either case.

The application object also acts as a policy guard on each interaction related to the binding on which it is running.

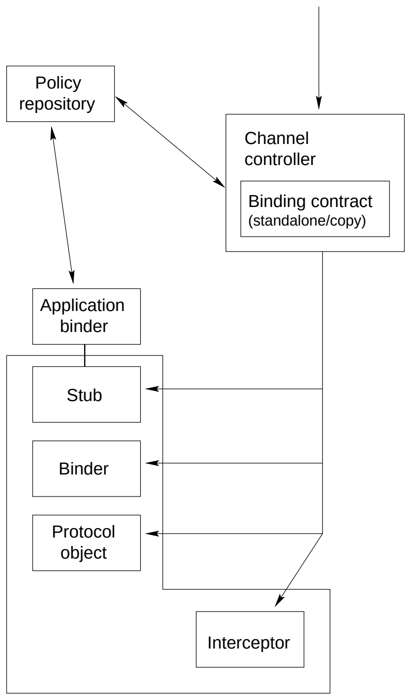

Figure 4.3: Channel section with channel controller, application binder and policy repository.

These extensions are located in the ODP infrastructure as follows. The type based factories become a part of a nucleus, while application binders are used between capsules either within a node or in separate nodes. The clusters represent for example CORBA components and capsules correspond for example to CORBA component assemblies. A node denotes an administrative domain contolled by a shared nucleus, for example, a distributed middleware platform of certain technology owned by an organisation.

4.3 Business service deployer

The business service deployer is responsible for deploying a business service requested by a client. The deployment process requires that the service providing elements are installed and available at the local system, in most cases not in a running state though. Each organisation has a private business service deployer.

The business service deployer works under the guidance of a named design and populates it by instantiating suitable component assemblies found via the computational assembly trader. The structure and dynamic properties of a design resemble much those of a business architecture description.

The process is platform independent, but exploits further services that are platform dependent but support a standard interface for the deployer to delegate tasks through. These services are traders, type repositories, and type based factories for component assemblies and bindings.

The population process is complex (in generic form, even NP complete). Each role in the design represent a placeholder for which a player has to be selected. However, the selections for each role are not independent from each other. In addition of finding a suitable candidate to fulfill a role, the process needs to check that the candidates in each role form a configuration that can be connected together with the available binding techniques.

In practice, the selection process is organised as follows. First, the business service deployer imports from the trader (simultaneously) a set of candidate component assemblies. The import criteria include

- service type that is either directly named in the role description or can be found via the local type repository;
- QoS requirements for the service; and
- policy rule requirements for the behaviour.

All these items are needed separately for all interaction partners the role is involved with.

In the import reply from the trader the deployer needs to receive

- identity of the candidate assembly for retrieving location information; and
- for each service interface, the technical requirements for bindings between the offered assembly and its peer in the interacting role.

The result of these import operations is a set of candidates for each role, over which a multicomponent optimisation task should be performed for finding the final configuration. The goal is to select a reasonably low cost binding to connect assemblies selected to roles. The selection process terminates with an acceptable solution for the whole design, and does not try to find the best solution.

The deployer forms for each binding in the design a set of alternative suggestions for the bindings (and roles involved) to be used by combining information from the offers to the roles.

In the type repository, a cost must be stored for each binding template. If a binding template involves a lot of interceptors and redirection, the cost is high; the lowest cost is for a template for local binding.

For each deployment of a design, an individual design controller is created to take care of the population process and to reflect the life cycle of the created community later on. The controller is needed to reassign assemblies to roles that become empty, or to move the whole community to a new epoch.

More technically, the business service deployer supports the operation for preparing a business service. The operation needs the parameters service type, business architecture description in which to play a role, the role to play. The business service deployer algorithm is presented in Figure 4.4.

