Quality of Service Aware Binding in Open Systems

Øyvind Hanssen

A Dissertation for the degree of Doctor Scientiarum

University of Tromsø
Faculty of Science
Department of Computer Science

January 2008

ISBN 978-82-92461-83-9
Computing Reviews (1998) Classification: C.2.4, D.2.1, D.2.12, D.3.1, I.2.4
Tromsø 2008

To my Parents, Harald and Randi

and my children, Ellen Miranda and Hanne Veronika

Abstract

This thesis addresses negotiation of *bindings* in QoS aware open systems and in particular how to characterize possible contracts, requirements and capabilities of heterogeneous environments. Our approach is (1) to use trading of policies as the principle for negotiation and (2) the use of declared or rule-based conformance for QoS statements to be used in negotiation and (3) a middleware binding framework supporting negotiable bindings.

A *policy* is an architectural entity which encapsulates a potential QoS contract plus a resource and implementation configuration to enforce the contract. The contract part (profile) consists of an obligation to be matched with requirements and an expectation to be matched with descriptions of the environment. A policy may encapsulate how implementation components and properties of environments are combined to achieve a QoS level. *Policy trading* is the proposed principle for negotiation. Potential contracts may be orthogonal to interfaces and interface types, and are associated with some trading service. Negotiation is essentially to collect requirements and environment descriptions from participants and match these with policies.

We investigate an approach to contract matching based on declared conformance. An important contribution is the development of a declarative expression language for profiles, requirement and environment descriptions used in negotiation at run-time. We propose to define conformance as rule bases. Such rule bases can be compiled to efficient conformance checking code. We propose two composition operators to combine expressions at run-time and develop the foundations of a generic conformance checking algorithm for profile expressions.

A prototype binding framework is developed, supporting activation of bindings, pluggable binder and activator components. The framework is extended to support negotiation using policy trading and profile expressions. The approach is validated by examples and proof of concept implementations. A profile model compiler, a policy trader and the binding framework are implemented and applied to example applications.

Acknowledgements

The work has been carried out over a rather long time. The work of this thesis started as a research fellowship on the DIME project (Distributed Interactive Multimedia Environment), supported by the Norwegian Research Council.

Professor Frank Eliassen has been the supervisor of this project. His advice, support and patience have been essential.

In 1997 I was seconded to the ANSA Phase-III project at APM Ltd. in Cambridge UK. This stay was supported by a NATO Science fellowship through the Norwegian Research Council grant 116590/410. In this phase I was involved in the ANSA FlexiNet project which was an early experiment in reflective middleware. Since then, much research has been done in the area of reflective middleware. Thanks to Andrew Herbert, Richard Hayton and the rest of the ANSA team for their hospitality.

Also thanks to Gordon S. Blair and the other researchers from the universities in Lancaster, Oslo and Tromsø who was involved in the CORBAng project in 1998 and 1999. I would also like to thank Gregor Kiczales and the other mentors and Ph.D. students at OOPSLA'99 Doctoral Symposium.

The time at Simula Research Lab in 2001 was useful and inspiring. Thanks to Olav Lysne for clarification on some formal issues. I would like to mention Richard Staehli, Viktor S. Eide, Jan Øyvind Aagedal, and others. I would also thank Agder University College, and the Open Systems group (I was there from 1998 to 2001). And at last I would like to thank the Department of Computer Science in Tromsø and the students and the staff Arctic Beans project in particular. Thanks to Anders Andersen, Randi Karlsen, Anna-Brith Arntsen, Åge Kvalnes and Tage Stabell Kulø and many others for motivation.

I would like to dedicate this work to my parents, Randi and Harald and my children Ellen Miranda and Hanne Veronika.

Table of Contents

List of figures

List of tables

Chapter 1.
Introduction

1.1. Motivation

Open systems are, according to Abadi and Lamport [Abadi94], systems which are to be run in "*environments neither they or their implementers control*". Open systems are typically specified in terms of components and relationships between these; i.e. components use each other's services, and reusable components (possibly produced by different implementers) can be added or replaced in a running system. As the complexity and size of computer applications grows, so does the importance of being able to compose distributed applications from various reusable components. Principles like data abstraction and polymorphism, service oriented architectures and object technology like e.g. OMG CORBA [OMG96], represent significant contributions to interoperability. Middleware essentially realise distribution transparencies and abstracts over platform specific services and resources and is therefore helpful in overcoming heterogeneity and distribution.

Furthermore, it may be desirable to re-use services in different contexts, which were not necessarily foreseen in the first place. From a user's point of view, it is desirable to have services which can *adapt* to varying needs, usage patterns or available computing and communication resources. However, it may be challenging to provide one implementation of a given abstract task or service which performs satisfactorily with more than a limited range of environments or usage patterns. Implementations often make implicit assumptions on the *extra-functional* behaviour (also termed quality of service) of the environment they run in. For instance, a component using the services from another may assume that interactions happen reliably or in bounded time. If such assumptions do not hold, the application may not work as expected. This may be particularly problematic when user requirements, resource availability and quality of service from the underlying platform may change dynamically.

Therefore, we need not only abstraction over platforms and implementations (as provided by traditional middleware), but also the ability to expose information about their extra-functional behaviour, not only at design time but at run-time as well since implementations and resource availability is not expected to be static. Systems having such expression and negotiation capabilities are termed *quality of service aware* [Frølund98b]. A common and widely accepted approach is to specify extra-functional assumptions and promises as *QoS contracts* [Beugnard99]. In an open system we may need to validate or negotiate contracts at run-time as configurations change.

Research on contractual QoS (e.g. [Loyall98a, Frølund98a]) tends to associate QoS contracts with interfaces, implementations, or the deployment of components. In open systems, in particular in service oriented architectures where components are loosely coupled, we find it useful to focus on *bindings* when considering QoS contracts, since this represents a level of indirection between the use of a service and how to implement its behaviour. In the context of distributed object middleware, a *binding* has typically to locate the implementation of a service, set up communication protocols, allocate resources, etc. in order to enable interaction. All these choices can affect the extra functional behaviour of the service as observed by the users of the binding and therefore, a desirable property of a middleware infrastructure is the ability to configure bindings dynamically in order to adapt to the situation, including QoS requirements, resources etc., or simply decide if a contract is valid or not. There has been much progress in this area of middleware research and in particular in how to control QoS via bindings [Blair00, Kon02, Blair04], but it is still not very clear how to bridge this with contracts and negotiation.

1.2. Problem and objectives

An overall theme of this thesis is how to design and configure components in open systems, such that the applications built from them are able to adapt to different and changing extra-functional requirements as well as different and changing environments. More specifically, our goal is to investigate how to support QoS aware *binding* to some abstract service at run-time. This includes how to define and negotiate *contracts*, certain aspects of how to enforce contracts, and how to expose varying platform capabilities and other environmental properties in a way that facilitates negotiation between heterogeneous components. This report describes work performed over a long time, where we have looked into various aspects of this problem area. In summary the following key questions have been driving our work:

1. How and to what extent extra-functional aspects can be separated from service interfaces or implementations and made negotiable.
2. How to characterise and negotiate (configurations of) such aspects, and how to do this in an efficient, scalable, interoperable, extensible and composable way.
3. How such characterisation and negotiation can be supported by an infrastructure.

1.2.1. Focus

QoS aware binding can involve several problem areas. This topic is related to the rather large area of Quality of Service management as well as flexible middleware. Our choice of focus includes the following:

- We focus on QoS expression and contract evaluation at run-time rather than QoS modelling or specification of systems. However, it is relevant to consider certain aspects of QoS modelling in order to define potential contracts and meta-information for run-time expressions.
- We focus on satisfaction of requirements rather than on how to maximise QoS or how to optimise resource utilisation. Our view on negotiation is limited in that we do not focus on how to agree about alternative QoS if the initial requirements cannot be met.

- We focus on static QoS management (establishment of bindings) rather than dynamic QoS management (maintenance of QoS).
- We are interested in how a middleware level infrastructure can support the implementation and deployment of various techniques for enforcing or maintaining QoS, but we do not go deeply into the specific techniques themselves.

1.2.2. Key issues and requirements

Orthogonality and negotiability

Extra-functional behaviour should in principle be *orthogonal* to service interface type[1]. It should be possible to make such behaviour *negotiable* at run-time and ideally, the implementation of such behaviour would be a concern of the binding. A question is how and to what extent such negotiability can be supported. Furthermore, orthogonality with respect to implementation components is a related issue but this seems to be more problematic. A service implementation may not always be a single software component but in some cases it is more of a composition of several aspects, and the composition of these is a result of negotiation. Here, we see a problem in that implementation parts make implicit assumptions about how the others behave.

Negotiation and QoS statement

We are interested in efficient methods for finding and agreeing on how bindings should be configured and what contracts they should be associated with. We refer to such a process as *negotiation* since the overall goal is to reach agreements between possibly autonomous parties and since it may involve exchanging statements (offers and requirements). *Interoperability* is an important issue; it is necessary to establish a common understanding of QoS statements and contracts amongst negotiating parties. These parties should be allowed to be heterogeneous and developed independently of each other.

First, we need to facilitate the search for policies. This may also involve making trade-offs between conflicting goals and may be a complex task since they may involve choices along many dimensions where the choices may interact. In theory, the search for policies may be viewed as a constraint satisfaction problem [Kumar92] which can be complex or even NP-hard. Furthermore, the mapping between the constraints in terms of QoS requirements and environment descriptions and constraints in terms of possible choices for each dimension is not necessarily straightforward.

Second, to enable such negotiation, it is important that the QoS and resource information to be expressed and exchanged is in a form that supports efficient *evaluation* to see if constraints are satisfied. The amount of exchanged information should be limited to the minimum necessary to make the decisions. Furthermore, the complexity and the meta-information that needs to be agreed on in advance should clearly define the syntax and semantics of expressions in order to support evaluation. At the same time, the meta-information should be simple and generic enough to be extensible when applications evolve.

---

[1]This principle has been used for persistence [Atkinson87]; i.e. objects should be allowed to be made persistent, independently of their type.

Composition

In open systems, services can be realised as compositions of other services or components. A contract may include assumptions about components of the environment the binding is running in. Negotiation and QoS statements should support *composition*, since descriptions from participants which do not necessarily know each other, may need to be combined into one describing the composed system. Given a set of statements about the behaviour of individual components of a system, it is not obvious how to deduce the behaviour of a composed system in terms of its components. We may need to be able to capture a holistic view on the system[2].

Generality

An important issue is to what extent we can produce *general* solutions to the problems. A QoS aware binding facility and associated languages should be useful for a range of applications. It should not be restricted to particular QoS categories or application domains, and it should not be restricted to particular technologies or platforms. In particular, we are interested in supporting binding types beyond the simple client/server model; i.e. bindings may have multiple participants playing different roles.

1.3. Main results

The main ideas of our approach to the problems are as follows:

1. Policies as architectural entities which represent QoS contract templates and encapsulation of enforcement policy.
2. Trading of policy as a principle of negotiation.
3. Declared conformance for matching dynamic QoS statements.

The research documented by this thesis has been done over a long time and contributions have been published in a series of papers [Hanssen98, Hanssen99, Hanssen00, Hanssen05a] and a technical report [Hanssen05b].

1.3.1. Overall binding architecture

Our approach to the complexity of finding contracts is that the set of possible end-to-end decisions are specified in advance and that negotiation is to select from this set. In our architecture, a policy is an entity which encapsulates both a *contract template* (a QoS specification) and a software component which represents a particular way to configure the binding. A policy would represent a *mapping* between environmental properties and the resulting QoS. The policy concept can encapsulate non-trivial relationships like how various components of the implementation and the environment can interact to achieve a given result. It can encapsulate and abstract over application- or technology specific solutions. Furthermore, our contract and policy concepts support the reasoning about extra-functional behaviour to be orthogonal to functional types, since it provides an extra level of indirection between negotiation and the implementation components.

[2] A system's behaviour may be more than a simple sum of its component's behaviour, since the interaction might itself influence the total behaviour.

We propose a distinction between bindings and their *activations*. This may simplify the understanding of QoS contracts and adaptation (replacement of activation) as well as late binding, since contract negotiation is decoupled from binding management. We also introduce the *metapolicy* concept. A metapolicy is a specification of how a binding activates, de-activates, negotiates contracts or adapts to changing environments or requirements.

Policy trading

We propose to use *trading* as a principle of negotiation. This can be seen as a generalisation of ODP trading [Bearman93, ISO97], which is to select a reference to a running service implementation from a (functional) type name and (possibly) an expression over a set of properties which may describe extra-functional behaviour. Service implementers register the available service references in a trader's database and clients search this database for matching offers. Instead of trading service references, we trade policies, i.e. predefined contracts along with policies for how to configure bindings to a service. Such trading should match both a QoS requirement expressions plus an expression describing the environmental capabilities. This means that contract templates are associated with a *trading service* instead of the components or interfaces to be bound.

1.3.2. Profile model

We develop a language for expressions to be used in defining the contract part of policies (profiles), as well as expressions stating QoS requirements or descriptions of environments used in negotiation at run-time. It supports composition and run-time evaluation to match expressions against each other for conformance.

The profile model is founded on *declared conformance* which was initially inspired by declared conformance graphs used for type matching in ODP trading. In its simplest form, a statement is simply a reference to a node in a predefined type graph (a name) and a conformance test is simply to check if two nodes are connected. The simple declared conformance is extended and generalised to allow simple numeric parameters to be associated with node names. This allows more generic expressions. Conformance is declared as *rule-bases*. Essentially, a set of axioms would be defined for an application domain, from which conformance between any pairs of simple expressions can be inferred. We find that it is convenient to *derive* a full set of conformance rules at the time of compiling a rule-set (which corresponds to computing a transitive closure) and generate efficient code for run-time evaluation.

The model is also founded on two *composition operators*: (1) simple addition which essentially corresponds to logical conjunction and (2) a special kind of addition which is a combination of statements in separate contexts. We develop conformance rules for composed expressions, and we show that a conformance checking algorithm for any pair of expressions can be developed.

The profile model has been shown to be useful in our experiments. It has a simple but effective support for composition through offering generic operators. The effect of non-trivial interaction is typically encapsulated in policies though, meaning that profile models should capture constraints on composition rather than what compositions result in. Furthermore, the declared conformance approach could allow QoS statement models which are less complex to agree about and evaluate expressions in than more traditional approaches comparing various parameter values, since profile models allow a high level of abstraction. However, this also means that we may need to specify meaning in terms of measurable characteristics elsewhere to aid component implementers and to avoid interoperability problems.

1.3.3. Infrastructure support

The FlexiBind framework was designed in an early phase of our work to investigate policy binding in the context of reflective middleware. We identify two types of pluggable policy components: (1) *Binders* (generators and resolvers) which represent the binding protocol and aspects of metapolicy and (2) *activators* which define how negotiable aspects (protocols, resources etc.) are configured. Negotiable aspects are dynamically selected and applied to target objects. Bindings (also non-active) are represented by *binding proxies* which can resolve activations.

In later work, we explore further how our binding framework can support profile expressions and policy trading. Binder and negotiator components define how environment descriptors and requirements are collected and composed. Dynamic profile expressions can contain placeholders for subexpressions to be determined by querying the platform at negotiation time. Such querying would be supported by pluggable *inspector* components. In the evaluation of this framework we demonstrate how we can interface with (at least some aspects of) platform dependent resource management and how we can design and implement publish/subscribe bindings.

1.4. Research method

According to [Denning89] the discipline of computing is a unique combination of *theory*, *abstraction* and *design,* which are rooted in mathematics, science and engineering respectively. The method used in this thesis is mostly based on the theory and design paradigms.

Based on the more intuitive understanding of profile expression models, we develop a model for profile expressions formally in terms of a set of definitions and theorems. We use predicate logic as the main foundation of the model and for proving the theorems. The approach to defining conformance rules is also axiomatic. We use first order logic and (to some extent) graph theory to define how we specify conformance rules and how conformance in general can be derived from such rules. These foundations are also used to analyse the model with respect to possible consistency and completeness problems.

Prototyping is important in different stages of the research to discover and explore issues and solutions and to provide *proofs of concept*. This means that we design and implement important parts of our ideas to validate that they are feasible. Based on the theoretical foundations of the profile expression model we implemented a prototype *profile model compiler* in order to demonstrate how rule derivation can be done and that this approach is feasible. We designed an implemented a conformance checker and a policy trader demonstrator using the output from the compiler. These prototypes have also been used as tools in further exploring and validating our approach.

In an early phase, we were involved in the development of a reflective middleware framework. We designed and implemented our policy framework as an extension to this. It was revised at a later stage and extended with support for profile expressions and negotiation based on policy trading. Having prototyped the infrastructure support, we could evaluate our approach experimentally as well as analytically. We demonstrated how we could incorporate system level resource management and how we could support publish/subscribe binding types. We also performed case studies where we applied aspects of our model and tools to application cases.

1.5. Thesis organisation

The rest of this report is organised as follows:

- In chapter 2 we give a overview of the main concepts of QoS aware open systems and the main problems of interest, and we survey some of the related work in these areas.
- Chapter 3 defines the overall architectural concepts, including policy bindings, policies as contracts and basic principles of policy trading.
- In chapter 4 we define the core model and language for profile expressions. We define the semantics of composition and develop conformance rules and a conformance checking algorithm for expressions. We define how rule-bases (concrete models) can be defined as axiom sets and how full sets of rules can be derived from these. We demonstrate this by implementing a profile model compiler. The model is also evaluated with respect to performance, completeness and consistency, interoperability and composition.
- In chapter 5 we study infrastructure support for policy bindings. It describes the FlexiBind prototype binding framework which is based on the ANSA FlexiNet framework. This is later extended to support profile expressions and policy trading. We also evaluate the framework by demonstrating how aspects of resource reservation (network bandwidth) can be incorporated and how to support binding types beyond simple RMI.
- In chapter 6 we further validate our approach by applying it to two different application cases. First, we test how we can produce a profile model and policies for web servers. Second, we investigate how we can support multi-subscriber video-streaming where different quality aspects are split into different event-channels and where negotiation is mainly about how to manage and subscribe to such channels.
- In chapter 7 we summarise and discuss our results, and in chapter 8 we conclude.

Chapter 2.
Context and State-of-the-art

This chapter defines the context of this thesis. It gives an overview over relevant concepts, problems and related research in the area of quality of service (QoS) aware open systems. This includes QoS specification, QoS negotiation and QoS management.

We are interested in how to support QoS aware *binding* to some abstract service at run-time. This involves negotiation of QoS contracts and configuration of the implementation of bindings (possibly including parts of service implementations). We focus on how to support QoS aware binding in the context of flexible middleware, and where the goal is some level of application- and platform independence. We therefore study how QoS management can be supported at the middleware level and in particular, how QoS can be defined and composed for the purpose of negotiating contracts. System level contracts or resource reservations can be important parts of QoS management and would be just consequences of middleware level negotiation. At that level we are more interested in abstractions than the details of QoS mechanisms.

Quality of Service management is a rather large research area. We have tried to limit our analysis to what is the most relevant. We start by giving an overview of important concepts and requirements of QoS aware open systems and in particular, the role of bindings and contracts. Section 2.2 discusses QoS modelling issues and issues related to dynamic QoS statement like conformance and run-time representation. Section 2.3 discusses some aspects of the infrastructure support, including system level and middleware level QoS management issues as well as configurability of middleware. In section 2.4 we summarise some of the most relevant related work in the area, in section 2.5 we do a comparative analysis of some selected relevant research, and in section 2.6 we conclude.

2.1. QoS aware open systems - concepts and requirements

In this section we discuss the most important concepts and requirements related to QoS aware binding in open systems, in particular the RM-ODP, the *contract* concept and how contracts are related to components and bindings. In this context, the relevant problem areas of QoS management are how to *establish* contracts and how to *maintain* contracts.

By *Quality of Service* we mean extra-functional properties such as performance, reliability, availability or security. Such properties are related to implementations or implementation environments rather than the abstract service. Different implementations of the same abstract service, different properties of the environment the service implementation is deployed in, or the use of different protocols for interacting with the service may lead to different QoS as observed by the user of the service. *Open systems* are typically specified in terms of components which use each other's services. QoS may be taken explicitly into consideration in the specification and design of systems in order to extend reusability and flexibility. As well as describing functional interfaces, we may specify QoS requirements and expectations of components as *QoS contracts* between components.

2.1.1. Reference model for open distributed processing

The ISO reference model of open distributed processing, RM-ODP [ISO95b] is a meta standard which describes an architectural framework for open and distributed systems. Many of the RM-ODP concepts are used in component middleware research.

Viewpoints

The RM-ODP introduces 5 viewpoints in which architectures of open distributed systems can be described. A viewpoint defines a model, i.e. a set of concepts, structures and rules. Viewpoints may be viewed as *projections* of a complex architecture to more comprehensible models, each which could be specified in some suitable language (may be specific for the viewpoint). The viewpoints are:

- The *enterprise viewpoint* defines purpose, scope and overall policies of a system.
- The *information viewpoint* defines the information which is processed by the system. Models include information entities, relationships and data flows.
- The *computational viewpoint* defines functional decomposition. A system is described in terms of objects and interactions between them. Object behaviour is typically invoked through abstract operations in abstract interfaces (one object may have one or more interfaces).
- The *engineering viewpoint* defines the infrastructure, i.e. the mechanisms and resources used to realise interaction between objects, possibly in a distributed configuration.
- The *technology viewpoint* defines the choice of platforms and other technologies for implementation.

Viewpoint descriptions should be consistent. In addition to models in each viewpoint one should also define relationships between viewpoints, for instance how a computational viewpoint concept is realised in the engineering viewpoint. There is not necessarily a one-to-one relationship between concepts in different viewpoints.

In our context, it is relevant to look at the computational and engineering viewpoints. The *computational viewpoint* is object based; i.e. objects encapsulate data-representation and behaviour. Each object offers one or more (abstract) *interfaces* for interaction with other objects. Interactions are defined in terms of *signals* (asynchronous notifications) *operations* (corresponds to methods) or *streams*. Computational objects are typically application specific.

The *engineering viewpoint* describes how computational interactions and distribution are realised. Models consist of *basic engineering objects* (corresponds to computational objects) and *infrastructure objects*. Interactions are realised by *channels* which may be composed from *protocol objects* and *transparency objects* (for instance remote stubs etc.). Other concepts related to the structure of a system include: *Nodes* which typically model physical machines or clusters of machines (physical location), *capsules* which encapsulate processing and which typically correspond to address spaces, *threads* and *clusters* (which are persistable memory units containing one or more basic engineering objects). A *nucleus object* controls the resources of a node and would typically correspond to the operating system.

Transparencies

Distribution transparency is an important concept of the RM-ODP. The idea is to hide certain aspects of distribution from end-users or application programmers. The complexities of distribution can thus be dealt with separate from application logic. A computational model could be concerned with functional aspects only; the engineering model can deal with distribution issues like remoteness, heterogeneity or partial failure. The RM-ODP describes a set of possible distribution transparencies (the set is not meant to be complete). For instance, *access transparency* means that application programmers do not need to distinguish between local and remote invocations, and *location transparency* means to hide the physical location of objects.

Binding

The concept of *binding* is important in this thesis. We typically view a binding as an association of a name in a client application program with an implementation of the service, such that the application is able to invoke it. In the RM-ODP, a binding is a coupling between two or more object interfaces which enables interaction. This involves configuration of protocols and associated resources. Bindings can be *implicit;* i.e. binding operations occur transparently the first time a client attempts to invoke an operation using a remote interface reference. Binding can also be *explicit*; it can be established through explicit operations and the resulting binding is visible in the computational viewpoint as an object, possibly with its own interfaces. Explicit bindings may be *primitive* (two interfaces of compatible types) or *compound,* meaning that a *set* of interfaces may be bound and the binding is realised by a *series of* primitive binding actions.

2.1.2. Component systems and contracts

Open systems are typically specified in terms of components and relationships between components; i.e. components use each other's services. According to [Abadi94], open systems are systems interacting with environments neither they or their implementers control. In [Szyperski98] a *software component* is defined as:

> *"a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

An open system is composed from components which may be independently developed and deployed in some running environment. It is specified in terms of contracts between components. Contractual design typically consider functional behaviour only. If extra-functional properties are not explicitly taken into consideration, this could lead to component implementations which are tied to specific environments, since the design includes implicit assumptions on the behaviour of the components or environment. QoS may be taken explicitly into consideration in the specification and design of systems. As well as describing functional interfaces, we may specify QoS requirements and expectations of components, i.e. *QoS contracts* between components. The resulting QoS properties of components may depend on the implementation of components, their architecture, interaction protocols, and how the resources used by implementations are managed.

QoS contracts and QoS aware systems

A *QoS contract* is equivalent to a logical implication: *Exp* ⇒ *Obl.* The service promises to meet its obligation (*Obl*) as long as the expectation (*Exp*) is satisfied by its environment (including the client). The contract is enforced by the component implementation, and possibly, also by the deployment process and the binding process by proper configuration of resources and implementation aspects of the supporting infrastructure.

Components may be deployed at run-time. User requirements and properties of environments may change dynamically. Therefore determining QoS properties and contracts at design time is not always sufficient. Components man need to be designed more independently of the environments they are going to interact with, and open systems need to be *QoS aware* to be able to dynamically adapt to changing usage and environmental properties. In QoS aware systems [Frølund98b], components know what QoS they require and provide, and they are able to communicate this knowledge among each other. This may involve explicit statements of how the behaviour of a component may depend on (abstract) properties of environments. In the following discussion, we use the term *negotiation* about the process of finding a contract and a way to configure the system to enforce the contract.

2.1.3. Scope of contracts

The scope of a contract negotiation is typically the *binding* to some service; i.e. one could specify QoS requirements when explicitly establishing a binding. Alternatively, we might specify QoS requirements per invocation or when deploying a component. It can also be convenient if we could associate QoS requirements and contracts with some grouping of related invocations, possibly involving multiple services. The *transaction* model may be suitable for this.

Bindings

In this thesis we focus on the *binding* as the unit of dynamic QoS provision. A binding has an associated *QoS contract*. In that sense, a binding is the association of a component/service with a specific environment as well as an obligation towards the client. According to e.g. [Li94], QoS offers/requirements associated with computational entities like objects or interfaces can be regarded as *QoS templates* which could be activated by establishing bindings at run-time. However, depending on the model of bindings, templates for QoS contracts may also be associated with bindings or binding templates.

The RM-ODP allows explicit as well as implicit bindings. It is useful to analyse what this means for the contract establishment and how agents participate in the contract negotiation. In the case of *implicit binding,* contracts are negotiated directly between a client and a server component. All behaviour observed by the client is in principle realised by the object implementation bound to, hence this model does not capture the handling of distribution issues like failure or communication delays, separate from the implementation of the service[3]. All QoS offers or potential contracts are associated with the service. Other components and resources which may help realising some QoS (including the communication channel) are modelled as part of the environment.

[3]Implementation may however be open and include components for dealing with such issues.

In the case of *explicit binding,* contracts may be negotiated between the components to interoperate and a conceptual *binding service* as illustrated in figure 2.1 below. QoS offers and potential contracts (templates) may be associated with the binding service. This means that contracts can be specified more independently of the server object. Other components and resources which may help realising a QoS may be modelled as part of the binding object, and components that implement the service may be modelled as part of the environment. For instance, a contract may have an expectation including the existence of a particular service implementation. On the other hand, techniques like caching, compression or encryption, which can be applied to a existing service without changing its base implementation, can be modelled as separate from the service implementation, i.e. as properties of the *binding* object.

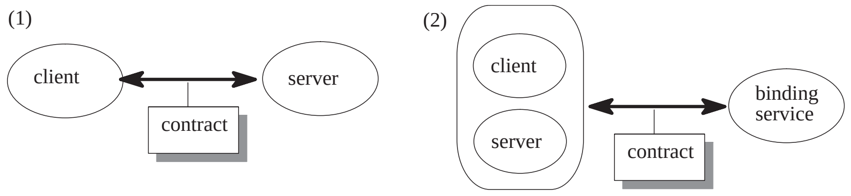

Figure 2.1. Implicit and explicit binding

In the case of non-operational bindings and in particular, multi-party bindings, it may be desirable to let each client negotiate QoS at local interfaces (for instance in a video conferencing scenario, different receivers may require different video streaming quality). Hence, one might argue that the binding is no longer a binding but a separate service available for clients to bind to, or that there are actually two levels of binding and QoS negotiation: One which is shared between all clients and one which is per client (local binding).

Components, deployment and containers

In the discussion above, QoS negotiation is part of the binding process, and a binding is associated with a contract. Alternatively, negotiation could be a part of *component deployment* like in Enterprise Javabeans [J2EE] or Corba Component Model [OMG02]. A component is deployed in an environment represented by a *container*. The extra-functional behaviour of the component's services may be a result of a contract between the component and its container. However, to support QoS awareness, there may be a need for more flexible and configurable containers than provided by typical EJB or CCM implementations as well as support for contract negotiation. Furthermore, there may be a need to address situations where different clients of a given component may have different QoS requirements.

[Beugnard99] argues that it is necessary to make components contract aware to be able to trust components deployed in mission critical systems and outlines how traditional components could be made contract aware. The contract concept has been evolved to involve dynamic QoS. Jamus [Sommer02, Sommer04] proposes a framework for dynamic negotiation of resource contracts between deployed components and their environments. The Arctic Beans project [Andersen01] investigates flexible container architectures, and the QuA project [Staehli04a], investigates dynamic service composition based on QoS negotiation. Other work on QoS aware component architectures includes [Wang00] and [Miguel02].

Transactions

In some cases it may be desirable to specify extra-functional requirements for a sequence of related operation invocations, possibly on different targets. Hence, an alternative model could be *transactions*. Traditionally, a transaction is associated with certain requirements (atomicity, consistency, isolation and durability). The traditional transaction model is questioned by several researchers, and some work suggests that transactional properties should be flexible and negotiable (e.g. [Karlsen03, Arntsen05]). We may take this idea further and view transactions as a unit of computation for which several QoS dimensions could be negotiable. It may be useful to associate transactions with e.g. security or timing constraints in addition to the usual atomicity and isolation constraints. These properties could be negotiated either by binding to a transaction service or to a certain transaction instance. A transaction service would need to ensure that each individual operation is performed such that the transaction as a whole conforms to a negotiated contract.