- *receive prepare service with parameters*
- *create new policy context to the local policy repository for the current usage of a business architecture*
- *retrieve design*
- *create design controller*
- *for each role in the design*
  * *import a component assembly; selection criteria: suitable external interfaces, suitable binding type support, accepts the current policy context*
  * *feed imported information to application component deployer which returns an instantiated component assembly and references to its free interfaces*
- *for each binding in the design*
  * *create application binder for each endpoint of the binding*
  * *bind the application binder to the primary service interface of the component assembly*
  * *tell each application binder the binding type and the other application binders involved for the application binder to create bindings*
- *in case of federation send prepare federation contract message as response to the initiating application binder*

Figure 4.4: The business service deployer algorithm.

4.4 Application component deployer

The application component deployer is a template based factory for application components. An application component is available as a platform dependent component assembly package and is stored to a platform specific component repository.

The deployment process allocates resources for running the assembly and keeps record on the location of individual objects within the assembly.

This service is platform dependent; a more detailed design is described in further documents on Pilarcos.

4.5 Binding factory

Binding factory is a type based factory for channels. The binding factory gets a request for establishing a binding i.e. at least a channel controller. The factory is given a reference to the binding type in type repository. For each type there is defined a set of templates referred to (component assemblies for implementing bindings). The binding factory chooses the one that is suitable for its own platform environment. Finally, the binding factory utilises platform dependent factories (like application component deployers) to instantiate channel components.

Figure 4.5 shows the actions taken by the binding factory.

- *search a channel type from the type repository; the channel type should match the binding type and the platform requirements*
- *create a channel controller with reference to current policy context*
- *create a channel section for each endpoint of the channel within the administrative domain of this binding factory*
- *feed the free interface locations to the appropriate channel controllers*

Figure 4.5: The binding factory algorithm.

4.6 Binding components

The infrastructure exploits two binding related component types that support complex bindings and their implicit management. These components are application binders and channel controllers.

Channel controllers are essential for any complex bindings; they provide management facilities for the channel and allow changes between modes of abstract liaison and a resource reserving implementation of that liaison.

Where present (e.g., between component assemblies and where a channel extends over administrative domain boundaries), application binders take the responsibility of using the channel controller interface for managing the binding on behalf of the applications. The application binder responsibility is to offer implicit creation and management of complex bindings, with transparent federation support.

4.6.1 Application binder

Application binder is created by business service deployer for each binding the component has according to the computational design.

The application programming interface supported should be able to provide an implicit biding model for complex bindings, and the model should be symmetrical for clients and servers. The invocation interface therefore supports a generic call routine for business services. For implementing this operation, the application binder is responsible for initiating the business service deployment process, taking part into the federation establishment protocol, and pass on the service invocation when the receivers are deployed - all on behalf of the application object.

The generic call routine (operation prepare service)

- specifies the called entity in terms of business service in question,
- specifies the suggested policies to be used for that business service,
- expresses the client view of the computational interface and computational binding model to be used for interacting with the selected business service,
- gathers information for constructing the necessary engineering machinery for fulfilling the tasks of the binding between the business service interface and the

caller; much of this is available in the container of the calling object, except of computational, information and enterprise level interceptors;

- initiates instantiation of the binding machinery and also of a controller object for that machinery; that machinery may involve federation;
- initiates the business activity through the constructed binding;

Figure 4.6 captures the call routine algorithm.

- *import a business service from the business service broker; the imported information contains location for the corresponding business service deployer in the service providing system*
- *send prepare service to the identified business service deployer*
- *respond to the negotiation with the peer application binder*
- *call binding factory and pass on information about peer application binder location*

Figure 4.6: The application binder algorithm for preparing a business service.

The application binders at client and server side of a binding are symmetrical, i.e., the interfaces support operations needed in either case. The application binder also acts as a policy guard on each interaction related to the binding on which it is running. The invocation process should involve security considerations [29, 31].

The application binder supports also a second interface, an interface for application binders to talk to each other. The interface is used both for creating a federated binding and for passing on communication between the clients of application binders. Figure 4.7 captures the mutual application binder communication.

- *receive/send message prepare federation from the server's application binder; the message contains suggested federation contract information for the established federation*
- *check the consistency of the suggested contract with the policy repository and if consistent, enter the federation contract to the policy repository; if the suggestion is inappropriate, send a contersuggestion to the server's application binder*

Figure 4.7: The application binder algorithm for federation establishment.

The federation establishment phase involves creation of federation contracts to the local policy repositories at each involved domain. The federation contract includes

- service type,
- binding type,
- test for contract breaches,

- protocols for failure recovery and contract breach sanctions, and
- contract terminating behaviour.

After the binding has been established, the application binder has the responsibilities of

- sending (receiving) operation invocations;
- checking with policy repository whether an interaction is acceptable; and
- checking when there is an obliged interaction to take initiative on.

These actions are part of the mutual application binder protocol.

4.6.2 Channel controller

The channel controller is responsible for creating, deleting, monitoring and controlling the channel between application binder interfaces at each party. The channel structures are platform dependent and thus also the control interfaces. There is a need to have a wide variety of channel templates and the corresponding channel controller templates available.

The channel controller supports operations for

- changing contract and
- terminating contract and binding.

It uses operations related to platform dependent management interfaces of stubs, binders, protocol objects and interceptors.

The channel controller contains the federated binding contract that is negotiated by the application binders. The binding contract includes:

- abstract service type identifier, that allows each administrative domain to access the related parts of the local type system,
- collectively selected behaviour,
- technical descriptions for interface signatures, which can be differently selected at each administrative domain,
- communication protocol,
- channel class that expresses the type of channel to be instantiated, allowing each administrative domain to access local details for stub, binder and interceptor objects,
- service type specific QoS agreement,
- failure detection and recovery protocols, failure defined as not being able to meet the QoS agreement,

- remuneration protocol,
- channel control interface reference separately for each administrative domain, and
- reference to the corresponding role context in the policy repository; this also indirectly indicates the corresponding community context.

4.7 Repositories

The repositories collect together components and meta-data about them, type information about services and interfaces, templates for instantiation, architecture descriptions and design patterns, etc.

The sources of these items is an open community of developers; each of them in principle free to register a new information item to a globally accessible repository. Multiple versions for the same item can be simultaneously available, and it is up to the user do decide whether to rush for the newest or stick with more experience. A gradual process of deprecation of older versions is naturally needed, and finally deletion of deprecated items from the markets.

Security issues are essential for all those repositories that are used in the binding process. These repositories should accept new items from only trusted providers, and have means to authenticate the parties accordingly.

The repositories are best suited to be supported separately by each organisation. This allows fast access to locally supported items, and a natural gateway to the global market via federation of the same type of repositories.

4.7.1 Business service brokerage

The business service broker, or a set of interworking business service brokers worldwide, implement an open marketplace for business services. Various organisations are able to export their offers to the broker and others able to import metainformation about those services.

Various constellations of broker servers supported by organisations are here omitted. There are multiple alternatives with differing scalability properties; however, the broker constellation is not in focus here.

Within the Pilarcos runtime architecture, the metainformation stored in the broker captures information needed to retrieve a set of applicable service providers in order to populate a business level architecture description.

Attributes describing the services within the broker are dependent on the service type they represent, but focus on business decisions. Examples of appropriate attributes include service type, price of the service, and framework for quality of service attributes. The full information model required at this spot is already described as a contract framework in Chapter 2.

The exploiter of broker is the federated binding protocol, thus, the fundamental requirement for each business service offer it to include an address for the service deployer able to instantiate the offered service.

The communication channels to the business service broker are fixed. In Pilarcos architecture, the component running the federated binding protocol (application binder, see Section 4.6.1) holds the necessary broker address(es) as part of its configuration information.