2.1.4. Contract establishment (negotiation)

The establishment of a *binding* involves agreeing on a QoS contract, establishing an enforcement policy (configuration of implementation and infrastructure) and making sure that sufficient resources are available for the enforcement to succeed.

Negotiation and resource orchestration

The negotiation of a contract involves the search for a combination of choices over many application and system level dimensions. It involves the choice of implementation components, protocols, or allocation of hardware resources like CPU time or network bandwidth. In other words, negotiation is also about *orchestrating* a set of resources, such that they together contributes to reaching end-to-end goals. The search for solutions would involve making trade-offs between possibly conflicting goals; i.e. a choice leading to the satisfaction of required QoS in one dimension may influence the QoS in another as well. For instance, if low jitter end-to-end is a goal, higher jitter in the network component may be compensated by buffering at the receiving side, having the cost of a longer end-to-end latency.

In addition, there may be *global goals*; i.e. the collective behaviour of *all* bindings should satisfy certain constraints. These may conflict with local goals, for instance providing the best possible end-to-end QoS for single bindings, while at the same time trying to achieve good resource economy or a highest possible success rate.

In principle, resource orchestration corresponds to a constraint satisfaction problem [Kumar92, Ruttkay88] or a constraint optimisation problem (if goals include optimisation). Finding optimal solutions is known to be NP-complete. Therefore, the computational cost may be high and the scalability may be limited. Some research has been done in algorithms and protocols for finding solutions. For instance, [Lee99] shows that approximation algorithms giving close to optimal solutions are much faster than algorithms giving fully optimal solutions. However this work is mostly theoretical and assumes a single service component and a static set of applications to be executed on the same time. [Xu01] proposes an algorithm based on graphs representing end-to-end resource requirements and availability and a heuristics based on identifying and focusing at bottleneck resources in reservation plans. Also, the priority is on resolving contention rather than maximising overall utilisation.

Trading

The concept of trading has emerged in the context of the RM-ODP [Bearman93, ISO97, Vasud98a]. Trading is used for servers to advertise (export) their services and for clients to find (import) instances of some service type at binding-time. Thus, a trading service can be said to act as a matchmaker between service offers and requests. Trading was prototyped by the ANSA Phase III project [Deschrevel93] and later adopted by OMG for the CORBA architecture [OMG00].

A trading service stores *service references*, where each of these is associated with a *service type* (functional type) and a set of *properties* (name-value pairs), which may describe extra-functional aspects. An import request names the type and provides a boolean expression over the properties. Matching is done by evaluating the expression with the property values of exported offers and by testing if the requested type is satisfied by the type of the exported service offer. This would typically be done using a type-conformance graph stored in the trader. The trading architecture also includes naming contexts and the possibility of federating trader instances.

Traditional ODP trading can be seen as simply selecting a suitable implementation of a given service type, but one may also involve parameters representing extra-functional properties, and the simple model of trading service references may be extended. For example, [Vasud98b] discusses the possibility of trading dynamic compositions of objects (acting as adapters and adapter chains), to produce bindings such that property constraints given by the clients are satisfied. This means that each adapter component needs to be traded and that the property model must be augmented with constraints on how adapters can be placed in relation to each other. Also, [Rafaelsen00] considers using trading to select templates for multi-party stream bindings. In [Rafaelsen02] a trading service for *media gateways* (which may be part streaming bindings) is designed and implemented. It has many similarities with CORBA trading, but needs a more expressive language for specifying offers and constraints.

Admission control and resource reservation

Admission control is the process of determining if the system is able to deliver a particular QoS at a particular point in time. This includes checking if there are sufficient resources available to support a contract, possibly by attempting to *reserve* the needed resources. Admission control and reservation is necessary if the QoS resulting from a negotiation includes timing constraints which are to be guaranteed, since shared use of resources combined with failures may lead to unpredictable fluctuations of QoS from the underlying system.

Reservations must be enforced by control techniques like scheduling of shared resources, shaping or policing of traffic. It is challenging to provide guarantees in a system with shared resources while preserving a high utilisation of the resources, and while allowing bursty traffic. Most transport (networking) related research and development on admission control and reservation has been done in the context of ATM networks and the internet, and this is typically motivated by a demand for distributed multimedia support. This includes the *XRM/XBind* architecture [Lazar94], ST-II [Topolcic90], SRP [Anderson91], RSVP [Zhang93] and more general measurement and probe based admission control [Kelly00]. Furthermore, a fair amount of research is being done in the context of operating systems, in particular in the area of CPU scheduling. System level QoS is further discussed in section 2.3.1.

2.1.5. Contract maintenance (adaptation and renegotiation)

A contract should be *maintained* during its lifetime. This could involve adjustments (adaptation) to changes in environmental properties. In principle, a system could detect when contracts are violated and initiate *renegotiation,* leading to a contract change. There are two types of contract change: (1) Transition to a contract which has a compatible obligation to the client and (2) transition to a contract which has not (the client requirement can no longer be satisfied). Typically, only the second type is referred to as renegotiation from the application point of view. Alternatively, one could adapt just to find a more efficient way of using the available resources. The first type is something close to the concept of *application transparent* adaptation [Jing99]. Environmental changes are masked by the contract and its enforcement policy. Alternatively, the application must adapt as well. Figure 2.2 below illustrates this distinction. If the change of contract leads to QoS inside the agreed region, adaptation is transparent, otherwise, a new agreement and a new region of acceptable behaviour must be established. Thus, adaptation can be application transparent within certain limits.

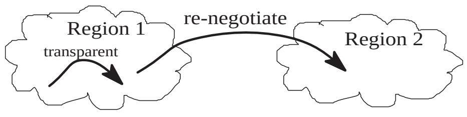

Figure 2.2. Application transparent and non-transparent adaptation

Contracts may be viewed as mappings from a *resource space* (environmental properties) to a *requirement space* (delivered QoS). A contract promises that the QoS of a binding will be inside a certain region of the requirement space as long as the environmental properties stay inside a certain region in the resource space. When adapting, a system should if possible select mappings that map to a point inside of the required region. If not, some mapping may still exist, but which operates outside the current user requirement. Then the application itself needs to adapt. Figure 2.3 illustrates how movement in the resource space lead to replacement of mappings and thus movement in the requirement space. This may be transparent (when mapping *m2* replaces *m1*) or it may result in application adaptation (when mapping *m3* replaces *m2*).

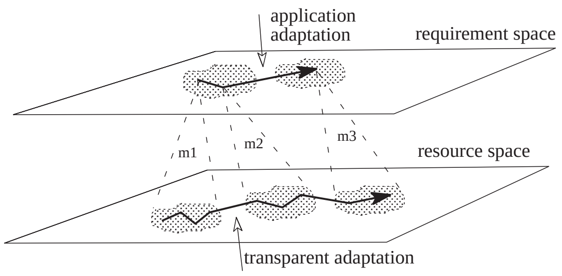

Figure 2.3. Contracts as mappings

Figure 2.4 shows an example of how contracts could be replaced for a video streaming binding caused by a reduction of available network bandwidth (degradation path). At a certain point, this path leads to a change in the requirement space from the agreed region called "*high quality video*" to "*video*". The "*high quality video*" region is compatible with "*video*" but implies stronger requirements. When the video is not "*high quality*", colours and resolution may be reduced and frames may be dropped. Also, adaptation and renegotiation can be done according to a *policy*. Such policies may specify transitions between contracts, and how to select alternative contracts in the case of a transition.

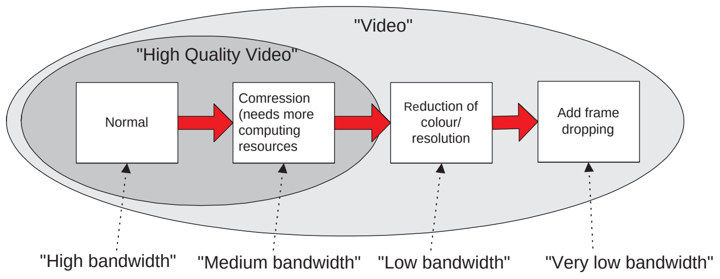

Figure 2.4. Adaptation scenario

2.2. QoS statement

If we look at QoS management from the information viewpoint (RM-ODP) we are interested in how QoS is described and what information is flowing between the parties which are negotiating contracts. *QoS statements* would be used for specification of QoS contracts and are eventually used in negotiation and configuration of the infrastructure (e.g. for requests to operating system or middleware services). In this section we give an overview of concepts for QoS statements, and we discuss dynamic (run-time) QoS statements.

2.2.1. Basic concepts

QoS characteristics and statements

According to [ISO95]¸ a *QoS characteristic* represents some identifiable and quantifiable aspect of the QoS of a system, service or resource, and it is a basic building block of QoS specification. Examples of characteristics include latency time, throughput or availability. A QoS characteristic defines a value domain and a constraint over that domain. One could also specify some interpretation[4] of the values, for instance that a positive integer represents latency time in milliseconds. Ordering may be defined in the sense that smaller values represent stronger QoS than higher values etc. Numeric value domains are not the only possible form of quantification. For instance, set or enumeration domains could be used.

QoS characteristics may be grouped into *QoS categories* where a QoS category represents a type of user or application requirement. *QoS statements* are predicates representing constraints on QoS characteristics. Complex QoS statements may be built from simple predicates using the operators of predicate logic, typically, conjunctions or disjunction. An example of a statement is: *'Latency < 10 and Throughput > 100'*.

QoS relation

The *QoS relation* is a fundamental concept for understanding QoS statements. Formally, QoS relations are assumption/guarantee formulas [Jones83]. A QoS relation is related to a given object o and it is expressed

Exp(o) $\rightarrow$ *Obl(o)*

This is a stronger version of logical implication: If the behaviour of the *environment* of *o* (other objects *o* is interacting with) satisfies the constraint *Exp(o),* the behaviour of *o* will satisfy the constraint *Obl(o)* (the obligation of *o*). [Abadi93] has shown that to simplify reasoning about

[4]We are not talking about a full semantic specification though.

composition, the implication could be restricted with respect to the time ordering of the satisfaction or dissatisfaction of *Exp(o)* and *Obl(o)*. In essence, violation of *Obl(o),* must occur *after* the violation of *Exp(o)*, not before or at the same time instant. *Exp(o)* and *Obl(o)* are safety properties; i.e. properties that can only be violated at a particular time instant. With this version of implication, QoS relations can be *composed* to be applicable to a composition of objects or components; i.e. with a proper formalism and the *composition theorem* it is straightforward to reason about open systems using well-established methods for reasoning about complete systems. The composition theorem gives rules for proving properties of the composed system *S* from a conjunction of component statements like e.g. the following:

$$Exp(o_1) \rightarrow Obl(o_1) \wedge Exp(o_2) \rightarrow Obl(o_2) \Rightarrow S$$

Types of QoS statement

In the ISO QoS framework [ISO95, ISO98] 5 categories of QoS statements are identified: (1) *QoS requirements* which are constraints users of the system (or its components) have on the system (or its components), (2) *QoS capabilities* which are the actual abilities of the system (or its components) with regard to QoS, (3) *QoS offers* which are advertised QoS, which are not necessarily identical to QoS capabilities, because of uncertainty, (4) *QoS contracts* which represent agreements between components of what QoS to provide to each other and (5) *QoS observations* which are values resulting from measurements.

Contracts are QoS relations with an obligation and an expectation. Requirements corresponds to expectations (could be viewed as QoS relations where obligation is set to '*true*'), Capabilities or offers corresponds to obligations (could be viewed as QoS relations where expectation is set to '*true*'). Observations are values which could be used to derive QoS capability statements.

2.2.2. QoS statement abstraction

A QoS relation for an application service involves an expectation towards the underlying infrastructure, while offering QoS to users (clients). The infrastructure can be a set of systems in their own rights, with their own QoS specifications containing obligations and expectations. QoS specifications may be nested recursively. Figure 2.6 illustrates how a QoS specification may depend on QoS specifications of other components, possibly at a lower abstraction level. For instance, the system may satisfy certain frame-rate and jitter constraints, and this will depend on both properties of scheduling of resources in the operating system and the properties of the transport. A transport QoS again depends on properties of the network links and routers.

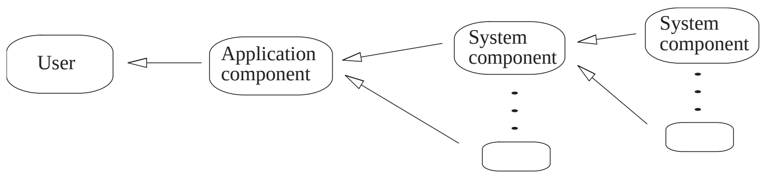

Figure 2.6. Nested QoS relations

Figure 2.7 (derived from [Aagedal01], figure 4) illustrates how system components and nesting levels could be classified. The operating system abstracts over hardware, and all software components may depend on properties of the operating system. Also, the user interface and business logic may be regarded as separate components as well as the transport and networking subsystems.

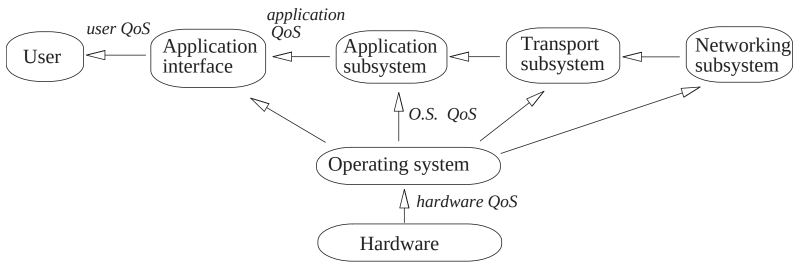

Figure 2.7. Typical nesting level classification

From this figure we also see that different QoS characteristics are used to describe the different components and abstraction levels and that characteristics could be classified according to the different nesting levels.

In the context of *middleware*, it makes sense to focus on, and distinguish between, application oriented and system oriented QoS. Application QoS metrics should be related to user perception or the abstractions application programmers deal with, like for instance invocation- or frame-rate, or CD-Quality versus Phone-Quality. System oriented metrics are related to properties of system resources. This can for instance be bandwidth in terms of bit per second rate, or a choice between the internet versus a dedicated ATM connection, or between service classes of a given transport service. In the context of middleware, system QoS will typically mean operating system QoS or transport QoS.

There are various ways to define the nesting levels. For instance, the TINA QoS negotiation architecture [Rajahalme97] defines a *mapping* between various QoS contexts in a 5-layer hierarchy, where the layers are: (1) The user-service level, (2) the media level, (3) the encoding level, (4) the middleware level and (5) the connectivity level (transport QoS).

2.2.3. Dynamic QoS statement

QoS should first be considered at *design time,* since QoS properties of components may influence architecture and how components are implemented. This is much like extending the traditional notion of interface with extra-functional properties. We refer to this as QoS modelling or *static QoS statement*. This is however not always sufficient for open systems where components are to be deployed or replaced at run-time, or where the system are to dynamically adapt to changing conditions or user requirements. As discussed in section 2.1, QoS-aware open systems should be able to dynamically *negotiate* QoS contracts between services and their clients. Hence, there is a need for a *run-time* representation of certain aspects of QoS specifications. We refer to this as *dynamic QoS statement*.

Dynamic and static QoS statements will serve different purposes. Static statements should provide complete and precise information of the meaning of QoS contracts to support design and implementation. This is the main purpose of QoS modelling languages (see section 2.4.1). Static statements could be used to specify the semantics of QoS values to be referenced at a later time. Dynamic statements should mainly support automatic *conformance checking*, *composition* and (possibly) *prioritisation* of alternative conformant candidate contracts during negotiation.

Conformance

A negotiation process needs information about the expectations and obligations of possible contracts. This information is needed for checking if the expectation is satisfied by the environment and if the obligation (offer) satisfies the requirement. If QoS statements are formulated as predicates, a statement p_1 *satisfies* a statement p_2 if $p_1 \Rightarrow p_2$ is true. Conformance is a partial ordering on QoS statements and is typically denoted by the operator '≤'. I.e. if $a \leq b$, a is equivalent to or stronger than b.

There are two approaches to dynamic conformance checking: The first approach involves run-time *evaluation* of the constraints against other constraints or parameter values. In this case, some representation of the constraints (or parameters) should be available at run-time. Alternatively, conformance relationships could be defined a priori (manually or by static analysis of specifications) between predefined qualities, resulting in a graph of such relationships. Declared conformance has been limited to operational interface types in ODP trading (section 2.1.4). Rules for determining conformance have been proposed for signature based specifications [Black87] and to some extent, for specifications where semantics are to be taken into consideration. Conformance rules for stream interface specifications have been proposed in [Eliassen98], and [Rafaelsen00] discuss conformance checking of binding types for trading. An advantage of declared conformance may be lower computational complexity in evaluating statements. However, a disadvantage is that it will only work with pre-registered conformance relationships and that conformance checking must be done by consulting a service like e.g. a trader where the graph is stored.

Ordering

The goal of negotiation is to reach an agreement on a contract. An additional goal might be to find the best possible contract, according to some ordering policy. For instance, one could try to maximise the QoS in particular dimensions or minimise the cost with respect to resource consumption. Such goals could be conflicting. Furthermore, a given goal is not necessarily equally important everywhere. Therefore, some trade-off policy may be needed. This cannot always be determined statically, ordering criteria may be different for different components and may change over time, due to varying resource availability.

Conformance defines a partial ordering on QoS statements. Negotiation eventually needs a total order to do the actual selection. Each component, *c,* which takes part in negotiation may have a different ordering function which could be stated as $W_c(Q)$, where Q is the domain of QoS statements. W_c defines a total order. It is however impractical to require participants to specify a total order, so in practise they specify a partial order $w_c(Q)$. $W_c(Q)$ could then be said to do random ordering where $w_c(Q)$ does not define an order.

[Koistinen98] proposes to apply worth calculation to negotiation; i.e. one may specify the relative importance of values of QoS dimensions, different contract types and different operations of an interface. The main idea is to have a *worth function* which computes the worth of an offer, based on the current preferences of a client (cf. $w_c(Q)$ above). A client could specify a *worth profile* which defines weights and functions, in order to compute worth values from offers. [Aagedal01] sketches a scheme where one can specify a worth function for each QoS characteristic; i.e. w_c operates on characteristic-values. The worth of a QoS statement is then computed as a product of the individual worths (based on the strongest possible characteristic value) if combined by conjunction or as the maximum of the individual worths if combined by disjunction. *Utility functions* [Capra03] is a similar concept to be used in QoS statements. Such functions are provided by clients and are used to compute the utility (a value between 0 and 1) of the offered QoS.

Composition

Service implementations or environments may be compositions of components. Hence, their resulting QoS may need to be derived by combining the QoS of each part. In particular, an expectation part of a contract could be satisfied by a combination of offers from different components of the environment. This may include components of the client environment, the server environment and other components in the connection path between them.

In open systems, one cannot always know statically what actual components the environment will consist of. Therefore it is important that QoS statements can be composed, not only at specification time, but also dynamically during negotiation. The semantics of composing QoS statements should be clearly defined, and here, it may matter what components QoS statements are about and *how* they are combined. For instance, an expectation may state that certain resources need to be available on the client side, which means that a component on the server side cannot satisfy it.

Composition seems to be weakly supported in most related work. Currently, there is no general solution to QoS composition. [Aagedal01] allows the specification of composition semantics of individual QoS characteristics. It is possible to define how to combine values of the same characteristic from different components into a single value, for three different composition patterns. To some extent, QoS composition is investigated in the context of composing web-services and workflow management. The AgFlow middleware system [Zeng04] specifies service compositions as task graphs and selects service compositions, or it performs adaptation through re-planning, based on some QoS criteria. The selection is mainly calculation of utility value, based on values on a fixed set of QoS parameters. The focus is on efficient searching for optimal plans, using e.g. integer programming. [Jaeger04] studies QoS parameter aggregation for web service composition and uses known workflow patterns to derive a set of *aggregation patterns* which can be used on QoS statements. Two patterns are identified for *sequential* composition and five patterns are identified for *parallel* composition. An aggregation function must be given per-pattern and per characteristic. This scheme can be seen as a generalisation of the composition construct in [Aagedal01].

Dynamic statement representation

Dynamic QoS statements need some form of representation to allow the exchange of values between components of an open system. For instance, we have a choice of using parameters and APIs to specify them and a more general language approach. For the language approach, expressions can be compiled or interpreted. A related issue is to what extent meta-information should be carried with dynamic statement representation to tell the parties how to interpret the values.

A simple approach is to use tuples of parameters. A tuple may specify specific values, ranges (or sets) for a set of QoS characteristics. A common matching function defines the semantics of the values a priori. One may define different tuples for different application domains or QoS categories. This is the typical approach of early QoS architectures which focus on end-to-end guarantees for multimedia streams, like e.g. QoS-A [Campbell96] where QoS statements are instances of predefined C-language structures.

More generally a QoS statement might be represented as a list of tuples, where each tuple represents a constraint on a QoS characteristic. Dini and Hafid [Dini97] propose a model of service parameters for negotiation between consumer and provider components. Here, parameters are tuples which carry the necessary information and meta-information to carry out a generic matching algorithm. The essential parts of such tuples are the parameter name, the value type, the request value for consumers, which can be a single value, an interval or a set, or alternatively, a value space which is used by providers to define the space in which the value can vary, the measure unit which allows the matching algorithm to convert values when matching values using different units, plus some fields indicating to what extent the requested values are negotiable. In this model, tuples represent QoS offers or requirements, and they contain the definition of the characteristics they constrain.

QoS statement types may be defined statically by the application designer in some *QoS modelling language*, which could be compiled into data-structures or classes which can represent actual values at run-time. The generated code may include methods for evaluation (for instance conformance checking), as well as for marshalling of the data structures since they are meant to be exchanged between different nodes of heterogeneous systems. QoS modelling languages like QML or CQML (section 2.4.1) are mostly focused on specification at design time to aid decisions on architecture and component design. However, it is recognised that to support open systems, run-time representations should be supported as well. Therefore, these languages support compilation into CORBA IDL type definitions to represent QoS statements, code in a programming language to instantiate such structures, as well as code for conformance checking. This means that some of the generated code must be statically linked in negotiating components. To allow full flexibility, QML also offers structures that contain all necessary meta-information to create and manipulate QML definitions at run-time, but at the cost of efficiency.

2.3. Infrastructure support

Application objects (or components) would be designed to run on some *infrastructure*. An infrastructure provides the services necessary to run and to interact. To support the composition of objects in a QoS aware open system, an infrastructure should support the enforcement (or engineering of) QoS contracts. This requires some level of flexibility and configurability of the infrastructure itself.

According to [Blair97] the distinction between application and infrastructure is mainly the same as the distinction between the computational and the engineering viewpoints of the RM-ODP (see section 2.1.1). The engineering viewpoint offers the means to explicitly engineer the infrastructure in order to realise selected distribution transparencies and QoS contracts. In the case of run-time adaptation there should also be some support for infrastructure engineering at run-time, as well as negotiation to find and agree about the contracts.

The enforcement support should also include system level QoS contracts (see section 2.2.2) and some inspection/reporting of system level QoS, since (as observed in section 2.2.2) QoS contracts can be nested in the sense that application level QoS is based on system level QoS etc. In this context we discuss QoS management in terms of three distinct nesting levels, or layers: (1) *The application layer* in which application components and application requirements are defined, (2) *the system layer* in which hardware and operating system resources are defined, and (3) *the middleware layer* in which the distribution infrastructure is defined.

Middleware realises distribution transparencies and abstracts over platform specific services and resources. Ideally, a middleware could define a single platform on which applications are run, and where differences between system level platforms are hidden. However, we need a more flexible notion of a middleware platform. According to [Almeida04], there would be a trade-off between (1) designing with *particular middleware platforms* in mind leading to reusability of middleware but also platform dependency, and (2) designing with *abstract platform requirements* in mind, leading to the opposite. A realistic compromise may be to specify for an abstract platform which could be mapped to specific platform implementations. The term '*infrastructure*' could then mean an abstract platform which is specified in terms of the bindings, transparencies and QoS constraints which are supported. Variations in both system level capabilities and application requirements would lead to concrete platforms which are differently configured at different nodes and which should support some level of dynamic reconfigurability.

In section 2.3.1 below, we first discuss system layer QoS mechanisms since this is to some extent relevant for how we understand the middleware level. In section 2.3.2, we discuss middleware layer QoS management mechanisms. In the last section, we discuss principles of how middleware implementations could be opened up for explicit engineering at run-time.

2.3.1. QoS mechanisms in the system layer

QoS contracts would eventually involve assumptions on how resources at the operating system or transport level, are to be associated with implementations. This is particularly important if contracts include guarantees on real-time behaviour. For example, if service invocations are to be performed within time constraints, the CPU would need to be scheduled to the application within certain deadlines, network bandwidth and delay would need to be within certain bounds etc. In particular, distributed multimedia applications like video conferencing have been shown to be sensitive to the QoS of the networking and how the CPU, memory and I/O devices on each platform are scheduled. In the following we discuss the concept of reservation, network QoS and operating system resource management.

Reservation

To provide *"performance isolation"* between concurrent applications or to provide real-time guarantees beyond statistical estimates, it may be necessary to *reserve* some portion of computing resources at the time of establishing a contract. A reservation can be viewed as a QoS contract; as long as the resource consumption of the application stays within certain limits, some behaviour is guaranteed. Reservation can be specified in various dimensions, for instance that the resource is available within certain time-limits, periods etc. The first phase of a reservation would be to perform an *admission test* to determine if the system is able to enforce the reservation with the current load. If successful, the system should enforce the reservations by proper scheduling, accounting, traffic shaping, policing, etc., until the reservation is released.

Network QoS

In the discussion of transport QoS, it makes sense to focus on internet technologies, since this is the most common transport service used (regardless of what technology is used underneath the IP layer). At this level, we are interested in how end-systems can observe and manipulate internetworking QoS, rather than the network specific protocols for signalling etc.

Two architectures for internet QoS management are dominant: *Integrated Services* (IntServ) and *Differentiated Services* (DiffServ). *IntServ* [RFC1633] can support end-to-end guarantees for individual data flows and is typically used along with a *resource reservation protocol*, RSVP [Zhang93], to negotiate QoS with the other end and to perform admission control and reservation in each router along the data path. IntServ requires per-dataflow state to be maintained in each router, which is not very scalable. We do not expect *IntServ* to be widely deployed all over the internet; it is more likely to be used within organisations.

Reservation requests to RSVP essentially consist of two parts: (1) A characterisation of the traffic to be generated by the application *(TSpec)* and *(2)* what QoS reservation is required *(RSpec). A TSpec* essentially describes the expected load (and is formulated in terms of a token bucket model). A *RSpec* describes constraints on the network delay. These two parts can be regarded as a contract proposal. Two traffic classes are supported: *(1) Controlled Load* which represents a statistical guarantee that the dataflow behaves like a best effort service under light load, regardless of actual load and (2) *guaranteed service* which assures that packets arrive within the specified time.

DiffServ [RFC2475] is designed to be more scalable. It does not support per-flow reservation but rather a fixed set of routing behaviours which may be requested using the DiffServ field in the IP header (former TOS byte). DiffServ implies making service level agreements (SLA), i.e. a kind of contracts between network providers and customers (and between different network providers). A SLA consists of a QoS obligation and traffic rate expectation. It should also be associated with rules for classifying packets belonging to it. As exemplified in [Evans04], one can define a set of service classes, for each of which a SLA can be defined, for instance a VoIP class or a throughput-optimised class. Routers at the edges of networks must shape or police the traffic according to the SLAs and classify packets entering the network. Since DiffServ supports contracts for aggregated flows, it should be possible to provide guarantees per application flow, by keeping track of how individual flows share an aggregated one, by doing admission control and proper queue scheduling in each end-system for packets entering the network.

Operating system resource management

Real-time QoS (guarantees in particular) depends on how the operating system schedules concurrent access to hardware resources like the CPU, memory, disk or network I/O, not only the characteristics of the devices themselves.

Common general purpose operating systems have rather limited support for QoS. They essentially offer timesharing CPU scheduling of processes or threads with priorities. Priority scheduling can be used to implement for instance Rate Monotonic (RM) or Earliest Deadline First (EDF) scheduling, and it is possible to provide soft real-time guarantees and reservation at the middleware level, as demonstrated by the DSRT project [Chu99]. Since Linux or BSD Unix dialects are often used in routers or large servers, they offer advanced network queue management (possibly supporting *DiffServ*), making it possible to reserve a bandwidth for certain classes of packets.

A fair amount of research has been performed in operating system resource management. In particular on CPU scheduling since the CPU is what actually executes any program or device driver code. The *Processor Capacity Reserves* framework [Mercer94] allows processes to make periodic reservations of CPU. It was implemented in Real-time Mach using EDF scheduling. A kernel *reserve* abstraction was introduced to keep track of CPU usage. Other work which further investigates CPU reservation includes Nemesis [Roscoe95], Rialto [Jones96] and Eclipse [Bruno98].

Supporting guarantees in general purpose operating systems can be challenging due to the different behaviours and requirements of applications to be run concurrently. As with QoS in general, reservations for a resource can be characterised along several dimensions. For CPU resources, one may for instance require a *minimum*, *maximum* or *average* percentage of the CPU cycles, but one may also want to be scheduled periodically within certain time frames, possibly with deadlines. In summary there may be different kinds of reservation requests for the same resource, and they are not necessarily trivial to accommodate at the same time. To handle different type of CPU guarantees, multiple service classes are proposed, for instance in DSRT [Chu99] or HLS [Regehr01]. Furthermore, one single scheduler (or scheduling policy) may not cover all behaviours. Therefore, one may allow multiple schedulers which are hierarchically composed. For example, one of the shares managed by a top-level scheduler may be further subdivided by another scheduler using a different policy. In e.g. Vassal [Candea98], schedulers are treated as pluggable components. The HLS project investigates service classes along with hierarchical composition of schedulers and proposes a way to specify schedulers as mappings between guarantees; i.e. a scheduler can provide certain guarantees (to its clients). To be able to do so, it requires a certain guarantee from its parent scheduler. Clearly, the properties of a given scheduler also depend on the properties of the parent scheduler, and not all compositions are meaningful.