4.7.2 Policy repository

A policy is a behaviour restricting decision that is decided by a controlling party and enforced on the controlled objects; a contract is a behaviour restricting decision that is mutually agreed by several peers that have no controlling power over each other.

A policy framework captures the areas on which such policy decisions can and should be made. The policy framework effectively guides and restricts application programming: the behaviour of applications is required to take into consideration the policy rules defined within their runtime environment.

Policy rules for a system include both rules for application area specific behaviours and for system services. Each service type is naturally associated with a possibly different policy framework.

A policy repository is an organisation-wide hierarchy on behaviour restricting decisions. The hierarchy is structured by levels starting from business areas, stepping down to constellations for supported services and individual activities. The hierarchy has branches for each currently running service, and at activity level, especially for federations where contracts between organisations become stored.

The policy repository has a management interface for

- entering policy frameworks,
- structuring the policy repository hierarchy for organisation, service and interaction levels using the policy frameworks,
- adding and removing policy objects or contract objects, i.e. contexts, at each level, and
- changing the contents of an existing context.

The contract objects each represent a policy context in which an application object is doing its interactions. The policy context includes

- an identifier;
- policy decisions in effect;
- an interface for changing the current rule; possibly involving a renegotiation protocol with references to involved parties;
- references to contexts this context is inheriting rules from; for example, organisation-wide rules are not copied but inherited via references.

The hierarchy of policy context include

- policies to govern all applications and infrastructure services within the organisation;

- policies to govern certain service type;
- policies to govern certain binding type; and
- policies to govern a currently valid federation.

This view can be compared with contract aware components [6].

The organisations maintaining policy repositories range from companies to individual users. Also, organisations can form a hierarchy. For example, a user can inherit policies from the company by which he or she is employed.

4.7.3 Design repository

The design repository contains the designs, i.e., computational architecture descriptions used within the organisation. It supports operations for storing, retrieving and deleting designs.

The structure of design and the routines exploiting the design information are discussed in Sections 4.3, 3.2.1, 3.5.1, and 3.5.2.

Designs are in many ways close to architecture descriptions of traditional distributed systems, and can therefore be modelled with similar methods. For formal modelling of architectures, a number of architecture description languages (ADLs) have been developed. The central modelling concepts in typical ADLs are [26]:

- *components*, units of computation and data storage;
- *connectors*, which represent interactions between components and rules for these interactions; and
- *configurations*, topologies of components and connectors representing a system.

These concepts are suitable for modelling the main elements of designs. The placeholders for component assemblies can be represented as components, the binding types as connectors, and whole designs as configurations. The same concepts can also be applied to business architectures, which makes refinement from enterprise to computational level easier.

An important aspect of architecture description in Pilarcos is modelling of behaviour. By behaviour, we mean the abstract protocols obeyed by components and connectors. On the business architecture level, the need for behaviour modelling is obvious, since it must be possible to express legal workflows and policies related to them. It is also needed for designs to ensure compatibility of components, proper functioning of the architecture and adherence to the business architecture. Some ADLs support behaviour modelling. These ADLs are typically based on a formal semantic model, such as the π-calculus [28]. The π-calculus is especially interesting in that it supports mobility, which manifests in the ADLs based on it as support for dynamic configurations.

Most ADLs have not been created with late binding in mind. They therefore do not offer direct support for assignment rules (Section 2.2). Assignment rules can, for instance, state dependencies between object identities and roles which must be respected when populating the roles with objects. The constraints offered by some ADLs come close, but cannot be used as such. Other criteria included in the assignment rules are interface and behaviour compatibility, suitability of non-functional attributes (regarding e.g. QoS), acceptance of current policy context, and technical compatibility.

We may summarise the desirable features for an architecture description language in Pilarcos as follows:

- implementation independence;
- refinement, at least in the form of compositionality;
- modelling and analysis of behaviour;
- support for dynamic configurations;
- separation of provided and required interfaces for composability;
- freely definable connectors, with a potentially unlimited number of endpoints; and
- constraints for types and identities.

ADLs that are particularly interesting from the Pilarcos point of view are Darwin [24], Olan [3], LEDA [7] and Wright [1].

Of these, Darwin is a fairly basic configuration language with support for some dynamism. A CORBA-based execution environment [25] has been developed for it.

Olan is a distributed component environment that uses an ADL based on Darwin. It is perhaps the most complete component environment available. Like Darwin, it does not have any facilities for modelling or analysing behaviour.

LEDA is a new ADL based on the π-calculus, providing for behaviour modelling with support for dynamic configurations and refinement. However, at the time of writing, no tools were publicly available for it.

The Wright ADL has good support for static analysis, and unlike Darwin and LEDA, it clearly separates components and connectors. Wright supports behaviour descriptions based on CSP [9], as well as architectural families (styles) and constraints for them. As a downside, it does not support any dynamism in configurations.

The problem with ADLs is that they are relatively immature, as they stem from academic settings and have not been extensively applied to real-world problems. Tool support is most often sparse, and different ADLs are largely incompatible. It seems that no one ADL as such is usable for Pilarcos architecture descriptions. Still, they are valuable sources of experience for developing information models and notations for architecture descriptions in Pilarcos.

4.7.4 Computational component assembly trader

In the service deployment process within each organisation, a design is populated with component assemblies. In this population process a trader is used to choose between available assemblies, thus, the trader collects together logically similar component assemblies that are realised on different platforms.

The exports to the trader originate from the organisation supporting the trader and a prerequisite is that the components exported are already installed and registered to the assembly repository.

The traded offers include

- service interfaces of the assembly,
- service types supported by the assembly,
- QoS information for each service type supported,
- technical requirements for the bindings needed to communicate with this assembly, and
- policy rules applied or applicable to this assembly.

4.7.5 Type repository

The type repository services [15] mediate type information that is available at run-time and at design time. The main users for this kind of metainformation are traders (and brokers), and type based factories.

For type repositories in Pilarcos architecture, the essential function is to support information on relationships between types and templates. Types serve as classifiers for somewhat similar templates. A type description is used in negotiation protocols instead of templates; it represent a global understanding of an entity. A template is a relevant implementation related version of the corresponding type; a template is always associated with a known platform. Thus the type repository maps together the negotiation language and the implementation instructions across various technology domains.

To be more precise, traders use type repositories for checking conformance between object interfaces. Traders could include the type management facilities themselves, but separation of these tasks to an independent module provides flexibility of configuration and supports independent evolution of the type systems. Type based factories use type repositories for selecting a suitable implementation to instantiate for a task. For example, a binding factory uses type repository for matching generic binding types to specific channel configurations.

The target concepts described in a type repository include service types, component types, binding types and their relationships to designs, component templates and channel templates. The actual templates are stored to assembly repository; the relationships stored only give references to them. Different type systems may have different sets of target concepts. For the ODP and CORBA target concepts and their comparison, see [15].

A service type definition includes

- an interface description interpreted as an abstract interface gathering only the common and essential features of the real services the type represents (it is a predicate inducing a group of technically slightly differing services);
- definition of epoch changes if applicable;
- QoS attribute framework related to this service type;
- policy framework related to this service type.

A binding type definition includes

- set of roles that can be bound together, especially that of the requesting client,
- channel types that can be used,
- attributes for choosing between channel types, i.e., selections for distribution transparencies, QoS attributes and security attributes.

The channel type in turn includes

- references to a set of channel templates, denoting a configuration of binder, stub and protocol objects, and
- a reference to a channel controller template.

The templates are stored to the assembly repository.

As the templates representing the same type fall onto multiple technology domains, the relationship between the type and the template may be decorated with some translation information or even references to transformation components, i.e., interceptors. Thus, the federated binding process has support for automatic interceptor insertion to communication channels, i.e., the architecture gains automated interoperability.