Resource contexts

An activity may need a combination of *"guarantees"* from a set of different resources, to function as required. Furthermore, an activity can be composed from different sub-activities, both in user-space and kernel-space. In the general case, it may even span different nodes. For instance, to provide a network throughput or latency, it is (strictly speaking) not sufficient to acquire a network QoS guarantee (e.g. by using RSVP), but one also needs to ensure that kernel threads processing data transfer are scheduled by the CPU sufficiently often. Also, queues of IP packets to/from network interface should be scheduled accordingly for sending or receiving on the wire.

In the design of an operating system there is a need to decide what entity (resource principal) reservations should be associated with; it could be applications, processes, threads, or network connections. If for instance resources are associated with threads, these threads do not necessarily map directly to the actual activities for which we want to guarantee a behaviour. An I/O bound user level thread may be idle most of the time, while waiting for interrupts from the device and while a larger part of the CPU work is performed by kernel threads running I/O driver code.

A fair amount of research on operating systems has therefore proposed a *separation of concerns* between some *resource principal* requesting a set of resources and the set of *entities* consuming them, like processes, threads or applications. This approach supports resource isolation, allows applications unaware of the resource management to take advantage of reservations. Furthermore, it avoids the NP-hard problem [Blazevicz86] of scheduling multiple resources with timing constraints. The *Resource Kernels* approach [Rajkumar98] decouples reservation and resource usage and proposes a *resource context* independent of the process abstraction. Nemesis [Roscoe95] and Eclipse [Bruno98] propose a concept of *reservation domains*. Processes and resource reservations can be put into such domains. A domain is like a virtual machine providing protection from other domains. This idea is also implemented in CKRM [Nagar04], an extension to the Linux kernel. Here, a domain is called a *class*, a class can be further subdivided into a hierarchy, and the properties of classes can be easily accessed through a virtual file system. CKRM can manage CPU, disk, memory and listening sockets, by use of fair share scheduling (can reserve percentages of a resource). New resource managers can be added as extensions. Threads or sockets can be put into classes or moved between them. *Resource containers* [Banga99] function as resource principals like in the other frameworks above. However, this abstraction allows a more flexible notion of what constitutes an independent activity. Essentially each thread can establish and change resource bindings to a container to charge resource consumption to it. However, this approach can lead to extra context switches which have some overhead.

2.3.2. QoS mechanisms in the middleware layer

The middleware provides the infrastructure to the application components. It abstracts over various hardware and operating system platforms and implements distribution transparencies. In this section, we look at how QoS contracts can be supported by middleware mechanisms. QoS enforcement is mainly supported in two ways: (1) By *mapping* application level QoS requirements to system level requirements (which are used to negotiate contracts with the system level), and (2) implementing mechanisms and protocols in the middleware itself.

QoS mapping

Resource reservation protocols operate at low levels of abstraction which are not straightforward for programmers (or users) to understand. Many different reservation models and protocols exist, using different parameters. In addition, different applications or application domains may use different sets of parameters. One role of middleware would therefore be to mask differences in underlying resource management and to support QoS by mapping between application level parameters and system level parameters.

QoS architectures aim to integrate QoS mechanisms in various subsystems, in order to provide end-to-end *"user-level"* QoS. Classical work [Aurre96] typically depends on particular platforms or application domains, or lack of transparency from system level (e.g. like QoS-A [Campbell96] or XRM [Lazar94]). The QoS broker/Omega architecture [Narhstedt95] introduces mapping, but in a centralised manner which (as pointed out by e.g. [Waddington97]) may require a huge amount of mapping information and have limitations with respect to flexibility.

Quartz [Sigueira00] introduces pluggable translator components to/from application- and system-level QoS parameters. Quartz is an open architecture where pluggable translators map from application specific to a generic application level parameters, or from a generic system level to a specific system level. Thus, the mapping from application level to a system level, which is the most complex part, can be handled by a single generic translator component. In the $2K^{Q+}$ architecture [Nahrstedt00] the mapping is performed by a compiler translating a QoS specifications (containing user- and application level QoS) to QoS profiles which also contains system level requirements. This process may use probing (testing the profiles on the running platform) to find proper system level parameters.

Reservation vs. adaptation

It is common to distinguish between static and dynamic QoS management approaches (cf. e.g. [Aurre96]). *Static QoS management* is done at the time of establishing a contract. It typically involves admission tests and reservation of resources to enforce the contract. A reservation based approach has the advantage of providing guarantees, and it is effective in the case of resource contention. However, there are some serious issues:

First, existing system level infrastructures are often best effort. As discussed in section 2.3.1, models and mechanisms are being developed, but are not widely deployed. End-to-end reservation based QoS management requires the deployment and integration of many different system level QoS aware components. This is a rather challenging task. Furthermore, the actual QoS requirements and resource usage of an application will vary over time, and it may be difficult to make precise estimates of the required resources a priori. It may be necessary to base reservations on predictions on worst case needs of a contract, which may lead to a waste of resources.

Alternatively, *dynamic QoS management* involves adapting configurations in response to some feedback. This is proposed in e.g. [Beaton97]. Implementations may contain run-time monitoring of system level QoS parameters or measurements to discover if QoS contracts are violated. Analysis of such measurements may trigger compensating actions or re-negotiation. Requirements can be specified as rules, contracts or policies. According to [Molenkamp02] a *policy* is in this context defined as "*a rule that describes the action(s) to occur when specific conditions occur*". Here, one typically specifies events and associated actions. Adaptation mechanisms may be used instead of (or in addition to) reservation to enforce QoS contracts by adjusting the implementations and resource allocations until the requirements are satisfied.

A fair amount of research investigates QoS provision through adaptation techniques in the middleware layer. Often, adaptation is modelled as feedback control loops like in e.g. SWiFT [Goel99] or AMIDST [Bergmans00a]. In QuO [Zinky97] object implementations are defined through aspect languages and adapted by explicit contract replacement (see also section 2.4.3). Agilos [Li99] takes the approach of adapting the application (as opposed to adapting mechanisms in the middleware). This adaptation is however controlled by the middleware. The reservation approach and the adaptation approach are not mutually exclusive as demonstrated in e.g. [Foster00].

2.3.3. Open engineering and reflection

In this section we discuss principles of how middleware implementations could be opened up for explicit engineering at run-time. The ability to inspect and modify the way programs are interpreted is often referred to as *reflection*. This concept was first introduced in [Smith82], where the reflection hypothesis is stated as follows:

> *"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures"*

This means that a program can inspect and alter its own interpretation. Access to the interpreter (virtual machine) is provided through a *meta object protocol* (MOP) which defines the services available at the *meta level*. A much cited work on applying the idea of reflection to programming language is [Kiczales91]. This is related to object orientation and the approach of Open Implementation [Kiczales96a]. By using meta object protocols one could expose implementation issues in an orderly manner without sacrificing the benefits of abstraction. According to Kiczales, all implementation issues cannot be hidden, because some of them are not just details but decisions which affect the behaviour of the application significantly. The approach to this complexity is to deal with such issues *separately* rather than hiding them (the principle of separation of concerns).

Much research is based on these insights. Reflection is investigated in the context of operating systems like e.g. Apertos [Yokote92], and since about 1997, there has been significant research on reflection in middleware for distributed systems, starting with [McAffer96]. Follow-ups include FlexiNet and OpenORB (see section 2.4.4).

Principles of reflective middleware

According to [Kon02], a reflective middleware is implemented as a collection of (engineering viewpoint) components that can be configured and reconfigured by the application level code. The APIs could be like in traditional middleware, but in addition, system and application code may be given the ability to inspect or reconfigure the internal configuration through *meta interfaces*. In [Blair98, Blair01], some general principles of reflective middleware are identified. This includes the concept of getting access to implementation aspects through per-object or per-interface *meta-spaces*. Access to meta-level objects is provided through *reification*, which makes some aspect of the internal representation visible from the program, where it can be inspected or modified. It is distinguished between structural and behavioural reflection and two meta-models for each type are identified: First, *structural reflection* deals with content and structure of a component. The following two meta-models are identified:

- The *Interface meta-model* which represents the functional aspect, i.e. properties of interfaces like operations, types or inheritance structures. This corresponds to the traditional view on meta-information in programming languages, for instance reflection in Java.
- The *Architecture meta-model* which represents the structural aspect; i.e. it offers access to an objects in terms of its constituent objects. This is essentially a graph of objects interconnected by local bindings.

Second, *behavioural reflection* deals with activity in the underlying system, i.e. the environment the components are running in. Two meta-models are identified:

- The *interception meta-model* which represents the execution environment for interactions between interfaces, as traditionally provided by the middleware platform. This includes transparency mechanisms like stubs, marshalling and message passing, but also QoS monitors can be addressed here. The focus is the dynamic insertion of interceptors.
- The *resources meta-model* which addresses management and scheduling of shared resources like memory, network bandwidth or CPU time.

The interface and interception meta-models are related to interfaces (since each object may have more than one interface). Architecture meta-spaces are related to objects, and resource management is relevant for both, since both object and interaction implementation may consume resources. Furthermore, resources are shared between entities. Therefore, it is proposed to have one resource meta-space per address space. Meta-spaces are (at least conceptually) reified through four corresponding operations available at each interface. Since meta-spaces and their constituent objects are first class objects, they may as well have meta-spaces; i.e. reification may be done recursively, revealing meta-meta-spaces etc.

Open bindings

Reflective middleware research has mostly focused on bindings as a means to connect multiple interface in a distributed environment. An *open binding* [Fitzpatrick98] provides access to its architectural meta-model. This composite distributed object is typically *nested* in multiple levels; i.e. a binding can be composed from other lower level bindings. Figure 2.8 illustrates how a binding can be composed from an inner binding plus other objects, interconnected by local bindings. The inner binding may be opened up similarly, recursively, allowing us to view bindings at different abstraction levels.

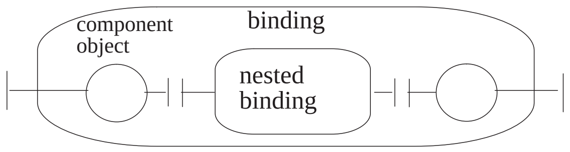

Figure 2.8. Open binding

The nesting stops at primitive and closed inner bindings, which could for instance be basic transport services that interconnect end-systems. An open binding corresponds to a *channel* in the RM-ODP Engineering Viewpoint terminology. Nesting levels may correspond to layers of a protocol stack, in the sense that objects of a binding use the services of a nested binding to interact.

To allow automatic configuration and reconfiguration of bindings, one would define meta-models of bindings representing important parts of bindings as well as rules for how bindings can be configured. In addition, bindings may for instance be created by a corresponding set of nested *binding factories*, or changed by *binding-mutators*. Since binding-factories need to coordinate actions across inner bindings there may also be a need for *nested binding-protocols* [Eliassen99] which might subsume QoS negotiation and nested QoS statements (section 2.2.2).

2.4. Related work

The above sections discuss requirements and to some extent the different approaches to QoS aware binding, QoS statement and infrastructure support. In this section we survey some of the most significant projects in the area of QoS middleware. First we look at QoS modelling languages, since it is important to know how QoS is specified and expressed in negotiations. Then we look at some middleware approaches: (1) QoS management research in middleware architectures, (2) approaches which apply the idea of aspect oriented programming to QoS and middleware, and (3) approaches which primarily study flexibility, componentisation and reflection as means to make middleware flexible and to support QoS management.

2.4.1. QoS modelling languages

Researchers have considered languages for QoS specification or modelling, but usually such approaches are based on a specific QoS category. Tina ODL [Tina94, Tina96] and MAQS [Becker97] propose IDL extensions to annotate interface definitions, thus tying QoS properties to interface types. The QuO (see section 2.4.3) contract definition language specifies contracts and adaptations in terms of QoS regions but does not support refinement, conformance or fine grained characterisations. QDL [Daniel99], allows the specification of QoS objects, each consisting of a QoS expectation and an obligation specified in OCL (UML Object Constraint Language). The constraints are expressed in terms of simple properties or other QoS objects.

QoS specification languages are surveyed e.g. in [Jin04]. In this section we look at attempts to define more generic *QoS modelling languages* focusing on the middleware and application layer. Such languages are meant to complement modelling languages like UML. Hence, precise QoS specifications would aid the design of architectures and component implementations. QoS modelling languages allow definition types or characteristics for any domain. Furthermore, they aim to support abstraction and refinement in an object-oriented style. QoS modelling languages can be useful in defining (by compiling specifications) data-structures, constructors and conformance checkers to be used at run-time in QoS negotiation.

Quality Modelling Language (QML)

QML [Frølund98a, Frølund98b] is a QoS modelling language for distributed object systems and offers the following main abstractions: (1) Contract type, (2) contract and (3) profile.

A *contract type* consists of a set of dimensions that can be used to characterise a particular QoS aspect. A dimension has a domain of values which can either be numeric, enumerated or set. For numeric and enumerated domains, one can define ordering in the sense that either higher or lower numbers mean stronger QoS. A contract type corresponds to a characteristic or a set of characteristics, grouped as a QoS-category. A *contract* is an instance of a contract type in the sense that it constrains the values of its dimensions. Hence, QML contracts correspond to QoS statements (offers or requirements). A constraint can be a <*name, operator, value*> triple where operators are either '==', '<', '>', '<=' or '>=' Alternatively, a constraint can be a set of statistical characterisations. A *profile* associates contracts with an interface type and its operations or attributes. There is a one to many relationship between interfaces and profiles. A profile may apply contracts to the elements of its interface. Profiles may be bound to actual entities. If bound to a client side reference, it represents a requirement and if bound to a service implementation it represents an offer. The binding could be done either statically or dynamically. The focus is on static binding, and an extension to UML to specify such bindings is proposed.

QML supports *refinement* of contracts in the sense that a new contract may be defined in terms of another plus a delta. The deltas are restricted to constraining unconstrained dimensions or making stronger constraints than existing ones. Hence refinement also implies a conformance relationship.

The QML team recognizes the need for dynamic QoS statement to support open systems and negotiation. QRR, a *run-time representation* is proposed, which can express all parts of QML. There are two alternatives with respect to contracts or contract types: (1) All expressions are formulated in terms of generic structure definitions, which carry all necessary meta-information. A QML compiler may generate constructors for such structures or they may be built manually by using library functions. (2) A QML compiler generates IDL types for each contract type plus C++ code for instantiating such structures. It may also generate conformance checking code statically. The second approach is more efficient but requires the meta-information to be linked statically in each process participating.

Component Quality Modelling Language (CQML)

[Aagedal01] proposes a modelling language for QoS, CQML, where the focus is mostly QoS specification in modelling and design based on UML. CQML is a more precise and general purpose language than QML (at the cost of being more complicated). Like QML, it is orthogonal to functional interface definition. CQML offers four main abstractions: (1) *QoS characteristics*, (2) *QoS categories*, (3) *QoS statements* and (4) *QoS profiles*.

A *QoS characteristic* definition consists of a name and a value domain which are similar to dimensions of QML contract types (numeric, set, enum). One can also constrain numeric domains, and one can further constrain the semantics of the characteristics using OCL. Characteristics may be *specialised* (subtyping) or *derived* from other characteristics (used for statistical aspects). Furthermore, characteristics may be grouped into *QoS-categories*, representing types of requirements. To allow composition of characteristic values, one can specify the semantics of *composition operators* for the characteristic in question (parallel-and, parallel-or, or sequential) by using OCL operators like e.g. '+'. This makes it possible to combine characteristic values from different components. Composition can however only be specified in terms of the same characteristic, meaning that CQML cannot express feature interaction this way.

A *QoS statement* is a user defined quality region with a name. It defines constraints on a set of characteristics by using the logical and relational operators of OCL. QoS statements may be further qualified using the '*guaranteed*' or '*best-effort*' keywords. Specialisation is supported in the sense that more constraints may be added (implies conformance relationship). A *QoS profile* associates QoS statements with component specifications, typically, CCM IDL definitions. Unlike QML, a profile specifies both an expectation and an offer (QoS relation). Hence, it is a more complete contract template. A *contract* is a set of chosen profiles. For instance, between two cooperating components a contract is the choice of a profile at each side. To establish a contract there must be a match between the expectation of each side with a combination of offers from the environment. The expectation may be satisfied by a composition of offers from different components. The composition theorem of Abadi/Lamport (see section 2.2.1) states that the QoS statements must be *safety properties*, which means a small restriction on how QoS statements are defined.

QoS profiles support adaptation and negotiation in the sense that one can specify alternative profiles plus their internal ordering. If environmental properties change, this may trigger a new matching attempt, where the first profile in the specified order which matches would be selected. Such matching is generally a constraint satisfaction problem [Kumar92]. Adaptation policies to address the case where no satisfactory match is found as well as worth based negotiation, is not within scope of CQML.

2.4.2. QoS (middleware) architectures

In the context of distributed multimedia, several architectures and schemes have been proposed for negotiating and mapping between application QoS, orchestration of system level QoS and reservation, in order to provide end-to-end QoS guarantees. Some historical work on QoS architectures was surveyed by [Aurre96]. For instance, a QoS brokerage model is introduced by [Nahrstedt95]. The QoS broker is essentially an architecture for handling QoS negotiation between logical entities called *"buyers"* and *"sellers"*. A broker also coordinates resource reservation on both end-systems and communication links in between. Other early work includes NRP/XNRP

[Rothermel97] which proposes a three-phase reservation and negotiation protocol which also involves a central resource orchestrator.

In the next sections we present some middleware level architectures for QoS management: Quartz which focuses on reservation and mapping of QoS parameters, Agilos which focuses on adaptation and OMODIS which focuses on hierarchical composition of various QoS management schemes.

Quartz

The Quartz architecture [Sigueira00] aims to offer flexible, extensible, platform independent QoS enforcement through mapping of QoS parameters. Quartz supports different application areas and different reservation protocols through pluggable components.

Quartz focuses on *mapping* between application level QoS parameters and system level parameters. A layered architecture with three tiers *(application*, *middleware* and *system)* is proposed. Application specific parameters are first mapped to *generic application level parameters* by pluggable *application filter* components. Generic application parameters are mapped to *generic system level parameters* which are again translated to *specific system level parameters* by pluggable *system filter* components. Translating between application and system level is a complex process which may involve trade-offs and resource balancing, and it is performed on generic parameters defined by the Quartz architecture. This translation (which is still a rather complex task) is handled in the middleware layer by an *interpreter* component.

Resource agent components encapsulate access to the actual reservation protocols (using system specific APIs). Quartz supports adaptation in the sense that the system level may choose to change contracts after a while. This may trigger either application transparent adaptation through finding new mappings, or non-transparent adaptation which involve notifications to application code. Therefore Quartz supports reverse mapping as well. The figure below illustrates this architecture. Quartz is prototyped on top of the CORBA architecture and validated with RSVP, ATM and Windows NT thread priority and memory management at the system level.

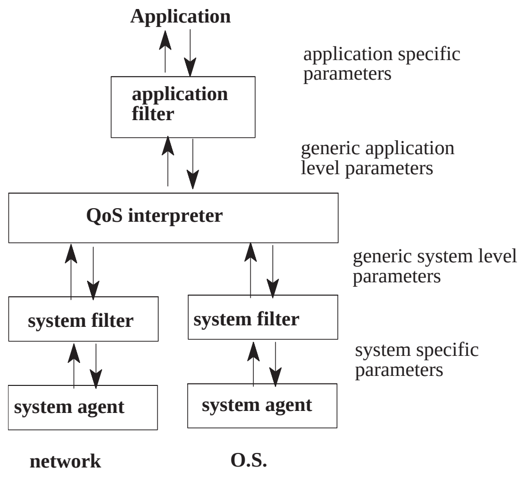
Application
application specific parameters
application filter
generic application level parameters
QoS interpreter
generic system level parameters
system filter
system filter
system specific parameters
system agent
system agent
network
O.S.

Figure 2.9: Quartz architecture

2K^{Q+} and Agilos

[Nahrstedt01] proposes a QoS aware architecture for ubiquitous and heterogeneous environments. This includes experiments on 2K^{Q+} - a unified QoS framework focused on static QoS management [Nahrstedt00, Wichadakul01], and Agilos - a middleware framework focused on dynamic QoS management [Li99]. Agilos also focuses on middleware controlled adaptation of applications rather than of middleware. A architecture is proposed, having three layers: (1) *QoS-aware resource management* layer*,* which is application independent and consists of resource specific *brokers*, *adapters* and *observers* which are responsible for handling admission control, reservation, enforcement, monitoring and adaptation of system level resources. (2) *QoS-aware service management* layer, dealing with configurations of application services and resources and (3) *application* layer running application level components. Figure 2.10 illustrates this.

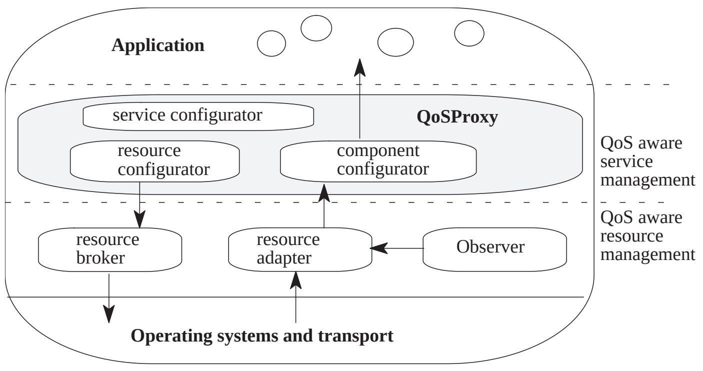

Figure 2.10: QoS aware middleware architecture

Three distinct phases of QoS management are considered: (1) the *application development phase*, (2) the *QoS setup phase* and (3) the *adaptation phase*. In the first phase, QoS is specified relating to target applications. Specifications also include functional dependency graphs constraining composition and *translation templates* defining QoS levels in terms of application level QoS dimensions. Specifications are compiled to generate *QoS profiles* (run-time representations), consisting of candidate application *component configuration*s along with *adaptation policies* and application state templates. The compilation process includes the complex task of mapping application QoS parameters to resource requirements either by analytical translations or by using *probes* running in a real system to gather information.

In the second phase, a QoS profile is first stored in the *QoS proxy* on the application server, and clients may establish bindings. This is done by service discovery and by selecting an end-to-end application configuration from the profile. First, resource brokers on each participating platform are queried for the current resource condition. The result is matched with the candidate configurations in the profile, in order to select one. After selecting the configuration and locating servers, an end-to-end resource allocation plan will be generated from the QoS profile. This is fragmented and sent to QoS proxies on each participating system which then call resource brokers to allocate resources.

In the third phase, QoS would be adjusted according to actual resource availability: This can involve resource adaptations in the lowest layer and adaptation of application components in the second layer. Here, QoS proxies can adapt application components and configurations according to adaptation policies stored in profiles. All this can be modelled as a control loop. Here, *QualProbes* do application specific QoS measurement and profiling (using values from observers and resource adapters) *Component configurators* make decisions on how to adapt the application. Configurators are specified as rule-bases and uses a fuzzy inference engine to decide on actions to be invoked on third layer objects or the application itself. The use of such rule-bases makes the middleware quite configurable.

2KQ and Agilos is prototyped on top of DynamicTAO [Kon00] and the 2K [Kon98] system and validated by applying it to a visual tracker application (OmniTrack).

OMODIS QoS architecture

The OMODIS QoS architecture [Ecklund02] aims to be more general than its predecessor. It is designed to combine various existing QoS management schemes like e.g. QuO (section 2.4.3) and to support both the adaptation and reservation approaches in combination. Important concepts are hierarchically composed policy domains and QoS managers. In OMODIS, QoS management services and managed components are separate entities, the management structures are dynamically configured based on client sessions as well as the set of (possibly nested) policy domains governing the components. The management structure can be adapted dynamically.

A *policy domain* contains a set of services governed by some common *QoS policy* and typically corresponds to an administrative domain. A domain can be nested and can typically have a long term existence. Each policy is independently updateable. *QoS managers* coordinate QoS contract negotiation and adaptation among *managed components*. QoS managers are organised in *hierarchies* corresponding to their domains. There are two types of managers: (1) *tactical managers* which controls application- and system-specific components (managed components), (2) *strategic managers* which enforces policy of a domain. A strategic manager coordinates the tactical managers within the domain and/or strategic managers of subdomains. A *session domain* exists for the duration of a client session. The initiation of a client session would trigger the creation of a session specific tree of QoS management connections among strategic managers. A session domain would be associated with a special strategic manager (session manager) which is the root of the tree.

An architecture for QoS managers is proposed [Ecklund01]. It is based on a classical feedback controller architecture, but extended to be effective in non-deterministic environments. The following extensions are proposed: (1) *Negotiation Agent* (QNA) to implement contract negotiation between clients and servers, (2) *Adaptation Agent* (QAA) to manage configurations of components and adaptations of such configurations, (3) *Admission and Reservation Agent* (ACRA) to interface with resource management functions. The QNA could use QoS negotiation protocols like e.g. [Koistinen98].

The QuA project

The QuA project [Staehli04a] investigates the idea that an open component architecture platform can effectively assume all responsibility for QoS management. Platform-managed QoS is proposed as a general solution to preserve the *safe deployment property*; i.e. applications assembled from independently developed components should function correctly when deployed on a platform with sufficient resources and services. This is meant to contribute to realising the vision that application software components could be written without dependencies on physical resources or knowledge of platform service implementations.

An important part of QuA is a framework for *implementation planning* [Solberg04, Eliassen06]. Planning is to compose a set of available software entities (components, services, resources) such that they implement a service which behaves according to a set of QoS constraints. A *planner* (which also may be a pluggable component) is responsible for discovering resources and implementation alternatives, and for planning the optimal configuration of components, in order to satisfy requirements. The result of planning is a *plan*, which is associated with a set of platform assumptions, a *blueprint* which represents a realisation of the service in terms of a set of components or other blueprints, a composition plan which specifies composition of the service from blueprints and the QoS properties the composition would result in. QoS is specified in terms of quality loss (QL) characteristics, which represent deltas of perfect (ideal) QoS. Domain specific semantics are defined as error models [Staehli04b]. User QoS constraints are specified as minimum and maximum QL values. It is proposed to use utility functions, which compute the utility of a possible service plan from the set of QL values.

2.4.3. Aspect oriented approaches

Aspect oriented programming (AOP) [Kiczales96b] is a programming paradigm supporting separation of concerns in the sense that various aspects of a program can be defined separately. This is related to the *open implementation approach* [Kiczales96a] as well as reflection (see section 2.3.3). Aspects are typically *crosscutting* in the sense that they involve many different parts of the base program design. Programs can be simplified by programming aspects in separate modules. A crosscutting concern can for instance be error handling, debugging, or distribution aspects. Handling extra-functional concerns would often be crosscutting and the AOP paradigm may be suitable for this.

Aspects may be defined in different languages (each suitable for the aspect in question). By using appropriate tools, aspects are woven into runnable programs. Aspect weaving can be done either at *compile time*, at *deploy time* or at *run-time*. Run-time weaving has some overhead with respect to performance and is still a research issue. Many implementations and prototypes have emerged, usually based on specific programming languages or middleware architectures. We refer to [Loughran05] for a survey of middleware exploring aspect oriented programming models like the one provided by AspectJ [ASJ]. Significant progress is being made in this area, for instance in treating aspects as pluggable components and in efficient dynamic aspect weaving. In the rest of this section we describe some projects which claim to address contractual QoS management with an aspect oriented approach.

The QuO project

The QuO project [Zinky97, Loyall98a, Loyall98b etc.] is motivated by a demand for distributed applications that can deal with QoS requirements, changing usage patterns and underlying resources and thereby support adaptive behaviour. The QuO architecture (QoS for CORBA objects) has been developed to support dynamic QoS management in the CORBA object layer by adapting object implementations. It extends the functional interface definition language (IDL) with a separate QoS description language suite (QDL) which capture expected usage patterns and QoS requirements for clients binding to objects.

To provide end-to-end QoS for distributed applications, system information from the server, the communication infrastructure and the client must be reconciled. The approach of the QuO project is to do this in the *service object* in the sense that the boundary of the object's implementation is moved into the client's address space. The connection boundary, where the contracts are agreed upon, is located in the clients address space. Advantages of this are that delay between client and the connection object can be ignored, and that client side parts of the object implementations never fail independently of the client.

Since the object implementations deal with distribution issues and QoS management like monitoring and adaptation etc., the QuO contract model can be understood as *implicit* (see section 2.1.3). Object implementations should therefore be opened up to some extent; i.e. one needs to identify certain component types and interfaces, to allow adaptation of those at run-time. QuO defines a model of object implementation components. In addition to the base implementation there are delegate objects representing remote objects. Delegate behaviour is controlled by *QoS contract objects*.

A *contract* object in QuO defines a set of nested *regions*, a set of references to system condition objects (which are responsible for measurement and control) and a set of *adaptation actions*, possibly with callbacks to clients (in the case of non-transparent adaptation). Regions model relevant states of the system QoS and are defined as predicates on values of system condition objects. To handle divergence between expected and provided system conditions, QuO allows specification of two levels of system conditions (regions): (1) *Negotiated regions* are regions of the requirement space (section 2.1.5) and represent QoS obligations to client. (2) *Reality regions* are regions of the resource space and represent measured QoS properties of the implementation or system resources. Contracts define mappings between reality and negotiated regions as well as transition rules. Contract evaluation determines which regions are active and is triggered by delegate objects or by changes in system condition objects which are observed by the contract. Delegates are informed about the resulting region and may adapt their behaviour according to this.

MAQS

MAQS [Becker97] also claims to adopt an aspect oriented approach to QoS. QIDL is the proposed extension to CORBA IDL which allows QoS annotations to interfaces. In QoS-enabled interface specifications one may define QoS characteristics using IDL types, and one may define operations for QoS management. Aspect weaving is realised by generating QoS *code skeletons* and *mediators* from QIDL. These may be completed by QoS implementers. Such components and other parts of QoS implementation are connected to stubs and skeletons which intercept requests on their way from the client to the (functional) server implementation. In this approach QoS is tied to interfaces, not components that implement them. Hence, new interfaces must be defined (by specialisation) in order to support different QoS for different implementations of the same interface.