In addition to local relationships between various templates for a type, there are relationships between type representations in separate type repositories. Mainly, these relationships are equalities, or require simple transformations like mapping of the type name from one repository to another. The purpose of these relationships is to facilitate federation of type repositories, thus enlargening the domain on which type conformance testing of interfaces can be performed.

A common storage service is to be used as storage for the type repository, extended with new target concepts required by Pilarcos architecture. These are listed in Section 4.7.6.

4.7.6 Common storage service for repositories

The Pilarcos architecture introduces a number of metainformation repositories, each of which is supported with a metamodel.

The metamodel captures some target concepts, structures for storing them and rules for their relationships. The requirements are equal to those on CORBA MOF service [35]. In fact, a MOF server with proprietary metamodel suits the purpose.

The repositories in Pilarcos architecture needing such a common storage service are the business service type repository supporting business service brokers, type repository that supports a) component trader in the deployment process, b) type based binding factory, and c) type based component assembly factory, and finally design repository. We can capture the needs of these services into items needed for instantiation and for binding.

For service instantiation purposes, the Pilarcos repository service need to support hierarchy of target concepts

- business architecture description,
- service type,

- design,
- interface type, and
- interface.

For binding establishment purposes, the Pilarcos repository service need to support a hierarchy of target concepts

- binding type, that defines the set of computational components and the protocol between them as a set of interfaces,
- channel type, that defines the distribution transparencies supported, QoS attributes to be agreed and security attributes involved,
- channel template that contain binder, stub and protocol object templates; and
- channel controller template.

The last two form a pair that is stored to the assembly repository and only referred to from the type repository.

A more detailed design shall define structures for storing these target concepts.

4.7.7 Business type repository

Business type repository is the type repository supporting business service brokerage. It has the same implementation technique as any other type repository, but its main responsibility is to store business architecture descriptions.

Each trader or broker in the system is supported by a type repository, but a single type repository possibly serves multiple traders or brokers.

4.7.8 Component assembly repository

Computational components can be produced either by Pilarcos tools or some other method. The component concept in itself is technology independent. However, the engineering aspects and communication capabilities reveal the technology choice (like EJB [40], CCM [37]).

A component includes

- a set of engineering object templates or references to such templates within a known repository;
- configuration instructions, i.e., rules for locating the components at instantiation time;
- meta-data describing the interfaces supported, strategy choices supported, management interfaces supported, references to architecture roles supported, properties like QoS attributes, etc.

In Pilarcos architecture, a component assembly is selected to be a primitive building block. An assembly capsules technology domain dependent aspects of configuration and communication to a package that is run within a single computer or within a distributed computing node under single administration. The assembly can even encapsulate legacy software.

A native Pilarcos software component is managed with identical procedures and metainformation as, for example, a CORBA component assembly.

4.8 Management services

In general, management services in a distributed system environment cover

- failure management, including failure discovery and recovery, and instrumentation of managed objects with failure management mechanisms;
- configuration management, including installation, instantiation, activation and deactivation of various object constellations, and also including dynamic changes within these constellations;
- accounting management;
- performance management, including instrumentation of communication mechanisms and computing mechanisms with performance monitors and controllers that are able to select configurations for various performance needs;
- security management.

The platform management services (listed in Section 4.1) answering to these needs are expected to have defined interfaces on all platforms on which Pilarcos services are implemented. For example, in CORBA environment, it is necessary to develop further facilities on this area as the solutions provided are vendor dependent.

The entities subject to the listed management services include the distributed platform itself (e.g., ORB), and various components of it (e.g., object adapters, processes, and other resources in the system), connections, and objects in general.

An essential management service is that of supporting autonomous domains and their management information in the system. This aspect is addressed by Pilarcos brokers, service deployers, component traders, and type based factories able to differentiate between various platform types.