AspectIX

AspectIX [Hauck01] (formerly LegORB) adopts a similar fragmented object model in the sense that objects may be decomposed into fragments to be distributed over multiple hosts, and where functional and extra-functional code may be separated into different fragments. A client binding to an object involves the loading of code fragments like stubs or delegates into the client address space. AspectIX provides tools for statically generating fragments which could be dynamically composed. Functional code written without QoS in mind can be converted to an AspectIX fragment which has the necessary hooks to be dynamically linked with other AspectIX fragments.

An *aspect* typically corresponds to a QoS category (or sub-category), it is described in CORBA IDL and represented at run-time by *configuration objects*; i.e. a configuration instance contains a set of QoS parameters representing a requirement statement. Clients may associate configuration objects with object delegates (fragments), which can evaluate configuration objects. Delegates can decide that the configuration is invalid when the requirement cannot be fulfilled, and they may decide to adapt transparently by replacing themselves.

Composition filters

[Bergmans01] introduces *composition filters* as a model of how *crosscutting concerns* (aspects) can be realised in a dynamic manner in object oriented designs. Method invocations are reified (represented as explicit objects) and matched with a set of filters when going to or from a given service implementation. Filters may for instance route invocations to alternative implementations, encapsulate invocations into other method invocations, or manipulate resource management policies. A declarative filter expression language is proposed. Some experimental work has also been carried out in applying composition filters on CORBA middleware [Bergmans00b], for instance by intercepting invocations between the IOP and the transport service, but some investigation has also been done on aspects that vertically crosscut multiple layers.

2.4.4. Flexible middleware

A fair amount of research has been carried out in developing flexible middleware architectures, motivated by the demand for distributed multimedia, mobile and ubiquitous computing, and a desire to adapt applications to a wider range of devices, environments, and usage patterns [Geihs01]. As also seen in operating system research, one has been looking at alternatives to the monolithic structure of middleware to make it more configurable. The study of reflection as a means to open up middleware [Kon02] has been particularly important. Research has also focused on structuring middleware using component frameworks [Clarke01].

Microkernel ORB architectures

Many research projects are motivated from the need to control dynamically which protocols are used for binding, as well as the management of resources, to be able to accommodate real-time requirements and a flexibility from the application programmer's point of view. Projects like ANSA DIMMA and ReTina aim to specify middleware architectures which are flexible enough to accommodate multimedia applications, based on a minimal ORB kernel where different protocols as well as API personalities may be plugged in or replaced.

DIMMA [Donaldson98] builds on a small nucleus and a generic API module supporting an API following the RM-ODP computational concepts. This can be specialised with a CORBA personality module or other personalities. Protocols (like IIOP or a RTP based flow protocol) may be plugged in at the bottom. However, the individual layers of protocols are not explicitly componentised. Resource management (buffers threads or scheduling policies) and protocol parameters may be controlled through simple QoS parameters which can be given by application components at binding time. However mapping of application oriented QoS to resource parameters is not supported by the platform. Bindings are supported by a hierarchy of *binder* components.

The Jonathan ORB [Dumant98] is a Java prototype implementation of the ReTina architecture, it aims to support pluggable binding policies and follows a microkernel architecture like DIMMA. However, protocol layers are componentised in terms of an architecture similar to the x-kernel

[Hutchinson88]. ReTina and Jonathan introduce the concepts of binding objects (explicit bindings) and binding factories (interfaces to create bindings). Jonathan supports extensibility with respect to *binding types*. Interface references may contain a binding type identifier used to select a suitable binding factory instance on the client side.

FlexiNet

The ANSA FlexiNet project [Hayton00] is a Java based toolkit for creating and (re) configuring ORB's. It allows programmers to tailor the platform for a particular application domain or deployment scenario. It provides a generic binding framework plus a set of basic engineering components (protocol layers etc. binding factories etc.) to populate the framework. FlexiNet is focused at operational interaction (RMI). Generic invocations are passed through a composition of objects that may transform them in various ways before they reach the destination object. The graph corresponds to a stack of "layers" at each side, where the lowest layer typically encapsulates communication sockets.

FlexiNet differs from protocol composition approaches like e.g. Ensemble [Hayden97] in that it focuses at flexibility on a higher level as well as the management of distribution transparency mechanisms like transaction, replication, security, mobility etc. To support transparencies like migration, relocation, persistence and to some degree security and transaction, the cluster abstraction (cf. clusters in RM-ODP engineering model) is introduced. The framework defines interfaces to the binder and generic call interfaces between layers as well as resource-pools used by the reflective layers for managing buffers and threads.

Configuration of stacks is done at binding time by pluggable binding factories (binders). A binder component implements a *Generator* or a *Resolver* interface, or both. The generator produces interface references (for interfaces to be made remotely invocable), which are associated with a server-side stack. The resolver produces client-side proxies from such interface references, associated with a client-side stack. In FlexiNet, binders can be composed and organised into hierarchies/graphs of binders, where each may take care of part of the job, or delegate to other binders according to some rules. This architecture enables dynamic choice of binders, as well as nested binding.

OpenORB/OpenCom

Influenced by the early Sumo work [Blair97] and RM-ODP [ISO95b] researchers at Lancaster and Tromsø universities explored the use of reflection and component technology in middleware in the OpenORB project [Blair01]. In principle, every application level component will offer a meta-interface that provides access to the underlying meta-spaces. This may reveal a set of meta-components which again may offer meta-interfaces to give access to meta-meta-spaces and so forth, recursively. Meta-components are not necessarily instantiated before they are needed.

The OpenORB architecture supports dynamic QoS management components [Blair00], i.e. monitoring, policing, maintenance and adaptation of QoS. The architecture identifies three types of management components: (1) *monitor*, (2) *strategy selector* and (3) *strategy activator*. A monitor collects information from the activities of components participating in a binding. It may attach to such components using reflection (for instance by inserting pre- and/or post methods). A strategy selector receives events from monitors and makes decisions based on this information. The decisions are carried out by strategy activators by manipulating the meta-spaces of each side of the binding. Strategy activators and selectors represent policies for QoS enforcement and adaptation.

Prototype implementations of OpenORB have been made using the Python language in e.g. OOPP [Andersen02]. Furthermore, a component model [Clarke01] has been developed, partially based on Microsoft COM. This is part of a middleware architecture which consists of frameworks for pluggable resource managers, protocols and binding types [Parlav03]. In summary, OpenORB/OpenCom is very customisable and adaptable. The disadvantage is that customisation requires a large amount of effort, since the mechanisms and programming interfaces are rather generic and low level.

DynamicTAO

TAO [Schmidt97] is a CORBA ORB known to deal with real time issues. DynamicTAO [Kon00] extends TAO to be a reflective middleware platform. In early releases, it supports dynamic intercepting of remote method invocations. This has recently been standardised by OMG and incorporated into TAO. DynamicTAO allows on-the-fly reconfiguration of the internals of the ORB and application components. The meta-space is reified through *component configurators* which represent the dependence relationships between ORB components and may include methods for preserving consistent configurations.

Multe

The MULTE project shares many ideas with OpenORB. Some collaborative work was made in the CORBAng project in Tromsø, Oslo and Lancaster [Eliassen99]. Much focus is on multi-party binding objects for multimedia, and the MULTE binding framework is based on the concept of *open bindings* (section 2.3.3). The architecture includes explicit bindings, binding factories, and binding mutators.

A conformance model of stream interfaces is developed, based on functional aspects [Eliassen98]. The project investigates *trading* as a mechanism to select amongst binding factories [Rafaelsen00]. A QoS management model is also outlined [Plagemann00] where binding factories may map application level QoS statements to parameters more appropriate for describing resources, which again may be mapped by pluggable middleware components to resource specific parameters and reservations. Resource adaptation is supported by the Da Capo monitoring component (where changes in underlying QoS may trigger renegotiation). Bindings may also incorporate media gateways which convert the format or QoS of a media stream, which is especially useful for multi-party streams. Prototype implementations of the MULTE ORB have been based on the COOL ORB (which is a CORBA 2.0 implementation) and the flexible protocol framework Da Capo [Plagemann94]. Da Capo allows dynamic selection, configuration and reconfiguration of protocol modules to dynamically shape the functionality of a protocol.

2.5. Discussion

Our interest is mainly how to support QoS aware binding at run-time. This involves negotiation of *QoS contracts* and configuration of the implementation of bindings. A large amount of research has been performed in the area of QoS, and we observe that much work sacrifice generality. As pointed out by [Ecklund02], many approaches are:

- *Domain specific.* Much work has been done in the context of specific networking technologies, or with certain application domains in mind, like e.g. multimedia. Domain specific solutions may increase the heterogeneity problem in distributed systems.

- *Embedded in particular technologies or components*, for instance in the handling of resources in the operating systems, in communication protocols or in application components. Type- or component specific QoS management techniques may not be easily reusable.
- Based on *traditional layered architectures*. This is not as flexible as required by open systems with components.

We focus on QoS contract negotiation in the context of *flexible middleware*, where the goal is some level of application and platform independence. In the following, we discuss related work with respect to two subtopics: (1) Contractual QoS statement and negotiation support and (2) QoS management. We also briefly compare them with our own approach (to be presented in the next chapters).

2.5.1. Contractual QoS statement and negotiation

To define and negotiate contracts, some commonly understood vocabulary and model (defining syntax and semantics of QoS statements) are needed. Rather than having one single defined model (categories, characteristics etc.), a trend is to define *application specific* models in which QoS statements (offers, requirements, contracts) are formulated. This is good if they can be supported by generic infrastructures based on a common meta-model. In table 2.1 below, we compare some work including QoS languages for declarative QoS statements. While QuO (section 2.4.3) and 2KQ+ (section 2.4.2) supports aspects of specification, QML and CQML (section 2.4.1) aims to be more general purpose modelling languages. Quartz supports pluggable application models but focuses on how parameters are defined. However, as pointed out in [Siqueira99], a QoS language could be used to define application parameters and filter components.

	QuO	QML	CQML	2KQ+	Our approach
Contract specification	Required QoS, usage patterns, adaptations regions (neg/reality)	Contracts (constraints) Profiles (offers or requirements)	QoS statements (constr) Profiles (complete QoS relations: Exp->obl))	Component descriptions: supported levels + hardware requirements	Profiles of policies
Contract negotiation	local only	Match client/server (can use negotiation protocol)	Client/server + composite environment (multiple profiles)	Compiler generation of QoS levels. (and possible configurations)	Policy trading
Dynamic QoS statement	N/A	QRR: compile to IDL types and C++ code, generic structures.	QRR similar to QML	limited (compiled QoS profiles)	Profile expressions
Adaptation	Regions with transition rules	Not directly	Composite profiles alternatives + transition rules	Compiler generation of alternative configurations	(policy/metapolicy issue)
Runtime evaluation	N/A	Yes (QRR)	Yes (QRR)	simple parameter matching (candidate configurations)	Yes (conformance matching)
Semantics definition	no	operators, decreasing/increasing keyword on numeric char.	operators, decreasing/increasing OCL constraints!	not discussed	Conformance rulebases
Composition support	N/A	no	simple conjoining define 3 operators on characteristics	not discussed	Composition operators

Table 2.1. QoS contract specification and negotiation

Contract definition and establishment

A typical approach is to define a set of contract fragments (QoS statements) at design time. These could later be composed and instantiated into contracts. Such fragments (offers or requirements) could be associated with components or interfaces to support QoS aware binding. The approach of QuO is to specify contracts as constraints on condition objects. Such constraints (negotiated/reality regions) may be seen as *QoS relations,* and adaptation is supported as predefined transitions between regions, which is useful. The reusability, extensibility and composability of contracts are limited due to how contracts are tied to object implementations, and since only simple numerical parameters are used. In 2KQ+, one can specify expectations (hardware requirements) and possible obligations (QoS levels) for components or component types, and a compiler can generate QoS mappings, candidate contracts and configurations of components. Also here, the reusability[5] of QoS specifications may be limited since specifications are tied to implementations (components).

QML offers abstractions for defining QoS statements, combining multiple dimensions of various types. Furthermore, it supports contract types, contract instances and refinement. A *profile* associates QoS constraints to elements of interfaces and can be used to represent either offers or requirements. QML does not consider the expectation part of QoS relations. In CQML, profiles are typically specified for component types and represent more complete QoS relations. A profile defines a possible contract between a component and its environment. Negotiation can involve finding a transitive closure of QoS profiles, whose expectations are met by a (conjunctive) composition of other profile's obligations. CQML supports adaptation in the sense that a profile can be composed of an ordered set of simpler profiles, along with transition rules which specify callback operations. This is similar to QuO regions but more implementation independent.

Dynamic QoS and contract evaluation

Contract negotiation involves the evaluation of QoS statements at run-time, for instance to see if a server offer satisfies a client requirement, or if the expectation of a possible contract is satisfied by a given environment. One approach to the evaluation of contract templates is to *compile* specifications into software components which can perform the evaluation. In QuO there is one such component per client binding, in 2KQ+ there is one per middleware instance (node or capsule), which takes part in the application. In QML and CQML, more generally available run-time representations (QRR) can be generated. QRRs are not tied to a particular architecture or negotiation protocol.

Remote QoS negotiation protocols typically exchange parameters, and constraints are often expressed as boolean expressions, parameter values, value sets or value ranges. In QML and CQML, constraints can be defined (at design time) as expressions over characteristics, using comparison operators. The meaning of such operators depends on the characteristic *type*. If a language supports the definition of characteristics, it may also be necessary to define some semantic aspects, for instance if a larger numeric value is stronger or weaker than a smaller one. The ability to define semantics of characteristics or contract types is typically rather limited. QML allow the use of an increasing/decreasing keyword on numeric characteristics. CQML has a richer support for defining semantics of QoS characteristics (OCL constraints) in addition to the increasing/decreasing keywords. The specification and matching of constraints can be a complicated task, and contract matching would be an instance of the constraint satisfaction problem which can be complex.

Composition

In open systems, a service or an environment would consist of different cooperating components. We may therefore be interested in deriving the resulting QoS in terms of QoS of each individual

[5]Since QoS aspects can be crosscutting concerns (with respect to type- and implementation hierarchies), it can be useful to support such sharing and re-use. This can reduce work of developers and reduce redundant code.

component. Composition seems to be little supported in most QoS work. CQML however, addresses the composition issue in the context of contracts in some interesting ways. First, composition operators on individual characteristics can be defined. Here, and in [Jaeger04], the semantics for each characteristic could be specified for each composition pattern. This may not be an easy task. Furthermore, an expectation of one profile would be matched against a composition of other profiles. Such conjoining is well founded theoretically. However, this model implies that pairs of profiles (e.g. client and server) would require a *partial* match, which is not clearly defined. Though an expectation could be satisfied by offers from various components, it may be necessary to specify additional constraints on what components offers may come from. For instance, a contract for a service may expect certain behaviour from the client component in particular, or it may require different parts of the expectation to be satisfied by components on separate nodes. Hence, simple conjoining alone is not always sufficient. CQML allows additional invariants in profiles, which deals with some notion of component identity. It is left to a QoS framework (middleware) to define component identity. It seems to be an issue for further research how composed QoS statement could express location constraints in an abstract and re-usable way. There is currently no generic solution to QoS composition, and support is typically limited to additative characteristics.

Other issues

There are some additional issues of QoS contract negotiation and modelling which are not much addressed by previous research. These issues include:

Extensibility, i.e. how the space of possible contracts and adaptations of an existing and running application can be extended. The compiler approach of QuO and 2KQ+ means that possible contracts and corresponding configurations are produced in advance. One can thus avoid some complexity at run-time, but choices and adaptations are limited and not easily extensible, since one may need to re-compile or extend agents at each middleware instance. In CQML, profiles can be replaced locally. But since profiles describe application components, the components themselves may need to be replaced. Alternatively, running implementation components could be altered e.g. by dynamic aspect weaving, or the infrastructure may change, but this is outside the scope of QoS modelling languages.

Interoperability. Components in independently developed applications could negotiate QoS contracts. To support this, a common understanding of the meaning of QoS statements is necessary. Components using the same application model (e.g. defined in CQML) are interoperable, but it is not that clear how to integrate different preexisting application models.

Consistency of QoS models. Inconsistency would mean that it is possible to derive contradicting or conflicting results from same model. This becomes an issue when languages support aspects of semantics specification and, in particular, when models are composed or integrated to support interoperability between pre-existing or otherwise autonomous applications.

Our approach

In our approach potential contracts are pre-defined as policies and negotiation is essentially to trade a policy for a binding. This model is extensible with respect to contracts. We propose an expression language in which contracts and other QoS statements can be formulated, composed and evaluated at run-time. Expressions are matched against each other for conformance. This can simplify contract evaluation. A basic idea is to define semantics as rule-bases from which conformance between any pair of expressions can be inferred. Our rule-base scheme supports checking for certain consistency and completeness problems. Rule-bases can also be designed to address some aspects of interoperability. Composition is addressed by proposing generic composition operators.

2.5.2. QoS management

An infrastructure is needed to support the establishment and the enforcement of QoS contracts. As discussed in section 2.1.4, QoS management involves the configuration of multiple components and resources *(resource orchestration)*, which is generally a complex problem. There has been much research in this area, where most approaches limit the scope to particular application domains, technologies or parts of the architecture. We focus on middleware level QoS management frameworks which aim to abstract over various system level platforms and to support a range of applications. Table 2.2 below summarises some characteristics of some relevant QoS middleware work: QuO (section 2.4.3) and 2KQ+, Agilos, Quarz and OMODIS (section 2.4.2).

	QoS enforcement	QoS adaptation	Configuration scope	Configuration time
QuO	Application specific implementation	Application specific implementation Contract objects (defined by QDL)	App. specific implementations	Design time.
2KQ	Minimum-amount reservations	Intraconfiguration adaptation Dynamic reconfiguration	Reservation parameters Application components	Compile time
Agilos	Best-effort (with control-based adaptation)	Appl. specific measurements by pluggable QualProbes. Fuzzy inference engine	Component parameters Application component Middleware services	Design time/compile time (see 2KQ)
Quarz	System level QoS mechanisms (pluggable components)	Replace mappings Requires reverse mappings (notifications to applications is possible)	Reservation parameters	Run-time
OMODIS	Control-based adaptation. Extended with reservation agent. Per domain QoS policy	Classical feedback controller Extended with adaptation agent for component configuration + negotiation	App components, middleware reservations, QoS managers (tactical/strategic)	Run-time
Our approach	(policy issue)	(policy issue)	Bindings	Run-time (policies are defined at design time/ compile time)

Table 2.2. QoS management frameworks

Static and dynamic QoS management

Static QoS management is about establishing a contract. Dynamic QoS management is about maintaining a contract during its lifetime. Some QoS frameworks perform *mapping* of application level QoS parameters to system level parameters (to be used for resource reservation) which is problematic with respect to complexity and flexibility. The Quartz architecture attempts to simplify the mapping and to provide a generic architecture by proposing three levels of translations, such that application to system level mapping can operate on a generic set of parameters.

Much research focuses on *adaptation* as a QoS enforcement technique. Adaptation is important for several reasons: It is for example known that reservation has significant issues with respect to resource efficiency (since it is based on predicted future resource usage). By adjusting the behaviour of middleware and applications dynamically, for example based on measuring the resulting behaviour or observing the actual resource availability, some end-to-end QoS could be maintained, even if the environment is best effort; however, strong guarantees are hard to achieve still. Both with static and dynamic QoS management, it may be a challenging task to find suitable end-to-end configurations. Adaptation may however be limited to adjustments of existing configurations in some cases and may use measurements which are not available at binding time (like e.g. measurements of resulting QoS).

Scope of configuration

The configuration (to enforce QoS contracts) can have different scopes. The simplest is to set or adjust *parameters* on fixed components, like time delays, buffer sizes, frame sizes, frame rates, etc. QoS management can also be supported by mechanisms in the infrastructure (middleware) itself, like interaction protocols, buffering or caching strategies, coding and encryption, etc. This requires configurable middleware (see section 2.3.3 and 2.4.4).

One may configure application implementations as well. Approaches to this include dynamic weaving of various aspects (implementation fragments) which define parts of the applications behaviour (see section 2.4.3), or composition of service components into complete service implementations. This is investigated for instance in limited contexts of web-services and workflow patterns [Zeng04, Jaeger04]. We often use the term '*planning*' about the task of composing and configuring service implementations. Automated planning in a distributed system, based on available components, resources and QoS requirements is attractive, but may be very complex in the general case. The QuA project investigates an architecture for pluggable service planners and specification of plans. The 2KQ+/Agilos allows QoS proxies at each node to select amongst compiler generated local configurations. The OMODIS architecture allows dynamic configuration of QoS managers as well as service components. QoS managers are defined for nested domains. A special domain and QoS manager is created for each client session. Each domain has a QoS policy which defines plans within that domain. The approach to end-to-end configuration and adaptation is to first try to adapt/plan within the local domain and push the responsibility for decisions upwards in the hierarchy when necessary.

Time of configuration

Configurations could be determined (1) *statically* (manually defined by the designer or programmer), (2) *at compile time*, or (3) *at run-time*. The advantage of the first approach is that a skilled designer can easily see solutions which are hard to find algorithmically. However, it may be hard (depending on the application) to manually produce and manage a potentially large numbers of configurations to cover all possible situations.

Run-time configuration can handle situations not foreseen in advance, but the computational complexity can be problematic. Even if the parameter mappings are done on generic parameters (like in Quartz), it is still a complex problem and it is hard to envision one single mapper or rule-set which generates good solutions in all situations. A compiler approach can be a reasonable compromise. The 2KQ+ approach demonstrates this, and also how this can be used to automate testing candidate configurations on a real system, which could be a tedious task for a system designer. Even if defining solution candidates in advance, one may still need to negotiate, in order to select from the precomputed candidates. This may involve conformance checking etc. which could have some complexity. In distributed systems with autonomous components, it will be a trade-off between predefining end-to-end configurations and predefining part-solutions (and composing these parts later).

Our approach

We do not propose a complete solution for QoS management but rather a framwork where policies could encapsulate many QoS management issues. Bindings is the scope of configuration. This is flexible in the sense that it can capture most issues. Furthermore bindings make it straightforward to associate configurations with contracts.

2.6. Concluding remarks

In this chapter we gave an overview over some relevant concepts, problems and relevant research in the area of QoS aware open systems. This includes QoS specification, negotiation and QoS management. Our main focus is on how QoS aware binding can be supported at the middleware level and, in particular, how QoS can be defined and composed for the purpose of negotiating contracts.

We first gave an overview of the concepts and requirements of QoS aware open systems and in particular, the role of bindings and contracts. A *contract* states that a given *obligation* will be satisfied as long as the environment satisfies a given *expectation*. Binding establishment means to *negotiate* a contract and establish and configure the system such that the contract is enforced. With our focus it is relevant to look closer at two problem areas: (1) contractual QoS statement and (2) QoS management by infrastructure. First, we need to define the syntax and semantics of QoS statements, including contracts and run-time statements used in negotiation. Many previous QoS architectures focus on particular application domains or technologies, and they are based on a priori defined models with fixed sets of parameters. Proposed QoS modelling languages allow more generic QoS frameworks to be specialised for specific applications. A QoS model should support effective and efficient evaluation of dynamic QoS statements, it should support conformance checking (or other criteria for ordering), and it should support composition.

There are still unresolved issues in defining the semantics of QoS statements. Semantic specification is to some extent addressed in QoS modelling languages but mostly on individual QoS characteristics. We are interested in how to describe *compositions* (possibly made at run-time) in terms of participating components. We may also need composition for dynamic QoS statements, since bindings may be supported by compositions not foreseen in advance. The typical approach is to specify composition semantics for additive characteristics, possibly with alternatives for a set of different composition patterns. Simple conjoining of QoS statements and additive characteristics may not always be sufficient and how to handle composition in general is still an open issue. Other issues in the area of contractual QoS statement include interoperability of different application models, consistency of models and how to extend the adaptation space of a QoS models.

Contract negotiation also needs QoS management support from an infrastructure. We discussed some aspects of infrastructure support, including system level and middleware level QoS management issues as well as configurable middleware, and we looked more closely at some concrete related work projects on QoS-languages and QoS-middleware. We distinguish between static and dynamic QoS management which corresponds to establishing and maintaining contracts. Furthermore, there are two fundamentally different approaches to QoS enforcement: (1) mapping application QoS to system level QoS contracts which can involve reservation of resources and (2) dynamically adjusting behaviour by observing resource availability and resulting QoS. Both approaches have limitations and advantages.

Determining end-to-end configurations is still a challenging task. There is a trade-off between predefining end-to-end solutions manually (which may not be flexible enough) and searching for solutions at run-time (which may be computationally hard). Approaches to this problem include to specify a set of alternative contracts (along with enforcement policies) at each participant, to be matched at run-time, and to generate candidate configurations from specifications by using a compiler.

The next chapters present our own approach which addresses some of the issues discussed here.

Chapter 3.
Overall Architecture

In this chapter we give an overview of our proposed architecture for binding in QoS aware open systems. We introduce the concept of *policy trading* to facilitate the selection of policies. Policy trading is based on the idea of ODP trading [ISO97], but rather than functional interface references we trade policies. Policies encapsulate contract templates and instructions on how to set up bindings (possibly including aspects of the implementation). Rather than matching interface types, we match user requirements with *user profiles* (extra-functional properties of resulting bindings) and environmental descriptions with *service profiles* (requirements to environments applications are running in).

Section 3.1 outlines and motivates how we understand binding, contract and policy. Section 3.2 defines more precisely the concept of policy. Section 3.3 discusses briefly the concept of metapolicy which tells how to negotiate and/or adapt. Section 3.4 introduces the idea of policy trading and discusses how this can be used in negotiation. In section 3.5 we conclude.

3.1. Architecture overview

The architecture described in this chapter represents our approach to QoS aware binding. Our architecture is based on some of the concepts in the RM-ODP model (see section 2.1.1), in particular the concepts of interfaces and binding. We also assume that the reader has some knowledge of the most important concepts of component models.

3.1.1. Binding concept

Extra-functional behaviour should be allowed to be negotiable, and negotiable behaviour should in principle be orthogonal to service types. Our approach to this requirement is to place the responsibility for negotiable behaviour in the *binding*, which (at least conceptually) is an entity separate from the service itself. A *binding* can be viewed as an association of a service implementation and its associated state, with a *client* using that service. More generally, a binding can involve multiple participants (service implementations, clients, senders, receivers, etc.), sharing the properties of the binding. Furthermore, a binding should represent a *contract* (in the *extra-functional* sense); i.e. the service behaves according to certain constraints if its environment behaves according to certain constraints. In our architecture, the QoS is *activated* by the binding; i.e. the process of establishing a binding may configure the implementation, communication path and associated resources, according to the contract.

Binding types

The type of an interface captures the functional behaviour of a single interface instance of a single component. This is not enough to capture the behaviour of a *binding*, since that may involve multiple interfaces (at least two sides), and since there may be more than one way to interact with an abstract service. To capture various patterns of interactions in open and distributed systems, we adopt the concept of *binding type* [Parlav03]. A binding type is defined in terms of constraints on type and number of participants and what *roles* they play in the binding, but also how bindings are established and controlled. An obvious example of a binding type is the RMI binding which defines two participant roles: Client and server, and where interactions of interest are individual method invocations. Note that even if a RMI binding type may have participants of different types (different operations in service interfaces), we could describe it as one single binding type. A more restrictive view on binding types would be to include the exact service type as well. A more pragmatic approach may be to define a binding type for a range or a set of service types, for instance all types which include certain operations. This supports viewing the concerns of bindings as *crosscutting concerns* (see section 2.4.3).

Service deployment vs. client binding

We distinguish between the *deployment* of a service and *client binding* to it. Service deployment means to make an abstract service available for clients to bind to, by generating a name (interface reference) and configuring a minimum of protocol stack, such that clients can initiate negotiations and establish bindings. Since service deployment would involve binding a name to some implementation, it would also make sense to refer to it as '*server side binding*'.

There will exist some entities in the binding architecture which are not negotiable at the time of client binding, since they are set up at deployment time or earlier: (1) the *interface*, (2) object *identity* and shared or persistent *state*, and (3) the *platform* whose capabilities would influence what could be negotiated. In many cases there will (4) exist a non-negotiable *base implementation*, for instance in the form of a deployed component. Negotiation can however result in changes or additions to implementations or platforms. The first two entities plus (possibly) an active base implementation are (in ODP systems) typically identified by *interface references*.

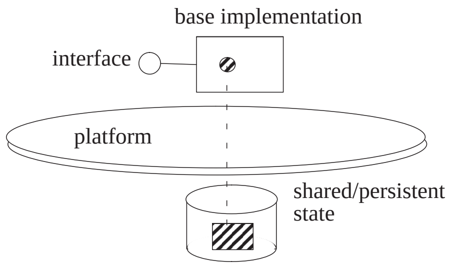

Figure 3.1. Non-negotiable architecture components

3.1.2. Contracts and policies

In related work (see section 2.4.1), *QoS profiles* (contract templates) are typically specified for service *interface type*, or *component types*. This allows QoS constraints to be associated with the individual interaction types specified by the interfaces. Furthermore, instances of service interfaces or components could be associated with profiles which can be used to compose a QoS contract when performing binding or deployment. Our approach to contracts is that QoS profiles are specified with respect to *binding types* rather than interface or component types. This allows a more explicit contract model (cf. figure 2.1). Besides supporting non-operational types of bindings more directly, this approach allows contract templates to be separated from application components and managed by a binding facility. This directly supports the idea that QoS is a crosscutting concern (since binding-types do not have to be strongly tied to interface types), and it has a potential to simplify QoS management and extensibility with respect to possible QoS contracts.

Figure 3.2 below illustrates the exchange of QoS statements between binding participants during negotiation: The application requirements and the environmental descriptions are in principle collected and composed by a binding facility and could be directly matched with policies to be introduced next.

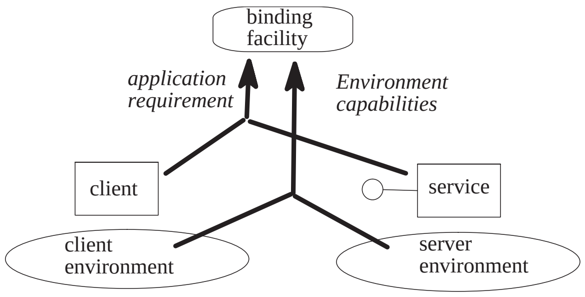

Figure 3.2. Flow and composition of dynamic QoS statements

Policy

A policy is an entity which consists of two parts: (1) a *contract template*, which is a QoS relation, i.e. a potential obligation (offer) and an expectation towards the behaviour of the environment, and (2) an entity representing a configuration of implementations and/or a set of system level resource requirements, to enforce the contract. The policy concept combines a contract template with a particular implementation into a single entity. Also, policies are specified statically and their implementations are closed[6] as seen from the negotiation's point of view. This approach could simplify the negotiation process in the sense that searching for a contract and searching for an enforcement policy are done in the same operation. In section 3.2 we define the policy concept more precisely. In section 3.4 we discuss the concept of policy trading and how this could be used for negotiation.

3.1.3. Activation and adaptation

To clarify the difference between policy and policy management[7], we propose a distinction between bindings and *activations* of bindings. A binding can either be *active* or *passive*. Passive bindings need to be activated before interactions may be carried out. Activation means allocating resources to a binding according to a policy. This may involve activating the service object itself, i.e. loading it (and its class) into memory, setting up protocol stacks, transparency objects, buffers and other resources needed to carry out interactions. Activations may be replaced during the

[6]This is strictly from the negotiation point of view. Policies may still be implemented using an open component arcitecture, to support re-use of implementation components.

[7]Policy management issues include how and when policies are selected, installed, removed and replaced. This is further discussed in section 3.3.

lifetime of the binding. When a binding is established, it is not necessarily active but it is associated with a *metapolicy* which knows how to negotiate what policy to use for activation.

Adaptation involves re-negotiation and re-activation. Note that this does not necessarily mean that policy replacement will replace the whole activation at each re-negotiation. Policy implementers could inspect and re-use parts of existing activations, but that is mainly an implementation issue.

Figure 3.3 below summarises the ideas of binding and binding management. The binding is managed by a *metapolicy*. A metapolicy decides how and when to activate (or deactivate), how policies are selected and (possibly) how to react to changes of the environment (adaptation). The metapolicy uses information from applications and environment and governs selection and installing of policies. Objects representing policies can be viewed as activation factories since they are responsible for setting up activations for the binding. The binding uses the activation to carry out invocations. Running activations may give feedback[8] to the metapolicy which can use this information to decide on adaptation actions.

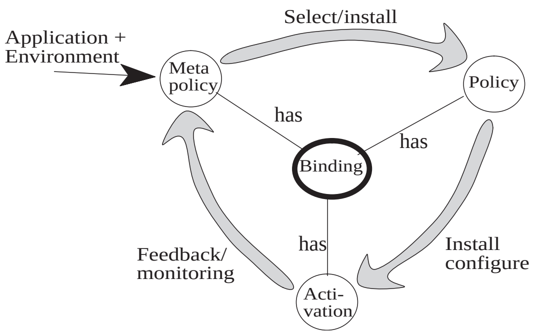

Figure 3.3. Adaptable binding model

3.2. Policy

In this section we give a definition of the policy concept. A *policy* is an entity which consists of (1) a QoS *contract template* and (2) a corresponding *enforcement policy* (policy implementation) which defines the composition and behaviour of a potential *activation* of a binding and (possibly) system level QoS requirements. A binding is responsible for selecting a policy during its establishment or when the binding is first used and for re-selecting the policy due to adaptation.

3.2.1. Contract template part

A *policy p* can be viewed as a mapping from some constraint on the environment, *S,* to the satisfaction of an user requirement, *U*.

$p{:}S \to U$

We refer to *U* as the *user profile* and to *S* as the *service profile*. *p* denotes a potential contract between the potential binding and potential users. *p* can be understood as a logical implication saying that if a requirement to the environment is satisfied, the policy will satisfy certain requirements.

[8]Feedback loops as an adaptation technique is not our main focus though.

$P(S) \Rightarrow P(U)$

Here, P is a predicate which represents some interpretation of S and U such that the implication is equivalent to a *QoS relation* (section 2.2.1). In chapter 4 we use the definition of P in the formal definition of the semantics of a language for dynamic QoS statements.

User and service profiles

The *user profile U* denotes the willingness to satisfy a requirement for extra-functional properties of the binding as perceived by users or applications. For example, it may state that the binding can guarantee a certain level of confidentiality and/or integrity of the exchanged information, or that messages are delivered within certain time-limits.

The *service profile S* represents an assumption on how the environment E behaves. If S is satisfied by E, the policy will enforce U. Service profiles may be statements about availability of services, the availability of resources which supports the engineering of bindings or other QoS properties of environment services. For instance, a policy may require the availability of a secure communication channel or a certain amount of available buffer space.

Contract establishment

The clients of a binding (which in a RMI setting is the client of the service as well as the server component to be bound to), has a *requirement, R*, for the binding. The environment (which may consist of multiple platform components) has some properties expressed as an *environment descriptor, E*. The process of *negotiation* is the process of establishing an agreement on a policy p such that

$U \leq R \wedge E \leq S$ *where '≤' denotes the satisfaction relationship.*

Simple requirement composition

Consider a client/server scenario (figure 3.4 below) where U is the user profile and S is the service profile of a policy p. R_C and R_S denote the (user or application level) requirements from the client and server respectively. For instance, the client may require a low latency time for its requests and the server may require that the communication preserves confidentiality. In order to use p as the policy for the binding, U must satisfy the combination of the requirements R_C and R_S. In order to express such combined requirements, we introduce the '+' operator such that the combined requirement is expressed as $R_C + R_S$. The fact that U satisfies the combination R_C and R_S, we express as follows:

$$U \leq \left(R_C + R_S\right) \Leftrightarrow U \leq R_C \wedge U \leq R_S$$

Combining environments at different locations

The environment of a binding is a combination of the environments of the locations of the participants, including communication services which connect them[9]. The service profile, S, of a policy must be satisfied by environments at both sides of a binding; i.e. it must be mapped to requirements which are valid at each location. S holds if satisfied by a composition of E_C and E_S where

[9]In the simplified description here, we assume that the communication channel is part of either the client side or server side environment.

E_C and E_S are the environment of the client and server respectively (see figure 3.4 below). In order to express composition (cf. section 2.2.1 and 2.2.3) of different locations like this, we introduce a special operator, '⊕'. The fact that the combination of E_C and E_S satisfies S we thus express as follows:

$$\left(E_C \oplus E_S\right) \leq S \iff E_C \leq S_C \wedge E_S \leq S_S$$
$$where\ \ S = (S_C \oplus S_S)$$

Note that using the '⊕' operator has a different meaning than using the '+' operator in the sense that each side of it apply to different components locations. Each side is treated separately with respect to the satisfaction relationship. In essence, this means that a composite service profile must be satisfied by a composite environment descriptor where each component's constraint is satisfied separately. This also means that a policy for a typical client/server setting can consist of at least two parts where one part is for the client side and one is for the server side. Each of these has a separate requirement for the environment, hence we can split S into S_C and S_S. The '+' and '⊕' operators are used in the definition of a general profile expression language in chapter 4.

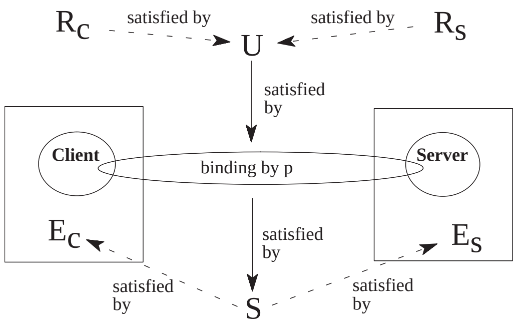

Figure 3.4. Satisfaction relationships and composition

3.2.2. Enforcement policy part

In addition to the contract itself, a policy also denotes a way to enforce its contract, i.e. a configuration of resources and mechanisms. We refer to this as the *policy implementation.* In practise, this could be a software component (for instance a script or a pre-compiled program fragment). This is executed to do the configuration (activation) of a binding.

Binders and activators

As a consequence of the distinction between bindings and their activations (cf. section 3.1.4 above), we identify two types of pluggable software components for binding management: (1) *Binders* which establish bindings and (2) *activators* which activate them according to some policy. Typically, a *binder* generates names for a service on the server side (service deployment) and resolves such names to proxy objects on the client side, as well as associating these proxies with communication and negotiation support needed to activate the binding. Activating bindings involves loading and instantiating of *activator* components.

Component frameworks

To support the goal of orthogonal and negotiable extra-functional behaviour, policies should to some extent be portable across different platforms or platform instances, and the platform should provide the necessary information to decide if a policy can be used there or not. The platform should therefore provide a component framework in which policy-implementation components can be plugged. This framework should give the policy implementers the tools necessary to enforce contracts by instantiating and configuring implementation aspects. A concrete middleware platform and component framework supporting pluggable policies, is presented in detail in chapter 5.

3.2.3. Meta policy

Metapolicies (policies for policy management) specify how binding policies should be selected, when activation should happen, and (possibly) how bindings may be *reactivated* using replacement policies, in response to changing system properties. The metapolicy concept capture parts of binding establishment and binding management aspects of the binding type (see section 3.1.1 above). A given binding type may allow more than one metapolicy, but the binding type would constrain what metapolicies are suitable. A metapolicy is installed at the time of service deployment and involves implementation decisions which constrain the later choice of policy. Aspects of metapolicy include:

- Passivation of the binding after a certain time of inactivity.
- Pre-activation of bindings which are likely to be used in the near future.
- Changing policies dynamically to adapt to changing resource availability, usage patterns and Quality of Service from the network. This may also include policies for degradation of the QoS delivered to the application.
- Negotiation between clients and servers of what policy to use.
- How to prioritise between candidate policies during negotiation.

An important metapolicy issue is how to decide or negotiate what policy to use for activation. A principle which metapolicies (or more specifically, negotiation protocols) can be based on, is *policy trading*. This is further discussed in section 3.3 below.

A metapolicy is implemented by a pluggable *binder* component, which enforces certain decisions on how to set up a binding. It may associate proxies or other transparency objects with meta-level components like monitors, strategy selectors or negotiation protocols. In this thesis we do not focus on dynamic selection of metapolicies, instead we adopt an approach (cf. chapter 5) where binder components are specified by the programmer or selected from protocol information in the interface reference.

3.3. Policy trading

We propose an architecture where policy implementers offer their policies to potential binders through a trading service. In this section we define policy trading and discuss how it can be applied to the negotiation problem. Policy trading is the process of finding a policy whose user profile and service profile matches the requirement R and the environment E. Trading may be viewed as a mapping, *trade* from a user requirement R and an environment E to a policy P:

$Trade{:}R \times E \rightarrow p$

Like in ODP trading, two operations are important: (1) *export* which registers a policy with the trading service and (2) *import* which returns a policy (maps directly to the trade primitive above). ODP trading is however limited to the locating of service implementations, and it does not incorporate expectations towards environments. Policy trading can be considered a generalisation in the sense that it can deal with all parts of the binding and that it can explicitly include the expectation part of contracts.

3.3.1. Policy trading and composition

In the simplest model, one single policy entity bundles all activator components necessary to activate a binding end-to-end, and these components are associated with one single contract template. We can define the policy selection function *Trade (R, E)* as follows:

Find a policy $p{:}S \rightarrow U$ such that $E \leq S \wedge U \leq R$

An open and distributed environment is a *composition* of environments, having different properties. Also the participants of a binding may have different requirements. Hence, it is useful to analyse how policies and the expressions used to describe them can be composed. A simple approach is to select a policy only at one side, which can be useful if for example the server side is non-negotiable with respect to client sessions.

Single side policy selection

In some cases we trade on one side only (typically the client side). The policy at the other side is shared between client sessions, fixed and known (possibly decided at service deployment time). Aspects of the server side policy (e.g. what protocols it supports) may be part of the environment of the client (E_C). We can describe single side policy selection, *Trade* $(R_C + R_S,\ E_C)$ as follows:

Find a policy $p{:}S \rightarrow U$ such that $E_C \leq S \wedge U \leq R_C + R_S$

Composite policy selection

Generally, we could trade tuples $<p_1, .. p_n>$ where each p_i is an a policy to be used by a participant *i,* of the binding. If we assume that there are two participants, e.g. a client and a server we can describe composite policy selection, *Trade*$(R_C + R_S, E_C \oplus E_S)$ as follows:

Find a policy pair $\left(p_c{:}S' \rightarrow U', p_s{:}S'' \rightarrow U''\right)$

such that

$E_C \leq S' \wedge U' \leq R_C + R_S$
$E_S \leq S'' \wedge U'' \leq R_C + R_S$
p_c is interoperable with p_s

The last requirement (p_c is interoperable with p_s) means that the policy components at each side must be able to interoperate with each other to fulfil their task, i.e. they use compatible protocols. It may be tempting to realise composite policy selection by automatic matching of such tuples at import time. One possible approach is to include the requirement for the policy of the opposite side in the service profile, i.e. it is regarded as part of the environment. However this implies a complex search problem. Obviously, the worst case complexity would be $O(n^2)$ for matching policies for two-party bindings and $O(n^m)$ for bindings with m parties (if we also assume that conformance checking is of complexity $O(1)$).

For most of the examples, we adopt a the simple approach where each traded policy p is one single profile pair associated with a bundle of activator components. It may also be convenient in some cases to trade repeatedly for each joining participant (section 6.2). The contract specification and search for end-to-end solutions are now less complex, but we still have to deal with problems like mapping activators to participants. Given a client/server model we now have:

Find a policy $p{:}S \rightarrow U$ such that $E_C \oplus E_S \leq S \wedge U \leq R_C + R_S$

3.3.2. Binding protocol issues

When different locations are involved in binding there is a need for a protocol to collect requirements and environmental descriptions from participants of the binding and to safely install and activate policy components in the participant's capsules. When looking at architectures for binding, an issue is *where* to locate the trading service, for instance on the client side or on the server side. The binding protocol depends on the structure of the binding and the metapolicy, so there is not one single solution to this. In the following we briefly discuss two client/server scenarios. In chapter 5 we further evaluate how a middleware architecture could support the policy trading approach as well as how binding-types with multiple participants (dynamically changing number of participants) may be supported.

Single side policy trading at client side

In some cases, the configuration of implementation on the server side can be regarded as fixed (decided at deployment time) and consequently, all configuration must be done on the client side. In this case, there is one policy component (one activator) to select, and it is selected (traded) on the client side when binding to a (remote) interface. The server has a policy component installed statically; i.e. this architecture implies single side trading (cf. section 3.3.1). The policy of the server side limits what could be used on the client side. For instance, a server policy may include concurrency control and logging (for recovery) and require bindings to it to be transactional. Exported interface references could if necessary indicate such server imposed requirements, which must be taken into account when trading for a policy.

The traded policy may include (partly) configuration of the server side: i.e. the policy on the client side may invoke an operation on the server to set some session parameters. However, the limitation of this architecture is that it does not take the server side environmental properties into consideration in negotiation or re-negotiation.

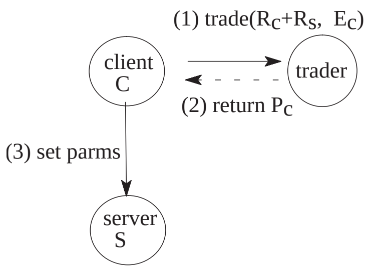

Figure 3.5. Single side trading scenario

Policy trading for two sides (simple negotiation)

To realise negotiation which allows the environments of both sides to be taken into account, we need at least one request and response to be sent between the client and the server. In figure 3.6 below, we show how this could be done by using policy-trading on the *server side*. First (1), the client requests a *bind* operation where the requirements and the environment descriptor are arguments, at the server. The response contains the policy. The server may also have some requirement R_s and it composes a requirement which satisfies both user profiles (the sum $R_c + R_s$). The server then invokes the trading service (2) to find a matching policy. It installs the server activator component of the result (4) before it returns the client component to the client (5) which then installs it (6).

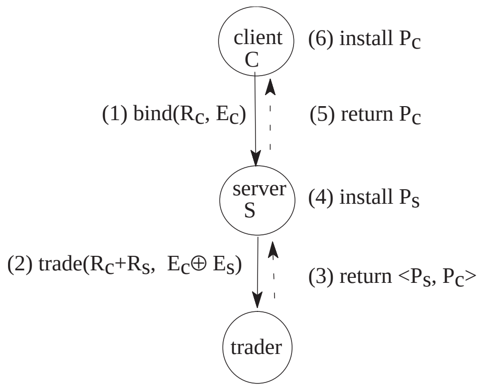

Figure 3.6. Double side trading scenario (negotiation)

Doing trading on the client side will require another invocation to be made after the negotiation phase, to install the server side policy component. It is therefore not the most efficient approach in the simple client/server scenario. However, when issues like resource reservation, admission control or worth based selection from a set of conformant candidate policies is to be taken into consideration, the picture would more complicated. This is further investigated in chapter 5.

3.3.3. Dynamic QoS expression

An important part of policy trading is the evaluation of dynamic QoS expressions with respect to conformance. As pointed out in section 2.2, dynamic expressions need to carry the information necessary to perform automatic conformance checking and should be composable as well.

A possible prioritisation of conformant policies, for instance to optimise resource consumption or to maximise the user QoS, is outside the scope of our trading model. The main requirements of our dynamic QoS expression language can be summarised as follows:

1. That expressions can be efficiently compared at run-time for conformance; i.e. we want to check if one expression satisfies another.
2. That expressions, describing different components, or coming from different participants, can be composed at run-time to represent the behaviour of the resulting system. This is also a consequence of the discussion in section 3.2.1, where composition operators ('+' and '$\oplus$') are suggested.

An additional goal is to minimise the amount of exchanged information at run-time as well as the model needed to be agreed upon in advance by the negotiating participants. Other issues are how to compile and check the consistency of models and how independently defined models can be integrated (interoperability).

3.4. Concluding remarks

In this chapter we propose a model of QoS aware binding and negotiation: The main parts of our approach are as follows: (1) to associate contracts directly with bindings, (2) policies as contract templates, (3) policies as QoS enforcement policies and their implementations as pluggable software components, (4) distinction between binding and their activation and accordingly, distinction between policies and metapolicies, (5) policy trading as a basis for negotiation protocols, (6) the requirements for a dynamic QoS expression language with support for conformance checking and composition operators.

We propose a distinction between bindings and their *activations*. Extra-functional behaviour is enforced by possibly replaceable activations. A binding uses a *metapolicy* which dictate how to negotiate and adapt. This distinction may simplify the understanding of a QoS contract and adaptation. The ideas of explicit binding, binding types and the direct association of contract templates with binding types rather than with components or service interfaces, support the view that QoS contracts can be realised as a crosscutting concern.

One of the typical problems of QoS negotiation in related work is how to deal with the *complexity* of defining and finding contracts. Our approach is to allow end-to-end solutions to be specified in advance as policies. Enforcement policies can be pre-implemented with one or more contract templates in mind. The set of available policies can be registered in a trader (database) as well as conformance relationships which are statically defined. A policy encapsulates a contract template as well as an enforcement policy, and it can therefore be viewed as a complete binding template. To establish a binding is simply to find a suitable policy and install it. Policy trading is our approach to negotiation and is based on determining conformance relationships between dynamic QoS statements (service profile vs environment descriptor and user profile vs. user requirements). However, there are issues in how to compose a policy from activator components to be distributed to distributed nodes participating in a binding. How this is solved, depends somewhat on the infrastructure and the binding type of interest.

This chapter has mainly given an overview of our architecture and leaves open some issues on how to design infrastructure support and how to design models for QoS statements to be used in negotiation. In chapter 5 we show how to support negotiability and adaptability in a reflective middleware platform and component framework supporting pluggable policy components as well as bindings which represent negotiable behaviour. But first, we investigate the problem of dynamic and composable declarative QoS statement. In chapter 4 we define a core model for such statements.

Chapter 4.

Core Profile Model

In this chapter, we propose a core model and an expression language for QoS statements (descriptions of requirements, offers, contracts etc.), mainly to be exchanged during negotiation. Any pair of expressions in this language can be evaluated at run-time for conformance (if one expression satisfies the other). Our model is partly based on *declared conformance*, meaning that conformance can be defined explicitly between identifiers, rather than doing complex comparison of parameter values or value ranges.

Our model also addresses *dynamic composition*; i.e. expressions can be composed from simpler ones, possibly originating from different components of a system. *Complex expressions* are constructed from atomic expressions (termed *basic profiles),* using composition operators. Basic profiles and rules for conformance are expressed by domain experts as *concrete models,* which are defined for specific application domains. Our focus here is on an abstract *core profile model,* which defines how concrete profile models are defined and how conformance can be inferred from concrete models. We also define how expressions that combine basic profiles relate to each other, thus allowing complex QoS statements to be formulated and compared at run-time.

The rest of this chapter is structured as follows: Section 4.1 gives some overall definitions and motivation, and section 4.2 defines the fundamentals of profile models. In section 4.3, we define how complex expressions are built from simpler ones, by using two composition operators: '+' (sum) and '$\oplus$' (component-sum). We develop conformance rules and a conformance evaluation algorithm for expressions. In section 4.4, we describe how concrete models can be defined as rule-bases. We also develop an experimental compiler which converts rule-base descriptions to testing-code, and we show how this involves computing derived rules from axiom rules. In section 4.5, we analyse our model with respect to how we can check profile models for consistency problems, as well as how interoperability and composition can be addressed.

4.1. Background

A main motivation for our approach is the need for negotiation of extra-functional behaviour resulting from *composition* in open systems. Composition may involve deployment of components in some environment, binding between components, etc. Either case would involve a decision on a *contract* and a corresponding configuration of implementation aspects, interaction protocols and resource management, in order to realise the composition. A goal of negotiation would be to reach agreements between possibly autonomous parties, on contract and configuration. The negotiation process may involve exchange of statements (offers and requirements).

In chapter 3, we introduce the concept of *'policy'* as an encapsulation of a potential contract and a corresponding *implementation* or *enforcement policy* [Hanssen99]. Furthermore, we propose to use *trading* of policies as a principle of negotiation [Hanssen00]. For a given service, there would exist a set of policies, each stating an *offer* and an *expectation*. The goal of negotiation is to find a policy whose offer satisfies the user requirement while its expectation is satisfied by the environment properties.

Our proposed language is meant to be used for the following: (1) Expressions representing user or application requirements, (2) *'environment descriptors'*, which are expressions describing environments, (3) *'user profiles'*, which are expressions representing offers of policies, and (4) *'service profiles'*, which are expressions representing expectations of policies. All such expressions are termed *'profile expressions'*.

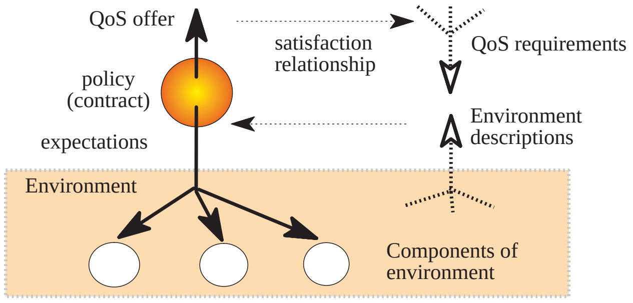

Figure 4.1. QoS statements and satisfaction relationships

The relationships between user requirements and offers, as well as the relationships between the QoS expectation and the capabilities of the environment, are *satisfaction* relationships. To facilitate conformance testing, our language should define a *partial order* on such expressions with respect to satisfaction. Thus, any statements could be mechanically evaluated for conformance. Figure 4.1 illustrates which roles profile expressions play in negotiation based on policy trading.

4.1.1. Profile models and declared conformance

Profile expressions are formulated according to a *profile model* which defines the vocabulary of expressions and rules for how expressions relate to each other with respect to conformance. A profile model would typically be defined for an application or application domain, but parts of it may also be shared between application domains. Traditionally (cf. e.g. [ISO95]), QoS statements are typically predicates formulated explicitly as constraints on parameter values. Models are typically defined (for instance in QML [Frølund98a] or CQML [Aagedal01]) as sets of QoS characteristics, and contract-templates. Negotiation can be a complex task of matching parameter values, and possibly mapping between different abstraction levels.

To reduce the computational complexity of matching QoS statements, it looks appealing to adopt the technique typically used in ODP trading [Bearman93, ISO97] where each requirement or offer is a reference to a type name, and where type conformance is declared a priori. This way of using declared conformance for negotiation was proposed in [Hanssen98]. Here, a profile model is defined as a set of simple names (profiles). Conformance relationships are declared explicitly. Conformance checking at run-time is thus very simple, compared to evaluating a potentially complex set of QoS parameter values and constraints against each other. Figure 4.2 shows a simple example of a profile graph for an e-mail application. Users can specify requirements for message delivery which are mapped to this graph. For instance, '*Authenticated*' is a subprofile of '*Secure*', i.e. it satisfies the requirements of '*Secure*'.

We consider this simple scheme to be too limiting since each declared type would need to capture all aspects relevant for the application. This may lead to conformance graphs which are too complex and application specific. Obviously, profiles would need to capture all relevant QoS dimensions, and in some cases, one may need to define a large number of profiles, for instance to cover all relevant values of a variable. Examples of this include bandwidth or latency time constraints where the differences within a group of profiles are just a numeric value, and it would be more efficient and readable to use numeric metrics.

To address these problems, we propose a *compromise* between simple declared conformance and parameter-based conformance. First, we propose a scheme for dynamic composition (see section 4.1.3 below). Second, we propose to allow profiles with simple numeric parameters. We may introduce profiles which takes one or more parameters from totally ordered domains (numbers) along with predicates denoting rules for how conformance relates to the parameters.

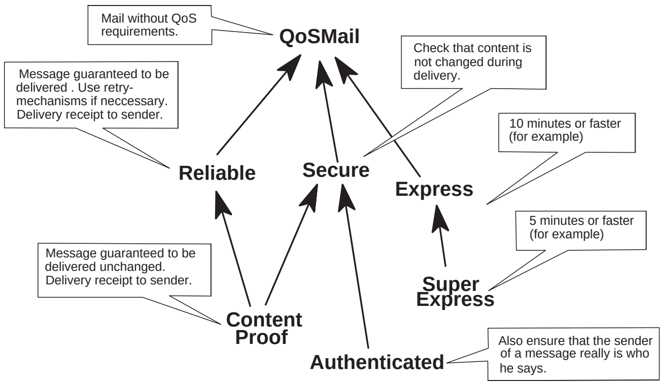
Mail without QoS requirements.
QoSMail
Check that content is not changed during delivery.
Message guaranteed to be delivered . Use retry-mechanisms if neccessary. Delivery receipt to sender.
10 minutes or faster (for example)
Reliable
Secure
Express
5 minutes or faster (for example)
Message guaranteed to be delivered unchanged. Delivery receipt to sender.
Super Express
Content Proof
Authenticated
Also ensure that the sender of a message really is who he says.

Figure 4.2. Example profile graph

4.1.2. Dynamic composition

QoS expressions and QoS negotiation should support *composition,* in the sense that expressions from components of a system which do not necessarily know each other can be combined into one, describing the composed system. Given statements about the behaviour of individual components, it is not obvious how to infer the behaviour of the total system. There are three different problems to be addressed when it comes to expressing the total behaviour:

- Autonomous users may issue different requirements for the same object and all users should be satisfied. For instance, the participants in a binding may have separate requirements for its QoS.
- We may need to combine expressions regarding the same component but in more than one dimension.
- Open systems are systems interacting with environments neither they or their implementers control [Abadi94]. Expectations may need to characterise a number of abstract components, for instance client, server and communication channel (with a separate QoS expectation for each). Our model should thus support dynamic composition of statements about different components of the environment.

To address the first and the second problem, we introduce an operator to construct statements from simpler sub-expressions meaning that the predicates stated by each part must be true in the same environment (cf. logical conjunction). We address the third problem by introducing an additional composing operator. It is partly addressed in QoS specification models like [Aagedal01], by allowing QoS-characteristics to be defined with such composition in mind. Work on formal models has shown that with certain assumptions on the temporal relationships [Abadi93], it is possible to make statements about the behaviour of composite systems as conjunctions of statements about each component.

4.2. Fundamentals

We are interested in defining models that let us state QoS and resource requirements as simple expressions. In addition, we will introduce operators for combining simple statements into complex expressions. Profile models define a (possibly infinite) set of values called *basic profiles* and how these values are related to each other by conformance. In this section we introduce basic profiles.

Definition 1 (profile expression and environment): A profile expression[10] x denotes a predicate $\sigma(x)$. σ represents the interpretation of the profile expression x in a specific environment. □

In the following definitions we use the term *'environment'* about σ. A requirement or offer may be stated as a reference to the profile by name, for instance by saying *'HighBW'*. The actual definition of σ is implicit in negotiation, but may be needed by the implementers of application components. For instance, *'HighBW'* may be defined as *"(bandwidth >= 10)"* in a particular environment, meaning that a measured bandwidth of a communication channel should be higher than 100 MB/s.

Definition 2 (basic profile expression): A basic profile a is either a symbol s_a or a pair consisting of a symbol $s_a \in$ *Identifier* and a parameter $t_a \in D_a$ where D_a is a tuple domain $[T_1, \ldots T_n]$. □

A profile may have parameters[11], which would be given some interpretation in the context of the profile symbol. For instance, *'Rtt[100]'* may denote a mean round-trip delay of 100 milliseconds. For the rest of this discussion, we limit the domain of parameters to integer numbers.

Definition 3 (Direct conformance relationship): A basic profile a is *satisfied* by (it is a *subprofile* of) a basic profile b iff for all possible environments σ, the predicate of a implies the predicate of b.