The application management services are effectively created by the additional Pilarcos services. The same functions now apply specifically on for example application peer objects, interfaces, designs, and interworking and federation relationships.

4.9 Development issues

The Pilarcos project is by choice developing its pilot tools and services in the CORBA environment. However, the model is equally well applicable to other component systems, like Java and EJB (Enterprise Java Beans). In fact, one of the goals for the presented model is to provide an automated method of creating bridges across technology boundaries.

A major concern is that of security. The architecture approach uses active components that themselves include features for security [23]. In addition to that, further trust model is needed between repositories that store components. Either a non-trusted repository can provide defective or destructive components, or a trustful repository can accept to involuntarily mediate such components. A qualification system [8] is required to form a network of trusted sources for services [2].

Chapter 5

Conclusion

The Pilarcos architecture supports large, inter-organisational component systems. Essential aspects supported include

- globally understood business architecture descriptions as a basis for cooperation agreements that support interorganisational applications;
- a protocol for sovereign systems to dynamically form federations based on the business architecture descriptions;
- a locally implementable mechanism for deploying services needed as part of the business architecture;
- a locally administrable mechanism for managing heterogeneity and distribution of service implementations within organisational IT systems;
- support for flexible evolution of services; this leads to changes in the way design time and runtime actions are seen and how they interact;
- more powerful programming primitives offered for programmers, especially for managing complex communication relationships.

The Pilarcos architecture requires a set of new middleware services to be introduced. These can be implemented on each applicable platform like any application program, and therefore, no platform is really contradictory for this design. However, most platform architectures, like CORBA, do not support all concepts essential for the Pilarcos architecture. For example, explicit difference between type and template concepts, or differentiation between client and server role interfaces are in most cases missing.

References

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

[2] Arshad, K., Atif, Y., and Siyal, M. A CORBA based Framework for Trusted E-commerce Transactions. In *IEEE Proceedings of Third International Conference on Enterprise Distributed Object Computing* (1999).

[3] Bellissard, L., Palma, N. D., and Féliot, D. The Olan architecture definition language. Tech. rep., INRIA, 2000. C3DS Technical Report 24.

[4] Berry, A., and Kong, Q. A General Resource Discovery System for Open Distributed Processing. In *Open Distributed Processing; Experiences with distributed environments, Proc. of the 3rd IFIP TC6/WG 6.1 International Conference on Open Distributed Processing* (Brisbane, Australia, 1995), Chapman and Hall, pp. 79–90.

[5] Berry, A., and Raymond, K. The *A1*√ Architecture Model. In *Open Distributed Processing; Experiences with distributed environments, Proc. of the 3rd IFIP TC6/WG 6.1 International Conference on Open Distributed Processing* (Brisbane, Australia, 1995), Chapman and Hall, pp. 55–67.

[6] Beugnard, A., Jezequel, J.-M., Plouzeau, N., and Watkins, D. Making components contract aware. *IEEE Computer 32*, 7 (July 1999), 38 – 45.

[7] Canal, C., Pimentel, E., and Troya, J. M. Specification and Refinement of Dynamic Software Architectures. In *First Working IFIP Conference on Software Architecture* (1999), Kluwer Academic Publishers.

[8] Ghosh, A. Certifying Security of Components Used in Electronic Commerce. http://www.rstcopr.com.

[9] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1990.

[10] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 1: Overview*, 1996. IS10746-1.

[11] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 2: Foundations*, 1996. IS10746-2.

[12] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 3: Architecture*, 1996. IS10746-3.

[13] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 4: Architectural Semantics*, 1996. IS10746-4.

[14] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing*, 1996. IS10746.

[15] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Type repository function*, 1997. IS14746.

[16] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trading function. Part 1: Specification*, 1997. IS13235-1.

[17] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – ODP Interface References and Binding*, Jan. 1998. IS14753.

[18] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Enterprise Language*, June 2000. CD13235.