$a \leq b \;\Leftrightarrow\; \forall\sigma\text{: } (\, \sigma(a) \Rightarrow \sigma(b)\,).$ □

Definition 4 (profile model): A profile model defines the domain of basic profiles, F, and a mapping $C(F, F) \rightarrow boolean$, defining conformance directly. A conformance relationship between a and b exist iff $C(a,b)=true$ or if there exist a set of conformance relationships which lead to this transitively.

$$\big(C(a,b) \;\vee\; \exists c \in F\text{: } (a \leq c \wedge c \leq b)\big) \;\Leftrightarrow\; a \leq b. \;\Box$$

A concrete profile model will define a set of basic profiles plus a set of rules defining conformance relationships between them. For conformance testing, the actual meaning of a profile, $\sigma(a)$ is not used directly, since we use conformance rules to test if one basic profile satisfies another. However, policy implementers and implementers of certain local middleware components, may need to

10 In this section we use the symbols a, b, c for profile expressions assumed to be restricted, for instance to basic (atomic) or sum profiles only. For other profiles we use the symbols x, y, z, u, v and w.

11 We use the notation of placing parameters inside brackets following the profile symbol.

know the exact meaning of a basic profile, for instance that a profile *'HighBW'* means a bandwidth higher than e.g. 100 MB/s. A concrete environment represented by for instance a middleware platform running on a node, may incorporate software fragments which evaluate or generate simple expressions by evaluating the current situation. Thus, implementers of such components would need to know the definition of σ for basic profiles of interest. Policy implementers define and implement configurations representing mappings between a current situation and a particular QoS to be observed by the client. Thus, policy implementers would also need to know the definition of σ. In contrast, a policy trading service only needs to know that for instance *'SuperHighBW'* is a subprofile of *'HighBW'*.

4.3. Profile expression composition

The simplest possible expression (except the empty one) is the reference of a single basic profile. However, if we were to cover all possible situations by single references to a priori defined profiles, basic profile models would need to be complex and domain specific, since an expression typically needs to describe many different dimensions, and possibly the properties of more than one component of the system. If models could be made as simple and generic as possible, this could allow higher degrees of interoperability across different components and domains. We therefore allow profiles to be stated as combinations of basic profiles.

We introduce two operators, '+' (sum) and '⊕' (component-sum) to form expressions combining profiles, and we deduce a set of rules defining the semantics of expressions (conformance rules). Hence, we develop an algebra of profiles. These ideas were first introduced in [Hanssen00] and they are generalised and formalised here.

4.3.1. Sum operator

The '+' operator is used to combine requirements (or offers) to be applied to the same environment. To satisfy a sum $x+y$ both x and y must be satisfied.

Definition 5 (sum): For any environment σ, for the predicate of a sum to be true, the predicate of both operands must be true:

$\sigma(x+y) \Leftrightarrow \sigma(x) \wedge \sigma(y)$. □

Note that our model does *not* assume that x and y are orthogonal to each other. It is also obvious from the rules of logic, that $x + x = x$.

Theorem 1 (associativity and commutativity): Sums follows the associative law: $x+(y+z) = (x+y)+z = x+y+z$ and the commutative law: $x+y = y+x$. □

This can be proved by writing profiles as predicates using definition 1 and the associative and commutative laws for logic.

$$\sigma(x+(y+z)) \Leftrightarrow \sigma(x) \wedge \sigma(y+z) \Leftrightarrow \sigma(x) \wedge \sigma(y) \wedge \sigma(z)$$
$$\sigma((x+y)+z) \Leftrightarrow \sigma(x+y) \wedge \sigma(z) \Leftrightarrow \sigma(x) \wedge \sigma(y) \wedge \sigma(z)$$

Theorem 2 (satisfaction by sum): A basic profile is satisfied by a sum, if and only if it is satisfied by at least one of the operands of the sum.

$(x+y) \leq a \Leftrightarrow x \leq a \vee y \leq a$ □

Proof: From definition 3 and definition 5 we can see that the property holds where x and y are assumed to be basic profiles only:

$$(a+b)\le c \;\Leftrightarrow\; \Big(\big(\sigma(a)\wedge\sigma(b)\big)\Rightarrow\sigma(c)\Big) \;\Leftrightarrow\; \neg\big(\sigma(a)\wedge\sigma(b)\big)\;\vee\;\sigma(c)$$
$$\Leftrightarrow\; \big(\neg\sigma(a)\vee\sigma(c)\vee\neg\sigma(b)\vee\sigma(c)\big) \;\Leftrightarrow\; \Big(\big(\sigma(a)\Rightarrow\sigma(c)\big)\vee\big(\sigma(b)\Rightarrow\sigma(c)\big)\Big) \;\Leftrightarrow\; a\le c\;\vee\;b\le c$$

Let us now assume that the theorem is true also where x and y can be sums of length m and n operands (induction hypothesis). If we substitute x with $(x+a)$ or y with $(y+a)$, where a is a basic profile, we observe that the theorem is still true for sums of length $m+1$ and $n+1$; hence, it is true where x and/or y are sums of any length:

$((x+a)+b)\le c \;\Leftrightarrow\; (x+a)\le c \;\vee\; b\le c \;\Leftrightarrow\; (x\le c \vee a\le c)\vee b\le c$

Theorem 3 (satisfaction of sum): A profile x satisfies a sum if and only if it satisfies both sides of the sum. Here, x, y and z may be any sum or basic profile.

$x\le(y+z) \;\Leftrightarrow\; x\le y \;\wedge\; x\le z$ □

We can prove this by first proving from definition 1 and definition 3 that the property holds for basic profiles.

$$a\le(b+c) \;\Leftrightarrow\; \Big(\sigma(a)\Rightarrow\big(\sigma(b)\wedge\sigma(c)\big)\Big) \;\Leftrightarrow\; \neg\sigma(a)\vee\big(\sigma(b)\wedge\sigma(c)\big)$$
$$\Leftrightarrow\; \big(\neg\sigma(a)\vee\sigma(b)\big)\wedge\big(\neg\sigma(a)\vee\sigma(c)\big) \;\Leftrightarrow\; \big(\sigma(a)\Rightarrow\sigma(b)\big)\wedge\big(\sigma(a)\Rightarrow\sigma(c)\big) \;\Leftrightarrow\; a\le b\wedge a\le c$$

To show that x can be any sum, let us now assume that the theorem is true for sum x which has a length of n operands (induction hypothesis). We then show that it is true for a sum of length $n+1$, i.e. where x is substituted with $a+x$. By using theorem 2 and the induction hypothesis we get:

$$a\le(b+c) \;\Leftrightarrow\; a\le(b+c)\wedge x\le(b+c)$$
$$\Leftrightarrow\; (a\le b\wedge a\le c)\vee(x\le b\wedge x\le c) \;\Leftrightarrow\; (a\le b\vee x\le b)\wedge(a\le c\vee x\le b)\wedge(a\le b\vee x\le c)\wedge(a\le c\vee x\le c)$$
$$\Leftrightarrow\; (a\le b\vee x\le b)\wedge(a\le b\vee x\le b) \;\Leftrightarrow\; (a+x)\le b \;\wedge\; (a+x)\le c$$

The sub-expressions $(a\le c \vee x\le b)$ and $(a\le b \vee x\le c)$ are always true due to the induction hypothesis and can then be eliminated. To show that y and z can be any sum, let us assume that the theorem is true for any sums y and z. We observe that it is also valid where y or z is substituted with $y+a$ (or $z+a$).

$(x\le((y+a)+z) \;\Leftrightarrow\; x\le(y+a)\wedge x\le z \;\Leftrightarrow\; (x\le y\wedge x\le a)\wedge\; x\le z$

Theorem 4 (comparison of sums): For any pair of sums x and y of basic profiles, a sum x satisfies the sum y iff there for all components of x exist a component of y that satisfies it.

$$\sum_{i=1}^{n}a_i \le \sum_{j=1}^{m}b_j \;\Leftrightarrow\; \forall a_j\in\{b_1\ldots b_m\}:\Big(\exists a_i\in\{a_1\ldots a_n\}:\, a_i\le b_j\Big).\ \square$$

This follows from theorem 2 and theorem 3. To prove this we first show that it is true for the base case where each sum contains only one element (i.e. they are simply basic profiles):

$a\le b \;\Leftrightarrow\; a\le b$

We assume that the theorem is true where x is a sum of length n (induction hypothesis). We then observe that this is also true where x is of length $n+1$, i.e. it is substituted with $x+a$ (where a is a basic profile):

$x+a \leq y \;\Leftrightarrow\; \exists u \in \{x, a\} : u \leq y \;\Leftrightarrow\; x \leq y \vee a \leq y$

which obviously follows from theorem 2. This is valid where y is a basic profile. It will also be valid if we assume that y is any sum profile.

Furthermore, we assume that the theorem is also true where y can be a sum of length n (induction hypothesis). If y is substituted with $y+a$ we get:

$x \leq y+a \;\Leftrightarrow\; \forall v \in \{y, a\} : x \leq v \;\Leftrightarrow\; x \leq y \wedge x \leq a$

which obviously follows from theorem 3. This is valid where x is a basic profile. It will also be valid if we assume that x is any sum profile.

4.3.2. Component-sum operator

Basic profiles or sums (of basic profiles) describe properties of single environments. However, we need to describe open component systems where each component represents a separate environment with separate properties. For instance, we may need to characterise the client side and the server side of remote bindings separately.

The '$\oplus$' (component-sum) operator is used to state expressions regarding separate environments. These environments represent separate contexts. The main idea is that to satisfy a component-sum $x \oplus y$ both x and y must be satisfied, but unlike sums, x and y cannot be satisfied by the same profile. The satisfying expression must be a component-sum with separate operands satisfying each x and y. To define the semantics of this operator, we start with the definition of composite environments.

Definition 6 (composite environments)*:* A composite environment σ is a collection of components $\sigma_1, \sigma_2, \ldots \sigma_n$ such that any given sum[12] profile a which is true with σ, is true with at least one of σ_i. No satisfaction relationship exist between profiles if they are applied to different component environments.

$\sigma(a) \;\Leftrightarrow\; \sigma_1(a) \vee \sigma_2(a) \vee \ldots \vee \sigma_n(a)$
$\forall\, i,j \in \{1..n\} \;:\; (\, \sigma_i(a) \Rightarrow \sigma_j(b)\,) \;\Rightarrow\; i=j \quad \square$

This reflects that component environments are to be treated as separate; a statement about one environment cannot be satisfied by a statement about another. For instance, a requirement for processing on the server side cannot be satisfied by processing capacity on the client side.

Definition 7 (component-sum): The '$\oplus$' (component-sum) operator denotes that each operand is applied to different components of the environment. As for sums, for the predicate of a component-sum to be true, both component predicates must be true, but in separate component environments. Observe that order of operands is not significant.

$\sigma(a \oplus b) \Leftrightarrow \exists \sigma_i, \sigma_j \in \{\sigma_1 \ldots \sigma_n\}\colon\; \sigma_i(a) \wedge \sigma_j(b)$
where a and b are sum profiles and σ *is a composition of* $\sigma_1 \ldots \sigma_n$. $\square$

[12]By a 'sum profile' we mean a sum of basic profiles, or a single basic profile, i.e. a sum of only one element, c.f. the identity law (theorem 9).

Theorem 5 (associativity and commutativity): component-sums follows the associative law: $x \oplus (y \oplus z) = (x \oplus y) \oplus z \;=\; x \oplus y \oplus z$, and the commutative law: $x \oplus y = y \oplus x$. ☐

The proof is similar to the proof of associativity and commutativity for sums (theorem 1).

Theorem 6 (satisfaction of component-sum): A profile x satisfies a component-sum y iff x is a component-sum and each operand of y is satisfied by a unique operand of x:

Let *Perm[1..n]* be the set of all possible permutations of the numbers *1..n*.

$x \le (b \oplus c) \;\Leftrightarrow\; \exists s \in Perm[1..n]: (x_{s(1)} \le b \;\wedge\; x_{s(2)} \le c) \quad where \quad x = (a_1 \oplus \ldots \oplus a_n)$. ☐

To prove this, we first show that $(b \oplus c)$ can not be satisfied by a simple sum profile, by using definition 6 and definition 7 to rewrite the proposition to a conjunction where the second and third part are always false due to the second part of definition 6.

$$a \le \left(b \oplus c\right) \;\Leftrightarrow \left(\sigma_1(a) \vee \sigma_2(a) \Rightarrow \sigma_1(b)\right) \wedge \left(\sigma_1(a) \vee \sigma_2(a) \Rightarrow \sigma_2(c)\right)$$
$$\Leftrightarrow \left(\sigma_1(a) \Rightarrow \sigma_1(b)\right) \wedge \left(\sigma_2(a) \Rightarrow \sigma_1(b)\right) \wedge \left(\sigma_1(a) \Rightarrow \sigma_2(c)\right) \wedge \left(\sigma_2(a) \Rightarrow \sigma_2(c)\right) \;\Leftrightarrow\; \mathit{false}$$

Now, replace a with a component-sum with n operands $x = (a_1 \oplus \ldots \oplus a_n)$. If we can show that

(1) If there exist two component-sum operands of x: a_i and a_j such that $(a_i \oplus a_j) \le (b \oplus c)$,
then $x \le (b \oplus c)$

(2) $(c \oplus d) \le (a \oplus b) \;\Leftrightarrow\; (c \le a \;\wedge\; d \le b) \;\vee\; (c \le a \;\wedge\; d \le b)$

Then it is straightforward to prove the theorem. (1) follows from the fact that if x satisfies a profile expression y, $x \oplus a$ also does. Using definition 7 this is equivalent to $(\sigma_1(x) \wedge \sigma_2(a)) \Rightarrow \sigma_1(y)$ which is obviously true if $\sigma_1(x) \Rightarrow \sigma_1(y)$. (2) follows from definition 6 and definition 7:

$$\sigma\left(c \oplus d\right) \Rightarrow \sigma\left(a \oplus b\right) \;\Leftrightarrow\; \left(\sigma_1(c) \wedge \sigma_2(d) \Rightarrow \sigma_A(a) \wedge \sigma_B(b)\right)$$
$$\Leftrightarrow \left(\sigma_1(c) \wedge \sigma_2(d) \Rightarrow \sigma_A(a)\right) \;\wedge\; \left(\sigma_1(c) \wedge \sigma_2(d) \Rightarrow \sigma_B(b)\right)$$
$$\Leftrightarrow \left(\left(\sigma_1(c) \Rightarrow \sigma_A(a)\right) \vee \left(\sigma_2(d) \Rightarrow \sigma_A(a)\right)\right) \wedge \left(\left(\sigma_1(c) \Rightarrow \sigma_B(b)\right) \vee \left(\sigma_2(d) \Rightarrow \sigma_B(b)\right)\right)$$

There are two possibilities for matching environment functions, either by setting index numbers: *A=1* and *B=2,* or *A=2* and *B=1* respectively. In each case one of the implications in each of the two disjunctions will be always be false due to the second part of definition 6. Thus the expression can be rewritten as follows:

$$\left(\left(\sigma_1(c) \Rightarrow \sigma_1(a)\right) \wedge \left(\sigma_2(d) \Rightarrow \sigma_2(b)\right)\right) \vee \left(\left(\sigma_2(d) \Rightarrow \sigma_2(a)\right) \wedge \left(\sigma_1(c) \Rightarrow \sigma_1(b)\right)\right)$$
$$\Leftrightarrow \left(c \le a \wedge d \le b\right) \vee \left(d \le a \wedge c \le b\right)$$

Theorem 7 (satisfaction by component-sum): A basic profile (or a simple sum) is satisfied by a component-sum, iff it is satisfied by at least one of the operands of the component sum.

$(a \oplus b) \le c \;\Leftrightarrow\; a \le c \;\vee\; b \le c$. ☐

The proof is an application of the definitions and elimination of implications with different environments:

$$\left(a \oplus b\right) \le c \;\Leftrightarrow\; \left(\sigma_1(a) \wedge \sigma_2(b)\right) \Rightarrow \sigma(c)$$

$$\Leftrightarrow\; \sigma_1(a) \Rightarrow \sigma(c) \;\vee\; \sigma_2(b) \Rightarrow \sigma(c) \;\Leftrightarrow\; \sigma_1(a) \Rightarrow \left(\sigma_1(c) \vee \sigma_2(c)\right) \;\vee\; \sigma_2(b) \Rightarrow \left(\sigma_1(c) \vee \sigma_2(c)\right)$$

$$\Leftrightarrow\; \sigma_1(a) \Rightarrow \sigma_1(c) \;\vee\; \sigma_2(b) \Rightarrow \sigma_2(c) \;\Leftrightarrow\; a \le c \vee b \le c$$

Theorem 8 (rule for comparing component-sums): A component-sum x satisfies a component-sum y iff every operand of y is satisfied by a unique operand of x. Formally (we use the symbol 'Φ' to denote a component-sum):

$$\underset{i=1}{\overset{n}{\Phi}} a_i \le \underset{j=1}{\overset{m}{\Phi}} b_j \;\Leftrightarrow\; \exists s \in Perm[1..n] : \left(\forall b_j \in \left\{b_1 \;...\; b_m\right\} : \left(a_{s(j)} \le b_j\right)\right)$$

$$where \;\; \underset{i=1}{\overset{n}{\Phi}} a_i = a_1 \oplus a_2 \oplus \dots \oplus a_n \quad \square$$

This follows from theorem 6 and theorem 7. The proof is similar to the proof of theorem 4: We first show that it is true for the base case where each component-sum contains only one element (i.e. they are basic profiles or simple sums):

$a \le b \;\Leftrightarrow\; a \le b$

We assume that the theorem is true where x can be a component-sum of length n (induction hypothesis). We then observe that this is also true for component-sums of length $n+1$, i.e. if x is substituted with $u = x \oplus a$ (where a is a basic profile). Since u here has two elements and b only one, this is equivalent to saying that either the first or second element of u should satisfy b.

$x \oplus a \le b \quad \Leftrightarrow \quad x \le b \;\vee a \le b$

This obviously follows from theorem 7. Furthermore, we assume that the theorem is also true where y can be a component-sum of length m (induction hypothesis) and see if this still holds where y is substituted with $v = y \oplus a$. Here, x must be a component-sum of n elements and there must exist a permutation *s(i)* of these elements, such that both y and a are satisfied by a separate element of v.

$x \le y \oplus a \quad \Leftrightarrow \quad x_{s(1)} \le y \;\wedge\; x_{s(2)} \le a \quad where\ x = (a_1 \oplus \dots \oplus \mathrm{a}_n)$

Which obviously follows from theorem 6.

4.3.3. General expressions and the normal form

Until now we have defined the semantics of expressions which are either basic profiles, sums of basic profiles or component-sums of sums. In the following define the semantics of expressions which may be compositions of sums and component-sums. Essentially, any profile expression can be represented in a normal form. We can then use theorem 4 and theorem 8 to test conformance between any pair of expressions.

Definition 8 (null profile): There exist a special basic profile named '*null*' such that for all σ:

$\sigma(null) = true \quad \square$

Theorem 9 (identity law):

$x + null = x$
$x \oplus null = x \quad \square$

Proof: $\sigma(x+null) \Leftrightarrow \sigma(x) \wedge \sigma(null) \Leftrightarrow \sigma(x) \wedge true \Leftrightarrow \sigma(x)$

Definition 9 (composite expressions): Expressions may be constructed by using basic profiles, the sum and component-sum operators according to operator precedence grammar where the '+' operator has precedence over the '⊕' operator:

$E ::= E + E \mid E \oplus E \mid (E) \mid \textit{basic-profile} \quad \square$

Definition 10 (distributive law): The + operator distributes over the '⊕' operator:

$x + (y \oplus z) = (x+y) \oplus (x+z) \quad \square$

A consequence of this is that for expressions of the form $x+(y\oplus z)$ the profile x applies to both environments of y and z.

The rules we have developed above define the semantics of profile-expressions which are either basic profiles, sums of basic profiles or component-sums of sums of basic profiles. Sums containing component-sums can here be regarded as short-cuts for component-sums containing (shared) requirements. Now we have a complete semantics for general profile expressions.

Definition 11 (normal form): A profile expression is in normal form if and only if it is a component-sum $a_1 \oplus \ldots \oplus a_m$ of sums $a_i = b_{i_1} + \ldots + b_{i_n}$ of basic profiles. $\square$

Using the distribution law and the associative law, we can rewrite any expressions containing component-sums, to the normal form, i.e. component-sums of sums of basic profiles. For instance:

$x + (y \oplus (z + (u \oplus v))) = (x+y) \oplus (x + z + u) \oplus (x + z + v)$

Note that basic profiles or simple sums (of basic profiles) or component-sums of simple basic profiles are also in the normal form. In such cases, some component-sums or sums have only one element. This follows from the identity law (theorem 9). We can now conclude that any pair of profile-expressions can be algorithmically tested for conformance, using the rule of theorem 4 and theorem 8 and the set of rules defining conformance between basic profile defined in the concrete model of use.

4.3.4. Conformance testing algorithm

From the conformance rules defined above, we can develop an algorithm to test any pair of profile expressions for conformance. We assume that profile expressions are first transformed into *normal form*. Expressions can then be evaluated against each other, using a mix of (1) a component-sum test, (2) a sum test and the (3) a basic profile test evaluating rules of a basic profile model. In the following we assume that x and y are profile expressions and that the goal is to determine if $x \leq y$.

Component-sum test

If all expressions to be compared are in normal form, we can start by using a test based on theorem 8 for comparing component-sums. A simple and naive algorithm can be formulated like this (the outermost call to the recursive function *isSubR* starts with i=1) :

```
boolean isSubR(Compsum x, Compsum y, int i)
{
    for (each p in x) {
        if (p ≤ y[i]) {
            remove p from x;
            if (isSubR(x, y, i+1))
               return true;
            re-insert p in x; // Backtrack
        }
    }
    return false;
}
```

The worst case complexity of this simple algorithm is *O(N!)* where *N* is the size of *x* or *y* (depending on which is smallest). Therefore we should look for a better algorithms if the sizes of expressions are not expected to be small. For instance, dynamic programming techniques could be used to eliminate repeated recursions on the same sub-expressions, possibly reducing the complexity significantly. Instead of investigating further how to improve our naive algorithm, we observe that the problem of testing component-sums for conformance corresponds (under certain restrictions) to the problem of determining the existence of a maximum matching in a bipartite graph. Each expression (*x* and *y*) corresponds to a partition, each component of a component-sum of a node in this graph and conformance relationships (from components of *y* to components of *x*), to edges. If a maximum matching involves all nodes in *y*, this means that expression x satisfies expression *y*. It is known from literature that this problem can be solved with algorithms of complexity $O(N \cdot E)$ where *N* is the number of nodes and *E* is the number of edges. However, in our case, to find the edges, we first need to use the conformance rules to determine conformance for up to every possible pair of nodes which is a $O(N^2)$ problem. This means that it is possible to do the matching with a worst case performance of $O(N^3)$.

Sum test

This component-sum test makes use of the test for sums of basic profiles which follows from theorem 4. A simple test with a worst case complexity of $O(N^2)$ can (in pseudocode) be formulated like this:

```
boolean isSubR(Sum x, Sum y)
{
    for (each p in y) {
        for (each q in x) {
           if (p≤q)
              return true;
    }
}
return false;
```

General algorithm

If expressions are in the normal form, a sum cannot contain component-sum components, i.e. *x* and *y* can be either component-sums (of sums or basic profiles), sums of basic profiles of just basic profiles. Furthermore, theorem 8 and the corresponding algorithm above allows component-sum components to be either sums or basic profiles. Therefore we can formulate a testing algorithm for any pair of profile expressions in the normal form. This would be a recursive function, using one of three different tests (basic profile, sum or component-sum test respectively) depending on the type of *x* and *y*. If *x* and *y* are of different types, this is resolved as follows.

- A basic profile and a sum: Use the *sum test* where one of the sums contains only one element.
- A basic profile and a component-sum: Use the *component-sum test* where one of the component-sums contains only one element. Note that a basic profile cannot be a subprofile of a component-sum.
- A sum and a component-sum: Use the component-sum test where one of the component-sums contains only one element; the sum. Note that a simple sum cannot be a subprofile of a component-sum with more than one element. The normal form ensures that a sum will always be of basic profiles.

It follows from theorem 9 that sums or component-sums with only one element can exist.

4.4. Definition of profile models

A *profile model* would be defined for an application or application domain. It can be defined as a rule-base, i.e a set of rules each which states a predicate for when there is conformance between two given basic profiles. A full formal analysis of how we can use rule-bases for this purpose, is outside the scope of this thesis. Here, we describe a relatively simple approach to *model definition* and our experimental prototype which demonstrates it. We show how model definition and rule derivation works using an example.

4.4.1. Foundations

A profile model defines a *partial order*, i.e. a set of binary, reflexive and antisymmetric relations between points in a profile value space. Our profile model definition language is founded on the definitions in section 4.2. In addition we need to define how conformance rules form *axioms* of a model, and how transitive conformance relationships can be inferred from such axioms.

Definition 12 (profile type): A profile type is a pair consisting of an identifier[13], a and a tuple domain $D = [T_1, .. T_n]$ and where $T_i \in$ Integer. A profile type is identified by a. ☐

Definition 13 (conformance rule): A conformance rule is a predicate $p\ (t_1, t_2)$ defining a conformance relationship between two basic profile types (a_1, D_1) and (a_2, D_2). We use square brackets to apply parameters to a profile type instance.

$\forall t_1 \in D_1, t_2 \in D_2\text{:} \quad (\ p\,(t_1, t_2) \quad \Rightarrow \quad x_1\,[t_1] \leq x_2\,[t_2]\ ).$ ☐

The implication is the minimum requirement for the predicate. It can be viewed as a safety requirement in the sense that the rule set should at least not produce false positives. Note also that in first order logic, the conformance operator '≤' should be understood as a predicate taking two basic profile types (a_1 and a_2) as arguments.

Computing derived rule set

The set of axioms should be the minimum needed to define all conformance relationships in the model. Some conformance relationships can be inferred directly from some axiom, others can be inferred from a combination of axioms. It is possible to *derive* an additional set of rules from axioms and it is convenient to do this statically since the axioms do not change at run-time. By

[13]In this section we use the symbols *a, b* and *c* for basic profile types, *x, y, z, u, v* and *t* for parameters, *P*, *Q* and *R* for rules, and *p, q* and *r* for predicates.

precomputing inference which involves more than one axiom rule, conformance checking at run-time becomes equally simple and efficient for any pair of basic profiles. We may view a profile model as a *directed graph* where nodes correspond to profile types, and where edges correspond to predicates of axioms. The derived rules correspond to paths in the graph, and the set of all rules which can be derived corresponds to a *transitive closure* of the graph.

We compute derived rules from axioms by using the principle of transitivity. Given two rules which share a common profile identifier *b*:

$\forall x,y\colon\ p(x, y) \Rightarrow a[x] \le b[y]$

$\forall u,v\colon\ q(u, v) \Rightarrow b[u] \le c[v]$

we want to derive a rule like this:

$\forall x,v\colon\ p(x, y) \wedge q(u, v) \Rightarrow a[x] \le c[v]$.

This derivation is possible if y and u are empty. If not, the new predicate expression would refer to unbound variables y and u. Therefore, simply creating a conjunction would not be a general solution. We need to eliminate y and u. If we assume that parameters y and u have the same domain and have the same value in all instances, such that they both can be re-labelled y, we can try to find a predicate r such that:

$$\begin{aligned}\forall x,y,z :\ & (\ r(x, z) \Rightarrow a[x] \le b[y] \wedge b[y] \le c[z]\ ) \wedge \\ & (\ p(x, y) \wedge q(y, z) \Rightarrow r(x, z)\ )\end{aligned}$$

We can then derive a new rule

$\forall x,z\colon\ r(x, z) \Rightarrow a[x] \le c[z]$.

Alternative paths and disjunctions

There may be more than one rule for a given pair of profile types. In particular, this may be the case after computing the transitive closure, since there may be more than one path between two nodes in the graph representing the rule-base. Alternative rules may correspond to *disjunctions* in the sense that conformance means satisfying at least one of them. They should be combined to one rule to facilitate efficient conformance checking at run-time. For instance, from

$p(x,y) \Rightarrow a[x] \le b[y]$ and

$q(x,y) \Rightarrow a[x] \le b[y]$.

we would derive

$p(x,y) \vee q(x,y) \Rightarrow a[x] \le b[y]$.

4.4.2. Rule definition language

In this subsection we define a language for defining profile models as axioms on the form given by definition 13. As a proof of concept and a tool for further exploration, we have made a compiler which takes a set of such axioms and generates conformance testing code for basic profiles. This code is used by the conformance-tester described in section 4.3.4.

A profile model is specified as a set of *axioms*, each stating a conformance rule. There are two kinds of rules: (1) simple conformance and (2) parametric conformance. Semantically, a simple conformance rule can be seen as a special case of definition 13 where t_1 and t_2 are empty and p= *true*.

Simple conformance

Simple conformance rules are on the form (EBNF):

```
conformance ::= profile-name  [ "<" profile-name ] + ";"
```

One can state conformance between pairs of profile-names, or one can chain several statements. For instance, the statement:

```
a < b < c;
```

is a short form for the following two axioms:

```
a < b;
b < c;
```

Parametric conformance rules

In general, profiles may have parameters, and one may specify rules for how the conformance relationship depends on the value of the parameters.

```
conformance ::= bprofile "<" bprofile  ","  "if"  boolexpr ";"
bprofile    ::= IDENT "[" parameterlist "]"
```

The boolean expressions (*boolexpr*) would specify comparisons of numeric values and/or identifiers. Identifiers referenced in expressions *must* be found in the parameter lists of the profile declaration part. For example:

```
Res[x1, y1] < Res[x2, y2], if x1 >= x2 AND y2 >= 23;
```

Our experimental prototype supports the "<", ">", "<=", ">=" and "=" comparison operators. Only one value type is allowed: integer numbers. The boolean operators: *AND*, *OR* and *NOT* can be used to compose more complex expressions.

4.4.3. Profile model compiler

Our experimental profile model compiler is constructed using Java tools like *jflex*, *cup* and *classgen* [Cup] and does its work in the following distinct phases:

1. Transform source text into an abstract syntax tree (lexical and syntax analysis), which can be traversed and transformed using the visitor pattern [Gamma95].

2. Semantic analysis: consistency checking, computing of parameter indices, transforming the AST to list of ASTs (representing rules) and transforming boolean expressions to the *conjunctive normal form*.

3. Compute derived rules representing the transitive closure of the initial rule set.

4. If there are alternative rules for any pair of profile-types, combine by disjunction as suggested above.

5. Generate conformance testing code representing the derived rule set.

Semantic analyser

The semantic analyser check that all variable names used in the boolean expression exists in the *'bprofile'* part (see section 4.4.2), and it assigns an index to each of them which corresponds to where it was declared. It would then normalise the predicate expressions by applying the following transformations:

$\neg(x < y) \rightarrow x \geq y$
$\neg(x > y) \rightarrow x \leq y$
$\neg(x \leq y) \rightarrow x > y$
$\neg(x \geq y) \rightarrow x < y$
$\neg(p \vee q) \rightarrow \neg p \wedge \neg q$
$\neg(\neg(p)) \rightarrow p$

Now the AST[14] for the predicate expressions will be in a *conjunctive normal form*, having the following properties: (1) The operands of an AND node can be AND, OR, or COMPARE nodes, (2) the operands of an OR node can only be COMPARE nodes.

Computing the transitive closure

We use an adapted version of a well known algorithm for computing the transitive closure of a directed graph[15]. The transitive closure algorithm inserts new edges where it finds a path of two edges between pairs of nodes. In our case, we operate on rules instead of simple edges; we try to create derived rules. Figure 4.3 illustrates this.

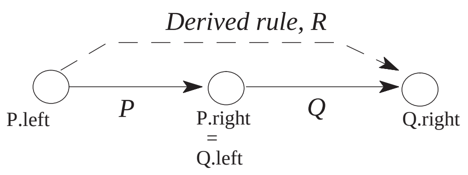

Figure 4.3. Rule derivation step

Given a pair of rules *P, Q* and where *p* and *q* denotes the predicates of *P* and *Q* respectively, the method for deriving a new rule *R* is as follows:

1. If *P* is a simple rule (*p* just returns *true*), just return *r* = *q*. Similarly return *r* = *p* if *Q* is a simple rule.
2. Let *P.left* denote the left *'bprofile'* node of a rule *P*, Let *P.right* denote the right side etc. If the identifier of *P.right* equals the identifier of *Q.left*, continue. If not, return no match.
3. Combine two expressions *p* and *q* by first constructing a conjunction node: *AND (p, q)*. Then we flatten AND-trees to a list, For instance, *AND(AND(p, q), r)* would be transformed to *ANDLIST(p, q, r)*.

[14]The abstract syntax tree will have nodes for NOT, AND, OR and COMPARE (which corresponds to the grammar production *expr* → *expr RELOP expr*, where RELOP is a comparison operator).

[15]It is straightforward to use a DFS based algorithm, for instance the one in [Sedgewick] p. 178.

4. For any possible pair *(p, q)* of conjuncts of this list, if *p* and *q* originate from different rules, we try to combine them using the subexpression matching rules described below. If matching is successful, the result is added to the resulting expression (by conjunction).

5. The remaining conjuncts (those which were not matched in step 4) are added to the resulting expression. Remaining subexpressions with unbound operands are removed since they are not transitively related to any other subexpression, and thus have no meaning in the derived rule.

6. Construct a new rule node *R* setting *R.left* = *P.left*, *R.right* = *Q.right* and using the predicate rule resulting from step 4. Use the semantic analyser to check that all variables in the predicate are defined in either *R.left* or *R.right*. If true, return *R*, otherwise return no match.

Subxpression combination rules

Assume that we have a conjunction ($p \wedge q$) of subexpressions each of which is either an arithmetic comparison expression (e.g. $x < y$), or a disjunction of arithmetic comparison expressions.

1. If *p* and *q* are comparison expressions, and the right operand of *p* is identical with the left operand of *q,* there is a transitive relationship between the two expressions. It is necessary to rewrite as follows when deriving a rule where the operand *y* is unbound:

   $x < y \wedge y < z \rightarrow x < z$
   $x \leq y \wedge y \leq c \rightarrow x \leq z$
   $x < y \wedge y \leq z \rightarrow x < z$
   $x \leq y \wedge y < z \rightarrow x < z$
   $x > y \wedge y \geq z \rightarrow x > z$
   $x = y \wedge y = z \rightarrow x = z$
   $x = y \wedge y < z \rightarrow x < z$
   $x < y \wedge y = z \rightarrow x < z$
   $x = y \wedge y \leq z \rightarrow x \leq z$
   $x \leq y \wedge y = z \rightarrow x \leq z$

etc.

2. If *p* is a comparison expression and *q* is a disjunction, we can use the distributive law to transform it such that we can apply step 1 above, for conjunctions of compare nodes.

   $p \wedge (q_1 \vee q_2 \vee \ldots q_n) \rightarrow (p \wedge q_1) \vee (p \wedge q_2) \ldots \vee (p \wedge q_n)$.

4.4.4. Example

The following example capture delay and display resolution. We illustrate the simple form of parametric profile (rule 2 below), meaning that a smaller number satisfies a higher number. We also illustrate various ways to state resolution in one, two or three dimensions (typically for display resolutions) where obviously higher numbers satisfy smaller numbers. We include examples of profile declarations for one dimension (rule 5 and 8), two dimensions (rule 7) and three dimensions (rule 9), as well as rules for comparing profiles of different dimensions (rule 6 and 10).

```
LowDelay < ModerateDelay < AnyDelay;                          (1)
Delay[x] < Delay[y], if x <= y;                               (2)
Delay[x] < LowDelay, if x <= 100;                             (3)
Delay[x] < ModerateDelay, if <= 200;                          (4)

XRes[x]  < HiRes, if x < 1000;                                (5)
Res[x1, y1] < XRes[x2], if x1 >= x2;                          (6)
Res[x1, y1] < Res[x2, y2], if x1 >= x2 AND y1 >= y2;          (7)

XRes[x] < XRes[y], if x <= y;                                 (8)
3Res[x1, y1, z1] < 3Res[x2, y2, z2],
    if x1 > x2 AND y1 > y2 AND z1 > z2;                       (9)
3Res[x1, y1, z1] < Res[x2, y2], if x1 > x2 AND y1 > y2;      (10)
```

From the above axioms we can for instance infer that `Delay[10]` satisfies `ModerateDelay` and that `Res[2000, 1000]` satisfies `XRes[500]`. In the following we give some examples of how rules are derived from other rules.

Example 1:

From rule 5 and 6 above we can derive

```
Res[x, y] < HiRes, if x < 1000;
```

When explaining how the rules are derived we use a slightly different notation where parameters are relabelled using the profile identifier with an index.

$XRes \le HiRes \quad \Leftarrow (\mathrm{XRes}_0 > 1000)$
$Res \le XRes \quad \Leftarrow (\mathrm{Res}_0 \ge \mathrm{XRes}_0)$

The predicates can be be conjoined:

$(\mathrm{XRes}_0 > 1000) \wedge (\mathrm{XRes}_0 > \mathrm{Res}_0)$

The two comparison nodes are transitively related and we can eliminate $XRes_0$ when deriving a rule:

$Res \le HiRes \quad \Leftarrow (\mathrm{Res}_0 > 1000)$.

Example 2:

From rule 6 and 10 we can derive

```
3Res[x1, y1, z1] < Res[x2, y2], if (x1 > x2);
```

Rule 6 and 10 are:

$Res \le XRes \quad \Leftarrow (\mathrm{Res}_0 \ge \mathrm{XRes}_0)$
$3Res \le Res \quad \Leftarrow (\mathrm{3Res}_0 > \mathrm{Res}_0) \wedge (\mathrm{3Res}_1 > \mathrm{Res}_1)$

The predicates are conjoined:

$(\mathrm{Res}_0 \ge \mathrm{XRes}_0) \wedge (\mathrm{3Res}_0 > \mathrm{Res}_0) \wedge (\mathrm{3Res}_1 > \mathrm{Res}_1)$

We combine the first and the second comparison node by transitivity. The third node still has an unbound variable and is removed. We derive:

$3Res \le XRes \quad \Leftarrow (\mathrm{3Res}_0 > \mathrm{XRes}_0)$

Example 3:

We can derive a rule for *Delay[x]* ≤ *AnyDelay*. From rule 1, 3 and 4, we see that there are two paths. Either via *LowDelay* or *ModerateDelay*:

$Delay \leq AnyDelay \Leftarrow (Delay_0 < 100)$
$Delay \leq AnyDelay \Leftarrow (Delay_0 < 200)$

The second alternative is widest in the sense that it returns true for a larger set of values for *x*, so it should be chosen. If combined by using a disjunction, the first case has no effect, since it is fully covered by the second case. Figure 4.4 below illustrates the relationships as a graph.

$Delay \leq AnyDelay \Leftarrow (Delay_0 < 100) \vee (Delay_0 < 200).$

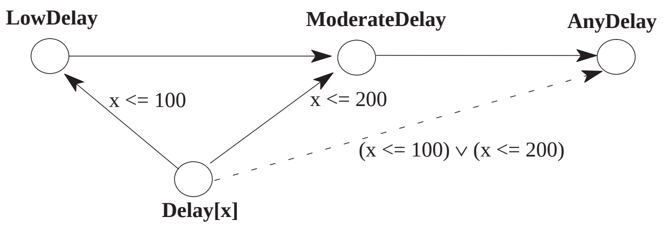

Figure 4.4. Rule derivation

Example 4:

Assume that we remove rule 5 and add some rules stating a resolution requirement in either the *x* or *y* dimension.

```
1Res[x]  < HiRes,   if x > 1000;                          (11)
Res[x,y] < 1Res[n], if x > n OR y > n;                    (12)
```

From these two we can conjoin the predicate expressions like we did for the examples above:

$(1Res_0 > 1000) \wedge ((Res_0 > 1Res_0) \vee (Res_1 > 1Res_0))$

If we use the distributive law to transform the expression to a disjunction of two conjunctions, and if we use the transitivity combination rules for each of these, we get a derived rule:

$Res \leq HiRes \Leftarrow (Res_0 > 1000) \vee (Res_1 > 1000)$

4.5. Evaluation and analysis

In this section we analyse the profile model with respect consistency and completeness in concrete model definitions, to interoperability between autonomous applications, and to how composition can be adressed using the model. But first, we briefly discuss the performance issues of evaluating profile expressions.

4.5.1. Performance

The complexity of evaluating profile expressions against each other to test for conformance is discussed in section 4.3.4. The matching of component-sums is equivalent to a bipartite graph matching problem. Therefore, it is possible to implement this with a worst case performance of $O(M^3)$ where M denotes the number of components of the component-sum in the normal form of the expressions[16]. Our naive but simple implementation has an exponential worst case complexity. The matching of sums is *$O(M^2)$* worst case (where *M* is the number of components of the sum).

It is generally harder to analyse the practical performance of expression matching independently of applications. First, performance is highly influenced by the structure of the expressions to be compared, not only their length. Furthermore, this analysis is based on the fact that M denotes the length of both expressions. Often, the lengths are different. The performance is also influenced on how often there exist conformance relationships between basic profiles used; i.e. the size and the structure of the profile conformance rule-base are also relevant.

We have implemented a prototype policy trader, using the algorithm as described above. This is a tool for experimenting on different application scenarios. The prototype is implemented in Java and contains an expression parser, conformance testing code, as well as a trading service interface. To get an indication of how comparison behaves, we run the comparison algorithm on examples believed to be realistic (see also section 6.3). In the following case we have two service profile expressions and two environments to match. We assume that there is a parametric rule for *'Lat'* (which represents latency) and that we have a rule saying *that 'Storage'* satisfies *'LimitedStorage'*:

SP_1="(Channel+Lat[30])⊕(Server+LimitedStorage+Lat[10])"

SP_2="(Channel+Lat[39])⊕(Server+LimitedStorage+Lat[1])"

```
E1 = "Server+((Channel+Lat[35])⊕(Storage+Lat[10])
        ⊕(LimitedStorage+(Lat[0.5]))"
```

```
E2 = "Server+((Channel+Lat[30])⊕(Storage+Lat[10])
         ⊕(LimitedStorage+Lat[0.8]))"
```

The number of basic profile comparisons lies between 9 and 16 in this example (the number basic components is 5 and 7 in the expressions to compare, respectively). If we use the component-sum operator to add a component to the service profiles, the number of basic comparisons rise to 25, at most. This and other experiments indicate that the number of basic comparisons would typically be lower than *M^2* (where *M* is the number of basic components in the longest expression), though theoretically the worst case complexity would lie between *N^2* and *N^3* depending on the expression structure. This analysis suggests that if the length of profile expressions are kept within manageable bounds, the comparison cost would be reasonably small.

Test			Result	Comparisons
E1	≤	SP1	FALSE	10
E1	≤	SP2	TRUE	16
E2	≤	SP1	TRUE	9
E2	≤	SP2	TRUE	16

Table 4.1. Basic comparisons of expressions

We have been able to produce a worst case behaviour for component-sums. Our experiments with application cases (see chapter 6) indicate however, that the probability of such scenarios is small. The average performance is expected to be significantly better than the worst case performance. We also observe that a critical operation in our algorithms is the comparison of basic profiles,

[16]Algorithms with complexity *$O(EV)$* may at first sight seem to be better, but finding all edges (conformance relationships) is a *$O(N^2)$* problem. So, in sum it is a *$O(N^3)$*.

which is expected to be cheap (see section 4.4). We believe that in most cases, the length of profile expressions would be reasonably low, and probably lower than the number of QoS characteristics in parameter-based negotiation schemes, since our declared conformance approach allows profiles which abstract over detailed characteristics. It is therefore likely that scalability issues would be related to the number of candidate policies to search in the trading process rather than the length of profile expressions.

4.5.2. Model consistency and completeness

When defining concrete models as described in section 4.4, it is possible to mistakenly define models which do not work as intended. One might make models with inconsistencies, or one may fail to completely cover the range of conformance relationships needed for the application in question. Therefore, additional guidelines and tools for checking can help to avoid or discover such problems.

In this section we propose a consistency criterion which let us detect some problems by systematic analysis. We look at some relevant cases, to discover what we can expect to be typical patterns of rule-derivation or sources of problems. We also observe that it can be useful to extend the model definition language with higher order constructs like equivalence.

Completeness

A model is *complete* if all possible conformance relationships may be inferred from the rule-set; i.e. if complete, the implication in definition 13 is also an equivalence. Definition 13 means that models do not need to be complete, but one should strive to define models which are *sufficiently* complete, meaning that the conformance relationships needed by the applications are covered.

Consistency

Consistency means that we would not infer contradicting results from a set of axioms. However, two rules returning different truth values for a given set of parameters are not necessarily inconsistent rules. Instead we base the notion of consistency on which *range* of profile values evaluate to true. To help finding inconsistencies we propose a simple rule which can be applied to the axioms of a model.

Proposition (consistency): The predicate p of any axiom, $a \le b$, would be *implied by* any derived predicates q, for $a \le b$. We say that *p covers q* in the sense that the set of values which satisfy q is a *subset* of the values which satisfy p.

$\forall x: q(x) \Rightarrow p(x)$. □

A derived rule will involve another profile type, c and a rule $c \le b$ such that all profile values which satisfy c also satisfy b. If we assume we have a complete and consistent model, the set of profile values which (when applied to a and b) satisfy $a \le c \ \wedge \ c \le b$, is a subset of the values which satisfy $a \le b$ directly. This follows from the definition of conformance relationship (definition 3) and the definition of transitivity:

$(\sigma(a) \Rightarrow \sigma(c) \ \wedge \ \sigma(c) \Rightarrow \sigma(b)) \ \Rightarrow \ (\sigma(a) \Rightarrow \sigma(b))$

We cannot prove consistency since our models are not complete. However, we can prove that inconsistencies exist. Given the argument above about consistent models, and definition 13, it can be proven that if the above proposition does not hold for a set of rules, the model is inconsistent: It is straightforward to prove that the following is true (*p,q,r,x,y* and *z* are logic expressions):

$(\neg(p \wedge q \Rightarrow r) \wedge x \Rightarrow p \wedge y \Rightarrow q \wedge z \Rightarrow r) \Rightarrow \neg(x \wedge y \Rightarrow z)$.

By definition 13 and transitivity we have

$p \Rightarrow a \le b,\ q \Rightarrow (a \le c \wedge c \le b) \Rightarrow a \le b$.

From this we can show that

$\neg(q \Rightarrow p) \Rightarrow \neg(a \le c \wedge c \le b \Rightarrow a \le b)$.

Examples

In the following we discuss some cases where this rule may help us detect problems. We also consider cyclic rule-sets and two patterns for rules which go in the opposite direction: Symmetry (to address completeness) and equivalence (to realise synonym profiles). We also consider the pattern of parallel paths.

Example 1 (inconsistent rules): Consider the following rules:

```
LowDelay < ModerateDelay;                         (1)
Delay[x] < LowDelay, if x <= 100;                 (2)
Delay[x] < ModerateDelay, if x <= 50;             (3)
```

Since *LowDelay* ≤ *ModerateDelay*, the requirement for a *Delay* profile to satisfy *LowDelay* should be at least as strict as for *ModerateDelay*. There is an inconsistency here in the sense that rule 3 does not cover the rule derived from rule 1 and 2. In this case, the result of combining alternative rules by disjunction is that rule 3 will have no effect. Figure 4.5 illustrates this by representing profile types as nodes and predicates as edges. We show derived rules by using dashed lines.

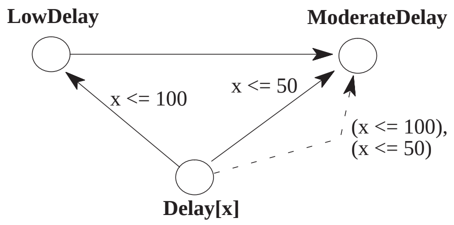

Figure 4.5: Derived rule

Example 2 (cycle): Consider the following rules:

```
LowDelay < ModerateDelay;                         (1)
Delay[x] < LowDelay, if x <= 100;                 (2)
ModerateDelay < Delay[x], if  x > 200;            (3)
```

This results in a *cycle* in the conformance graph. A cycle is not necessarily an error, but may be problematic because of the derived rules it produces.

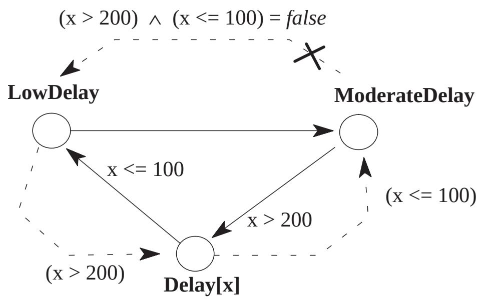

Figure 4.6: Conformance cycle

No new rule will be derived for *ModerateDelay* ≤ *Lowdelay* (conformance is *false*) since *x* is not bound by any of the profile types. In this case, there should not be any conformance in this direction anyway. The derived rules for *Lowdelay* ≤ *Delay* and *Delay* ≤ *ModerateDelay* are not wrong, but imprecise in the sense that they do not cover all cases where we expect conformance. Here, we would expect conformance to be true for exactly the set of values for which conformance is false in the opposite direction.

These observations suggest that cycles may lead to derivations which are not intuitively foreseen, and that some of those are not necessarily as complete as we may wish. They may still be useful, but somewhat complicated to analyse by a programmer. Therefore, we believe that a profile model compiler should warn or inform about cycles, except when introduced by symmetric rules in opposite directions (directly between two nodes), or by defining equivalence. We illustrate the usefulness of this case in example 3 and example 5 below.

Example 3 (symmetry): Consider the following rules:

```
LowDelay < ModerateDelay;                       (1)
Delay[x1] < Delay[x2], if x1 <= x2;             (2)
Delay[x] < LowDelay, if x <= 100;               (3)
LowDelay < Delay[x], if x > 100;                (4)
ModerateDelay < Delay[x], if x > 200;           (5)
Delay[x] < ModerateDelay, if x <= 200;          (6)
```

Here, we want to represent the fact that *LowDelay* satisfies *Delay[x]* if *x* is smaller than 100 (rule 2), while also *Delay[x]* satisfies *LowDelay* if *x* is bigger than 100 (rule 3 and 4). This realises a more complete connection between the two profile types than the previous example. A similar rule is made to connect *Delay* with *ModerateDelay* (rule 5 and 6). These rules are consistent with the rule for two *Delay[x]* profiles in the sense that derived rules for *Delay* ≤ *ModerateDelay* and *Delay* ≤ *Delay* are covered by axioms

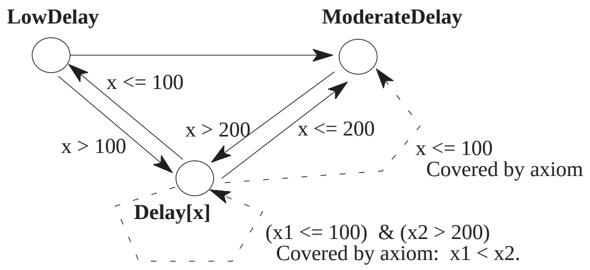

Figure 4.7. Symmetric rules

Example 4 (Conflicting rules): Assume that we make a mistake in example 3 above and define conformance between *ModerateDelay* ≤ *LowDe*lay in the wrong direction. Now, there will be a derived rule for *Delay* ≤ *LowDelay* which is *not* covered by the axiom. In fact, the comparison rules go in the opposite direction. Observe that the derived rule for *Delay* ≤ *Delay* is not covered by the axiom as well. We have two indications that the model has inconsistencies.

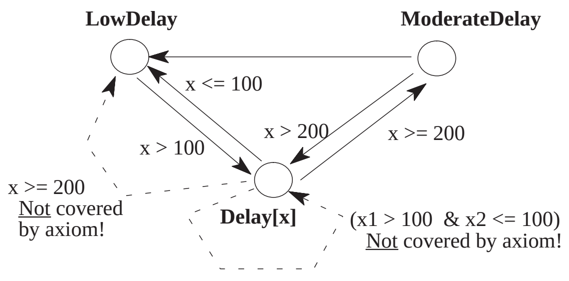

Figure 4.8. Conflicting rules

Example 5 (Equivalence): Consider the following rules:

```
LowDelay < LowLatency < LowDelay;                 (1)
Delay[x] < Latency[y];                            (2)
Latency[x] < Delay[y];                            (3)
Delay[x] < LowDelay, if x <= 100;                 (4)
LowDelay < Delay[x], if x > 100;                  (5)
```

In some cases, we may want to define *equivalence* between pairs of profiles, meaning that there is a conformance relationship in both directions at the same time. We limit the discussion to the cases where there is equivalence for all possible parameter values. In figure 4.9 below, we indicate equivalence by thicker, double arrowed lines. It illustrates how equivalence and rule derivation effectively mirrors the conformance rules to *synonym* profile types.

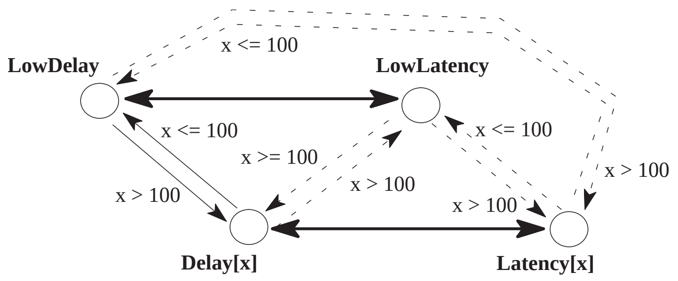

Figure 4.9. Equivalence

Example 6 (parallel paths): Consider the following rules:

```
XRes[x] < HiRes, if x >= 1000;              (1)
YRes[y] < HiRes, if y >= 700;               (2)
Res[x,y] < XRes[x], if x1 >= x2;            (3)
Res[x,y] < YRes[y], if y1 >= y2;            (4)
```

Rules 3 and 4 define the relationship between one-dimensional and two-dimensional profiles describing (display) resolution. But they will also define a relationship between *Res* and *HiRes*; i.e. it represents a composition of two rules into one. If we follow the simple rule of composing alternative rules to disjunctions, we get

```
Res[x,y] < HiRes, if (x >= 1000) or (y >= 700);
```

Figure 4.10 illustrates how rules are composed. We have two parallel derived rules which do not cover each other in any way. In this case they cannot, since they are based on different parameters in the *Res* profile. Intuitively, it looks like it is possible to define conflicting parallel rules, but to be able to detect or prove the existence of such conflicts, we need to add further constraints to the model. This may be a topic for further investigation.

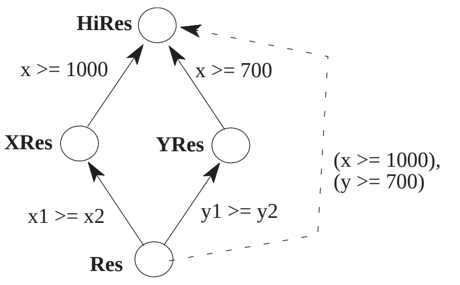

Figure 4.10. Parallel rules

Another possible problem of this pattern may occur if we actually want to compose the two rules (between *XRes*/*YRes* and *HiRes*) by *conjunction* instead of disjunction. Note that we may express conjunctive composition in an expression by using the '+' operator (cf. section 4.3.1). To build conjunctive composition into the model we may be tempted to define a rule explicitly between *Res* and *HiRes*, using conjunction. However the explicit rule would then clearly be inconsistent with the two derived rules (a conjunction does not cover a disjunction or any of the individual conjuncts). In addition, it is somewhat redundant. A possible resolution is to recognise this particular pattern, and simply override any derived rules, or we may add a keyword to the profile definition language to explicitly request such overriding.

Discussion

It is possible to develop a notion of consistency for profile models. We have seen that there is a set of inconsistencies which can be detected and reported by a profile model compiler, to the extent it can detect if one predicate imply another.

Cycles in the conformance graph are legal, but may lead to derived rules which are not easily foreseen by the model designer and which may be imprecise. Therefore cycles are likely to be problematic, except when introduced by (1) *symmetric rules*, i.e. rules which go in the opposite direction and where one rule is a negation of the other, or (2) *equivalence*, i.e. rules which evaluate to true in both directions at the same time. It is useful to restrict equivalence to simple rules (always true). It is possible for a compiler to warn about cycles which do not follow these constraints. It would also be useful to extend the profile definition language with higher order constructs like an equivalence operator and an operator to define symmetric rules automatically.

Example 5 shows that there exists a type of inconsistency which we may want to resolve by letting the axiom override (disable) the derived rules. We observe that the pattern of parallel paths can be a source of some problems and it may be helpful for a profile model programmer, if a compiler could detect and report this case.

4.5.3. Interoperability

Components participating in a binding may be heterogeneous and not necessarily designed for one single purpose. Still, there is a need to express and convey QoS and resource information among the participants. In this context, interoperability is about establishing a common understanding across different subdomains and component applications, of profile expressions to be exchanged during negotiation, and consequently, of the resulting contracts. We may distinguish between two levels of interoperability: (1) Interoperability among components in a single application sharing a single profile model. (2) Interoperability among applications using different profile models.

We assume that participants have agreed on a shared profile model which defines the syntax and semantics of profile expressions to be exchanged. However, the semantics of profile identifiers and parameters are not completely defined, since a profile model is limited to conformance relationships. Each participant needs to interpret profiles in terms of measurable characteristics or platform properties, and they should do it in a consistent way. For instance, the parameter in a profile *Delay[x]* may mean the maximum end-to-end execution time for empty operation invocations measured in milliseconds. It is obviously a problem if the other participant understands it as a mean estimate, or is using microseconds as the unit. It is also important that programmers of policies or platform components have precise information about the meaning of basic profiles, in addition to the profile model itself, to be able to implement the components correctly. The consequence of making the wrong assumptions could be that the wrong policies are selected. This could lead to failure of policies or that they violate the contracts.

Integrating models

Applications may want to interoperate and negotiate, even if they use different profile models locally. This problem is comparable with the problem of integrating schemas for federated databases [ShLa90] or different data-sources described by ontologies [Wache01] (for instance in the context of the semantic web).

We can distinguish between (1) *loose coupling strategies* where mappings between applications to cooperate are defined in an ad. hoc. way and (2) *tight coupling strategies* where one would statically define one or more federated (global) schemas based on local schemas. [Wache01] also classify the approaches to mapping based on ontologies as follows

1. Defining *one shared ontology* (or merging local ontologies into one) for all data-sources to interoperate. This approach limits the diversity of local applications.

2. *Multiple ontologies* which is somewhat similar to a loose coupling strategy, but this makes it more difficult to compare local ontologies.

3. *Hybrid approaches* allowing local ontologies which share a common vocabulary. Local ontologies may be defined in terms of this vocabulary, and this supports easier comparison and mapping. However, with this approach it is harder to integrate existing ontologies which are not based on the common vocabulary.

In the following discussion, we assume that all local models to be integrated follow the core profile model defined in this chapter. This simplifies the problem compared to a more general case. We also assume a tight coupling strategy and that the integration is done by merging relevant parts of profile models (termed local models) into one global model. We may also define new profiles at the global level to generalise over local concepts. We may provide *mappings* such that expressions in terms of one local model can be understood in terms of global model (or another local model). Mappings should preserve autonomy in the sense that existing negotiations within a single application do not need to be affected by the integration. Each application would use their own profile models in platform implementations and policy implementations.

In our case, relationships between local and global profile types can be defined as *conformance rules* (see section 4.4). This means that a policy trading service (chapter 3) would automatically perform the necessary mappings if it knows the complete integrated model (including the mapping rules). Alternatively, one may add run-time interception and translation of expressions which cross application boundaries.

Heterogeneity and conflict types

The problems arising from data heterogeneity are well known within e.g. the federated database community [ShLa90]. In our context we are mostly concerned about semantic heterogeneity [Kim91]. Integration of models will need to resolve conflicts. According to a taxonomy of [Goh97], data heterogeneity leads to three main types of semantic conflicts: Naming conflicts, scaling conflicts and confounding conflicts. Because of the strong limitations of exchanged data types in our model, the schematic and intensional conflicts (except generalization conflicts), are not directly relevant for us. The taxonomy is shown in figure 4.11 below.