[19] Kähkipuro, P., Marttinen, L., and Kutvonen, L. Reaching Interoperability through ODP type framework. In *TINA'96 Conference: The Convergence of Telecommunications and Distributed Computing Technologies* (Aug. 1996), VDE Verlag, pp. 283 – 284. Extended abstract.

[20] Kähkipuro, P. *Performance Modeling Framework for CORBA Based Distributed Systems*. PhD thesis, Department of Computer Science, University of Helsinki, 2000.

[21] Kitson, B. Intercessory Objects within Channels. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 233 – 244.

[22] Lepreau, J., Hibler, M., Ford, B., and Law, J. In-kernel servers on Mach 3.0: Implementation and performance. In *Third USENIX Mach Symposium* (USA, Apr. 1993), pp. 39 – 55. Also http://www.usenix.org/publications/library/proceedings/mach3/lepreau.html.

[23] Liu, D., and Wiederhold, G. Chaos: An active security mediation system.

[24] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference ESEC'95* (Barcelona, Sept. 1995).

[25] Magee, J., Tseng, A., and Kramer, J. Composing distributed objects in CORBA. In *Proceedings of the Third International Symposium on Autonomous Decentralized Systems* (Berlin, Germany, 9–11 Apr. 1997), IEEE, pp. 257–63.

[26] Medvidovic, N., and Taylor, R. N. A framework for classifying and comparing architecture description languages. In *ESEC/FSE '97* (1997), M. Jazayeri and H. Schauer, Eds., vol. 1301 of *Lecture Notes in Computer Science*, Springer / ACM Press, pp. 60–76.

[27] Meyer, B., and Popien, C. Flexible management of ANSAware applications. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 271 – 282.

[28] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes. *Information and Computation 100*, 1 (Sept. 1992), 1–77.

[29] Object Management Group. *CORBA Security*, Dec. 1995. Document 95-12-1. Also http://www.omg.org/docs/1995/95-12-01.ps.

[30] Object Management Group. *OMG Business Application Architecture*, Mar. 1995. http://www.tiac.net/users/jsuth/oopsla/bowp2.html.

[31] Object Management Group. *Common Secure Interoperablity*, July 1996. Document orbos/96-06-20. Also http://www.omg.org/docs/orbos/96-06-20.ps.

[32] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, May 1996. OMG Document No. 91-12-1 (Revision 2.1).

[33] Object Management Group. *Common Facilities RFP-5: Meta-Object Facility*, 1997. OMG TC Document cf/96-05-02.

[34] Object Management Group. *UML Semantics*, Sept. 1997. OMG Document No. ad/97-08-04 (Revision 1.1.). Also http://www.omg.org/pub/docs/ad/97-08-04.ps.

[35] Object Management Group. *Common Facilities RFP-5: Meta-Object Facility*, 1998. OMG TC Document cf/96-05-02.

[36] Object Management Group. *OMG, Workflow Management Facility*, 1998. OMG Document bom/98-06-07.

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

[38] OSKIEWICZ, E., AND EDWARDS, N. A Model for Interface Groups. Tech. Rep. APM.1002.01, APM, May 1994. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1002.01.html.

[39] SLOMAN, M., MAGEE, J., TWIDLE, K., AND KRAMER, J. An Architecture for Managing Distributed Systems. In *Fourth IEEE Workshop on Future Trends of Distributed Computing Systems* (Lisbon, Portugal, Sept. 1993), IEEE Computer Society Press, pp. 40 – 46.

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

[41] TORVALDS, L. Linux: a Portable Operating System. Master's thesis, Department of Computer Science, University of Helsinki, Jan. 1997. Report C-1997-12.

[42] VOGEL, A., KERHERVÉ, B., VON BOCHMANN, G., AND GECSEI, J. Distributed Multimedia Application and Quality of Service – A Survey. *IEEE Multimedia 2*, 2 (Summer 1995), 10 – 19.

ISSN -
ISBN -
Helsinki 2001