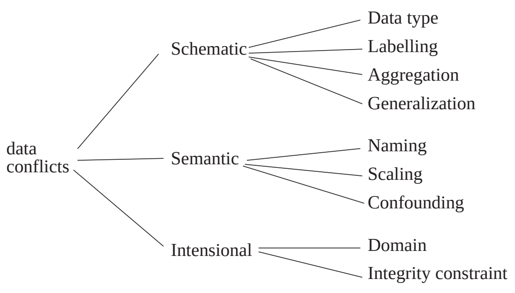

Figure 4.11: Conflict taxonomy

Naming conflicts stem from differing naming of values, typically synonyms and homonyms. In our model, this is also similar to what [Goh97] refers to as *labelling conflicts* since the labels of profile types also denote possible profile identifier values. Synonyms are handled by defining equivalence (cf. section 4.5.2). Homonyms can be resolved for instance by prefixing names, such that they are recognised as distinct.

Scaling conflicts stem from the use of different units for values. This includes using different currencies, or expressing delay in milliseconds in one place and microseconds in another. Our models do not state units and scaling explicitly. It is possible to resolve such conflicts by defining rules which do the necessary conversions. However, this will require that the language in section 4.4 is extended with arithmetic operators. For example, if delay is given in both milliseconds and microseconds:

```
Delay[x] < us_Delay[y], if  x <= y * 1000;
us_Delay[x] < Delay[y], if  x * 1000 <= y;
```

Given the idea of extending with an equivalence operator (section 4.5.1), we may add scaling. From such a rule, proper conformance rules may be derived. For instance:

```
Delay[x] = us_Delay[x * 1000];
```

Confounding conflicts occur when information items seem to have the same meaning but have not, because they are defined in different contexts. For instance it is a conflict if a *Delay* parameter denote a maximum delay in one instance and a mean delay in another. It is not trivial to map between various interpretations for this type of conflict. Note that profiles like e.g. *Delay* may be used in different contexts, and a rule-set for *Delay* can have different contextual interpretations. Recall that the component-sum operator (section 4.3) can be used to express constraints in different contexts. It is possible to use the sum operator to relate a constraint to a context. For instance:

```
( Mean + Delay[40] ) ⊕ ( Maximum + Delay[100] )
```

state requirements for both mean and maximum delay (the *Mean* and *Maximum* profiles represent contexts for delay expressions). Expressions in different contexts are not comparable unless we define an additional rule defining that one context is subsumed by another. For instance, we may also specify a rule *Maximum* ≤ *Mean* meaning that a maximum delay offer will satisfy a mean delay requirement, from which we can infer that *(Maximum* + *Delay[40])* satisfy *(Mean* + *Delay[40])*. Note however that this example reveals a dangerous trap. Consider a parametric profile *Bandwidth[x]* instead, where the higher parameter value satisfies a lower. Here, it will be wrong to use it with the *Maximum* ≤ *Mean* contexts, since they were meant for use with a parametric profile where the smaller value would satisfy a higher. In this case, it would be better to relate the context profiles to the more abstract notion of conformance relationships. For instance to define: *Best* ≤ *Mean* ≤ *Worst*.

Model consistency

In section 4.5.2 above, we discuss consistency with respect to conformance rules and transitivity. The types of conflicts discussed there can also arise from heterogeneity in conformance rules. It can be viewed as a *generalisation conflict* [Goh97] in the sense that profile expressions subsumes each other by conformance relationships. Integrating models may create new paths of transitive conformance, and conflicting derived rules if conflicts between models are not properly resolved. The following example illustrates how inconsistency can arise from wrongly assuming that two profile types are equivalent (we assume that we can define equivalence using the '=' operator).

```
Delay[x] < LowDelay, if x <= 100;              (model 1)
Latency[x] < GoodLatency, if x <= 50;          (model 2)
Delay[x] = Latency[x];                         (mapping 1)
LowDelay = GoodLatency;                        (mapping 2)
```

The error here is that *GoodLatency* is assumed to be equivalent with *LowDelay*. It is not. This problem can be detected using the consistency test from section 4.5.2, i.e. the derived rule for *Delay* ≤ *LowDelay* should not be covered by the axiom (from model 1). This problem is easily resolved by changing mapping rule 2 to a one way conformance:

```
LowDelay < GoodLatency;                        (mapping 2)
```

4.5.4. Composition

We want to support composition of profile expressions. The profile model can capture basic conjunctive composition using the sum operator and composition of separate contexts using the component-sum operator. In this section, we look into the pragmatics of expressing composition using these operators, and discuss certain limitations of our model in addressing composition.

First, it is important to observe that it is not within the scope of profile expressions themselves, to define *how* components are combined and what a composition actually results in, but rather *constraints* on how policies can combine the resources. Our approach of policy trading implies that the *effect* of composition, i.e. the *relationship* between the expectation (service profile) and the obligation (user profile), is encapsulated in the policy. It is the concern of policy implementers how available resources should be used in combination to reach a goal. Given the same environmental properties, making different choices with respect to protocols and other parts of policy implementations, could result in different interactions between the resources. This could again lead to different resulting QoS.

This is somewhat different from CQML [Aagedal01] where the result of composition may be specified per QoS characteristic. Three types of composition are considered: *'parallel-or'*, *'parallel-and'* and *'sequential',* and an application specific model may define functions defining characteristic specific meaning for each of them. As an example, Aagedal considers (as an example), the start-up time for a composition of two components. The total start-up time could either be the quickest one if the policy is to select the best one (parallel-or composition), the latest one if both are needed (parallel-and composition), or a sum of the start-up times if the second one cannot be started before the first is ready (sequential composition). For instance, the sequential composition of startup times would be the sum of the component startup times.

In the following, we discuss how the patterns of composition in [Aagedal01] (*parallel-or*, *parallel-and* and *sequential)* can be captured by our profile model. We also discuss how to address nested composition. But first we briefly discuss some pragmatics in describing entities.

Entity descriptors

Environmental expressions (in normal form) will typically be component sums, where each element is a sum of some entity name, defining the context, plus a set of constraints to be associated with it. We refer to such subexpressions as *'entity descriptors'*. Note that the term *'entity'* is not a syntactic category of our expression language; it is rather a way to describe pragmatics of structuring expressions.

$entity + property_1 + ... + property_n$

Expressions may be composed from a set of entity descriptors. We should normally use the component-sum operator to separate entity descriptors, like this:

$(entity_1 + properties) \oplus ... \oplus (entity_n + properties)$

The entity name may be skipped if there are no ambiguity with respect to what entity the property refer to, for instance when the expressions describe only one entity. Furthermore, expressions can be sums of entity names only, meaning that all entities are (or is required to be) available. For instance, one could say *"network + CPU + memory"*. The existence of entities may be viewed as properties, but observe that the choice of this form must be consistent amongst the negotiating participants[17]. For instance, there will be no conformance relationship between *"A + B + C"* and *"A ⊕ B ⊕ C"*. If other properties are to be associated with any of the entities, they must be separated by using a component-sum like in following expression:

(Network + HighBW) ⊕ CPU ⊕ (Memory + HighCapacity)

In contrast, the expression

Network + HighBW + CPU + Memory + HighCapacity

would not express the same fact. In this example we can no longer tell which entity properties like *'HighBW'* are associated with, i.e. what context they appear in.

Parallel-or composition

If there exist no dependencies between (possibly equivalent) components, the composition can be classified as parallel. *Parallel-or* composition corresponds to the case where one of the components is selected for use.

In the context of policy trading where SP denotes the service profile of a potential policy, and where E denotes the description of the environment, consider the example of combining properties of communication channels (assume that *SuperHighBW ≤ HighBW*).

SP = Channel + SuperHighBW, E = Channel + (HighBW ⊕ SuperHighBW)

The policy requires one *SuperHighBW* channel. The environment offers two channels, one of them satisfy the requirement, which is sufficient to result in a match. Note that if a policy implementation uses one channel, and the environment supports two or more channels, the implementation must be able to select a suitable channel from the available ones.

Parallel-and or sequential composition

Parallel-and composition means that components are unrelated but all are used in combination. For the example of startup time, this may mean that the longest time must be used as the result, since the components can startup in parallel, but we need all to finish. For certain capacity characteristics, parallel-and may mean that the result is the sum of the component capacities, e.g. for processors that can run in parallel without need for synchronisation. *Sequential* composition means that there are dependencies between constituent components, for instance that one cannot run before the other has finished. For the case of startup time this would typically mean to return the sum of component times.

Consider the following expressions.

SP = Channel + (HighBW ⊕ HighBW), E = Channel + (SuperHighBW ⊕ HighBW)

The policy expresses the need for the satisfaction of two separate channel entities by using the component-sum operator, and only environments with at least two channels will satisfy it. This is the way to express parallel-and or sequential composition. Observe that the expression pattern is the same for these two cases. In essence, policies express the need for multiple resources. How these are combined (parallel, sequential or something in between) is an implementation issue.

[17]This particular case could be viewed as a possible conflict which to be resolved to provide interoperability (cf. section 4.5.3).

However, environment descriptors may need to include additional constraints on how components can be combined by policy implementations; for instance, they may explicitly disallow parallel composition. Such constraints may be specific to component types or application domains. The only way to express them, is to define basic profile types denoting composition constraints or location, and to use these in profile expressions. Figure 4.12 illustrates how we may define a profile model for channel entity profiles:

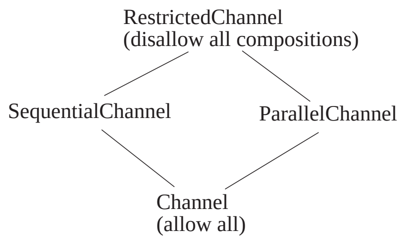

Figure 4.12. Constraints on composition

For instance, a pair of channels which can only be combined sequentially may be described like this:

$E = SequentialChannel + (SuperHighBW \oplus HighBW)$

Observe that a profile which denotes a restriction of composition compared with another profile, would be a *superprofile* of that other profile (a requirement for a more capable channel is a stronger constraint on the environment). A requirement for a resource which can be used in any way, is a stronger requirement than a resource with restrictions on the use.

It may also be necessary to state the *roles* of each component or the context they are expected to be used in, as additional constraints. Consider the following expression:

$E = (SeqSourceChannel+ SuperHighBW) \oplus (SeqSinkChannel + HighBW)$

Here, the basic profile types *'SeqSourceChannel'* and *'SeqSinkChannel'* describe channels which could only be used with the source and the sink of some stream when used in sequential composition.

Nested composition

An entity descriptor may be a property of another entity. For instance, we could have a component-sum of two entity descriptors describing the client and the server respectively. Each of these could contain a set of sub-entities to describe local services and resources at each side. Given an entity *e* with properties $p_1, p_2, \ldots p_n$, if at least one property p_i is a entity descriptor *f* with its own properties $q_1, q_2, \ldots, q_m$, it is necessary to separate this part from the other properties of *e* by using the component-sum operator. Expressions should be on the form:

$e + ((p_1 + \ldots + p_n) \oplus (f + q_1 + \ldots + q_m))$

For instance, if we have a channel with the property *'HighBW'*, which also has a special subchannel with the property *'SuperHighBW'*, it cannot simply be expressed:

$Channel + HighBW + (SubChannel + SuperHighBW)$

According to the associative law, the parentheses do not mean anything, and the sum *'HighBW + SuperHighBW'* is actually equivalent to just saying *'SuperHighBW'*[18]. Therefore, this expression does not capture that the outer and inner components have different bandwidth constraints. Instead we need to express this as a component-sum of the outer and the inner components:

Channel + (HighBW $\oplus$ *(SubChannel + SuperHighBW))*

for which the normal form is:

(Channel + HighBW) $\oplus$ *(Channel + SubChannel + SuperHighBW)*

Limitation of nested composition

Consider the example:

E = (Server + ((Disk + LowLatency) $\oplus$ *HighPerformanceCPU))* $\oplus$
(Client + LowPerformanceCPU)

It may seem that expressions can express nested (or hierarchical) composition. In this example a disk and a CPU component appear in the context of a server. Observe that the nesting hierarchy is flattened when transforming to the normal form. However, we do not lose all nesting information this way. For instance, '$(a+(b\oplus c)) \oplus d$' is equivalent to '$(a+b) \oplus (a+c) \oplus d$'. If any constraints are associated with a containing context, any subcontexts would inherit those constraints. This follows from the distributive law. An identification of the context (i.e. an entity name) should also be viewed as nothing more than a constraint. One could view normalisation as writing the expression as a component-sum of leaf constraints along with (the constraints of) the contexts they appear in. The top level context would appear repeatedly for each operand. This means that our model captures nested composition only in a limited sense; i.e. the component-sum separation cannot actually be nested. To see this clearer, consider the following example:

SP = (Server + ((Disk+LowLatency) $\oplus$ *HighPerformanceCPU)*
E = (ServerA + Disk + LowLatency) $\oplus$ *(ServerB + HighPerformanceCPU)*

In *SP*, the profile *'Server'* seems to occur once as a shared context for the two sides of the component sum, and it may look like the S*P* expression describes one single server instance. This is not necessarily true; it may be two as well. If *'ServerA'* and *'ServerB'* are subprofiles of *'Server'*, *E* will satisfy *SP*. Strictly speaking, we cannot tell from this expression if *'ServerA'* and *'ServerB'* describe separate containing server instances or just two components (disk, CPU) sharing the same server instance.

We may want to express that the separate components describe the same instance or separate instances. With our current profile model, the way to do this is to use basic profiles which identify particular instances. This is obviously not a flexible or scalable solution, since the instances must be known a priori.

We see from this that our model has significant limitations in expressing nested composition. This should be a case for further work in extending the profile model. A possible approach is to introduce labels to indicate instances. If we for instance want to describe an environment with

[18]If we assume that *SuperHighBW* $\leq$ *HighBW* $\leq$ *NormalBW*.

two separate environments and where there should be separate instances of the entity *'Server'* associated with each component, we might say:

SP_1 = *(x:Server + some-cpu)* ⊕ *(y:Server + some-storage)*
SP_2 = *x:Server + (some-cpu* ⊕ *some-storage)*
E = *(a:Server + some-cpu)* ⊕ *(b:Server + some-storage)*

Here, *E* would satisfy SP_1 but not SP_2. If we say *'x:Server (some-cpu ⊕ some-storage)'* we clearly state that the CPU and the storage must be associated the same server instance. Note that we suspect that this approach may increase the computational complexity of conformance checking and should be carefully evaluated before making any conclusion.

4.6. Concluding remarks

In this chapter, we define a language for dynamic QoS expressions which can be evaluated at run-time for conformance. We define how expressions can be constructed from atomic QoS statements termed *'basic profiles',* using composition operators. Two such operators are defined: The *sum* ('+') which corresponds to simple conjunction and *component-sum* ('⊕') which assumes that the operands denote properties of separate environments and therefore must be satisfied separately. To define the semantics of expressions using both operators, we define a distributive law and a normal form. Based on these rules, as well as conformance rules for pairs of component-sums or sums, algorithms for conformance checking any pair of expressions can be developed.

Concrete models define the basic profile space and explicitly establish conformance relationships between basic profiles. They are typically defined for specific application domains and can be defined as *rule-bases*. They are essentially sets of axioms from which we can infer conformance between any pair of basic profiles. From an axiom set, we may derive a full rule set, covering any pair of profile-types for which there may be conformance. Such a rule set can be directly mapped to executable code which allows efficient conformance checking at run-time. As a proof of concept we implemented a profile model compiler (which also has been useful in analysing consistency and performance issues). The compiler performs a basic semantic check of rules, computes a derived rule-set by using a transitive closure algorithm and outputs conformance checking code.

Our analysis shows that there exist some types of inconsistencies which are detectable by a profile model compiler. It follows from the model that an axiom should *cover* (its predicate should be implied by predicates of) any derived rules between the same pair of profile types. Furthermore, some problems can be detected if additional constraints are introduced. We also observe that the pattern of parallel paths can be a source of some consistency problems. In that case, it would be useful if an axiom could be set to override any parallel derived rules. Cycles in a conformance graph may lead to derived rules which are not easily foreseen by the model designer.

The problem of supporting interoperability between applications using different profile models is comparable with the problem of integrating data-sources described by different ontologies. In analysing interoperability we see that certain conflict types are relevant when integrating profile models from different sources. Domain specific models define profile names and semantics only. Therefore, conflict resolution is mostly limited to semantic conflicts. The sources of problems comes from differences in how expressions are interpreted locally (scaling and confounding conflicts), and in how names and rules are defined (naming, and generalisation conflicts). The latter is partly detectable. In our context, it is possible to define mappings and possibly merge models by defining additional conformance rules. In particular, we can use equivalence to handle synonyms.

When discussing composition, it is important to note that profile expressions are used to express requirements or constraints on composition rather than what composition results in. It is up to a policy implementation how available resources are combined. We observe that we may need to add profile types to represent constraints on how resources can be composed. We also observe that the ability to express nested composition is limited because of the distributive law. A problem which results from this is that it can be hard to express properties which apply to different instances of a profile type.

4.6.1. Cases for further work

Our analysis suggests that we further investigate some possible extensions to the model. Some are higher order constructs which can be defined in terms of the core model. Others may be more fundamental. We believe that the following issues should be cases for further investigation:

- An equivalence operator '=' which is straightforward to define in terms of two conformance rules. We have shown some cases where this is obviously useful.
- A symmetry operator where one conformance rule can be defined in relation to another, meaning that there is conformance exactly for the parameter values where it is not conformance in the opposite direction.
- Overriding of any derived rules (in the case of parallel paths).
- Arithmetic operators in rules to support scaling.
- The core profile model is designed with conformance checking and composition in mind. This is however not necessarily easy to read by humans and we may need some composition pragmatics in addition to the model. We may benefit from defining some higher order syntax, for instance to simplify expressing entity descriptors with constraints like discussed in section 4.5.3.
- Variables (labels) in expressions to distinguish between separate instances as discussed in section 4.5.3.

Chapter 5.
Infrastructure Support

In chapter 3 we gave an overview of our approach to QoS aware binding in open systems. In chapter 4 we mostly focus an expression language to be used in the negotiation process. In this chapter, we describe our experimental work on a middleware level component framework which explores the idea of negotiable policies for bindings. Our experimental work is based on the FlexiBind binding framework, which was originally developed as an extension to the FlexiNet middleware framework developed in the context of the ANSA Phase III programme. The FlexiNet/FlexiBind experiment can be regarded as an early contribution to reflective middleware research (cf. section 2.4.4) and also a background for the research conducted in this thesis. Many of the ideas in this chapter were published in [Hanssen99] and in [Hanssen05a].

The rest of this chapter is structured as follows: In section 5.1, we motivate and give an overview of our approach; in section 5.2, we introduce the FlexiNet framework, in section 5.3, we describe the FlexiBind framework in detail. In section 5.4, we describe how our framework can support negotiation using policy trading and in particular how we can support profile expressions. In section 5.5 we evaluate how our framework can support resource reservation and binding types beyond simple client/server interaction.

5.1. Architectural principles

As indicated in chapter 2, a middleware platform for QoS aware open systems should (1) support negotiable and adaptable behaviour (it should be able to reconfigure itself at run-time to change its internal behaviour) and (2) support the process of negotiating such behaviour. In particular, it should offer inspection of platform properties as well as composition of them. The architecture described in this chapter is founded on the architectural principles of chapter 3.

5.1.1. Binding type

The basis of our investigation is the family of binding models of ANSA [Hayton00], FlexiBind [Hanssen99], OpenORB/OpenCom [Blair01, Parlav03], etc. We believe that these models defines a set of concepts which are useful in an overall binding model. [Parlav03] in particular, focuses on *binding types* which capture various forms of interaction between components in an open system. For example, a binding for multi-party media streaming may need to be handled differently from a binding for remote method invocations between one client and one server. A *specification* of a binding type would define collaborations with (1) *binding participants* (the application components that interact through the binding) and (2) *binding managers* (components that establish and control bindings). Four kinds of collaborations are defined:

1. What kind of interaction (between participants) is supported, i.e. the main purpose of a binding type.
2. How remote interface references are generated and managed.
3. How bindings are established through use of binding manager components and negotiation protocols.
4. How already established bindings are managed (e.g. monitoring and adaptation).

A typical binding type is one that supports remote method invocation (RMI) between a single client and a single server. Other examples include continuous media streaming, group communication, and the publish subscribe pattern. In our experimental work, we start by focusing on the client/server (RMI) case, and how client initiated binding would lead to a session specific end-to-end configuration. However, we believe that the ideas explored here are applicable to other binding types as well (see section 5.5.3).

5.1.2. Flexible bindings

We can summarise the requirements for flexible bindings as follows: Extra-functional behaviour should be negotiable. Such negotiable behaviour should (ideally) be orthogonal to the functional type, and the negotiable behaviour should be adaptable at run-time. In our architecture, such behaviour is the responsibility of the *binding*. A middleware platform should be able to support bindings which realise negotiable and adaptable behaviour.

To address some of these issues, we propose a distinction between bindings and their activations. When a binding is established, it is not necessarily active. Activation typically involves loading the object (and its class) into memory, setting up protocol stacks, transparency objects, buffers and other resources. This distinction is useful, since it provides a separation of concerns between issues relevant for the whole lifetime and scope of a binding and more resource and implementation related issues which can change or be decided at a later stage. It allows bindings where the configuration of protocols and resources is not done yet or even where the policy is not known yet. It also allows adaptation of a given binding (or parts of it) through re-activation.

A *binding* to an object is represented (on the client side) by a *proxy object* which knows how to reach the object's implementation if it is active (typically a name/address if it is remote). A passive binding does not have all resources necessary to interact, but it has knowledge of *how* the binding should be activated.

5.1.3. Binding phases

We can decompose the process of binding into four distinct phases, where the system can perform configuration of a service implementation and where negotiation would be of interest:

1. *Service deployment* (server side binding). An abstract service is made available for clients to bind to, by generating a name and configuring a minimum of protocol stack, such that client can establish bindings and initiate negotiation.
2. *Client binding*. A client is associated with the service. This would not necessarily lead to a complete configuration, since there may still be parts which need to be negotiated.

3. *Activation*. As a result of negotiation, the binding configuration is completed and associated with necessary resources, such that invocations of the service may be carried out.
4. Run-time adaptation by *re-activation*. Existing activations may be taken into account when re-negotiating the policy. It is also possible to encapsulate some adaptation within a single policy if it does not violate the contract.

Note that we distinguish between *component deployment* (cf. CCM, or EJB) which means installing an implementation of an application service, and *service deployment* which means making some abstract service available for remote binding. Component deployment may include service deployment.

5.1.4. Policy and metapolicy

Along with the distinction between binding and activation, we propose a distinction between policy and metapolicy. A *policy* (cf. section 3.2.2) dictates the behaviour of activations, while a *metapolicy* (cf. section 3.3) dictates the behaviour of bindings. Metapolicy components are deployed during binding establishment, and policy components are deployed during activation or re-activation. Note that the concept of metapolicy would capture (parts of) *binding establishment* and *binding management* aspects of the *binding type*. A given binding type may allow more than one metapolicy, but the choice of binding type would constrain what metapolicies are suitable. For instance, certain negotiation protocols may be suitable for client/server interactions only.

A binding will be associated with a *metapolicy* which constrains how and when it is activated, how the policy is negotiated, what scope a policy will have (e.g. invocation, session, transaction etc.), and how the binding is adapted by re-activation in response to changing environmental properties. Our concept of metapolicy captures how services are set up in the service deployment phase, as well as in the client binding phase. A metapolicy may therefore involve implementation decisions constraining the later choice of policy.

Our approach to implementing policies is that each policy is represented by a pluggable piece of code (for instance a Java class). Sometimes multiple policies may share the implementation (or parts of it); for instance, a particular implementation may deliver different QoS, depending on the environmental properties, or it may be instantiated with different resource reservations (as discussed in section 5.5.2) of the operating system, leading to variations in behaviour.

5.2. The ANSA FlexiNet framework

The ANSA FlexiNet framework [Hayton99, Hayton00] is a Java middleware system built to address some issues of configurable and extensible middleware. It allows programmers to tailor the platform for a particular application domain or deployment scenario. The FlexiNet framework can be viewed as a flexible toolkit for creating and (re) configuring ORBs, and stands out as an early contribution to reflective middleware research. It provides a generic binding framework plus a set of engineering components to populate the framework. By appropriate configuration of components, one can achieve many different middleware facilities, e.g. mobile objects, transactions, security, persistence etc. FlexiNet is focused at operational interaction but other interaction types (for instance flows) are possible as extensions.

5.2.1. Reflective interaction framework

An interface on a remote object is represented by a local proxy object, typically a *stub* that converts a typed invocation (method call) into a generic form and passes it through a series of reflective layers (which include a protocol stack) before it reaches its target in the server capsule. Compared with traditional architectures such as CORBA, FlexiNet puts more of the stub functionality into the layer stack instead of in the stubs. FlexiNet stubs use the Java run-time typing information to convert each invocation into a generic form and let the layers in the stack do the rest. This makes stubs so simple that they can easily be generated at run-time by using introspection on the interfaces. This technique has later been adopted by the *Java 2* platform[19]. When stubs are simplified, more responsibility is moved to layer stacks which must include higher level functions such as serialisation, replication, object management etc.

Protocol stacks and layers

The layers of the FlexiNet communication stack can be viewed as reflective objects that manipulate the generic invocation in different ways before it is invoked on the destination object. There is no need for interface specific skeletons, just a generic function that converts a generic invocation object to specific calls on target interfaces. Higher level transparency objects can also be regarded as layers in this architecture.

The generic form of the invocation allows simple interfaces to bind layers together. On the client side, layers implement the CallDown interface which has one operation:

```
public void calldown (Invocation inv)
```

Server side layers implement the CallUp interface:

```
public void callup (Invocation inv)
```

After doing its own part of the work, each layer forwards the generic call to next layer in the stack by calling the calldown or callup method recursively. The invocation object may be manipulated by each layer until it reaches the point where the call is converted either to messages to be transmitted over the network or to a call to the target interface. Returns of the call carry result values with the invocation object in the opposite direction. This model requires that the communication between layers are in form of request/reply pairs. Below the RMI protocol layer, we can only talk about unrelated messages going up or down; thus, at that level and below, each layer must implement two interfaces: MessageUp and MessageDown.

An example configuration.

The first protocol configuration that was realised in FlexiNet was named "green" (by convention, early FlexiNet protocols were named after colours) and based on the REX-protocol [Otway87]. This configuration illustrates what a typical RMI configuration looks like, and it is shown in figure 5.1 below. Most of the layers shown support both clients and servers at the same time; thus, much of the protocol stack instance can be used both as client and as server.

[19]The dynamic proxy construct was introduced in Java 1.3.

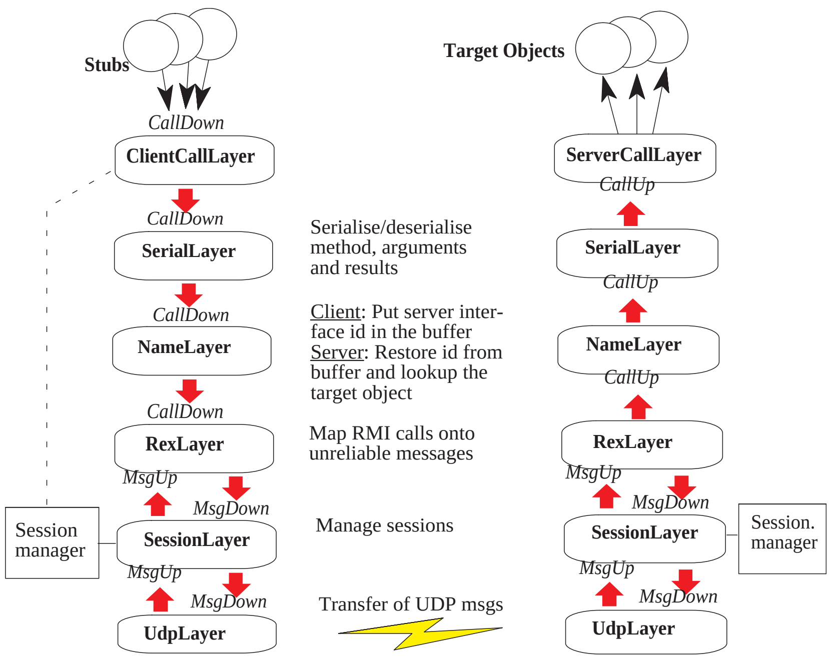

Figure 5.1. Example protocol in FlexiNet

A generic call first goes from a stub to the *client call layer* which acquires a *session* (explained below). The *serial layer* serialise methods, their arguments and results. The *name layer* takes the server interface identifier (which is a part of an interface reference) and writes it to the output buffer. On the server side, the name layer reads this identifier from the input buffer and uses it to look up the target object (name layer represents a mapping from identifiers to target objects). The server call layer uses information from the generic invocation object to generate a method call on the correct target object. The *rex layer* (and the session layer) provides RMI semantics over an unreliable message transfer service.

Sessions

Many RMI protocols maintain state across a number of invocations or messages. For instance, an UDP based protocol (like REX) may need to keep track of unacknowledged replies, and a TCP based protocol may need to maintain a connection. FlexiNet provides *sessions* as an abstraction for managing such information. Sessions are also used to provide concurrency control in the protocol stack, essentially by using per-session locking. A session is typically related to a client thread's association to a server; i.e. on the client side, there is typically one session per server (per thread), and on the server side, there is typically one session per client (thread). When messages arrive from the network or invocations arrive from stubs to a protocol stack, they first need to be associated to a session. In the above example, the client call layer acquires a session object from a *session manager*. For downgoing messages, the *session layer* writes a session identifier to the output buffer which is read by the receiving session layer and used to look up (or create) the session object there. A session object typically contains the RMI protocol state, but also a dictionary to be used by higher layers to store session related information.

5.2.2. Protocols, names and binders

The concepts of names, protocols and binders are keys to extensibility of FlexiNet. The FlexiNet architecture allows different types of *names* (interface references to possibly remote objects). Binding on the server side involves generating a name for the interface to be exported (and registering a mapping between the name and the target). A name contains the information clients need to bind to the target. A name consists of a *protocol name* and protocol specific information needed to locate the target. This is typically a port address plus an identifier for the interface instance.

To support different protocols, different *binders* can be provided. On the client side, binders resolve names of remote objects to *proxy objects* (stubs), together with the proper communication stacks (which typically should be shared between bindings). On the server side, they generate a name for a target interface plus the necessary means to be able to receive invocations. Thus, there are two types of binders: *Resolvers* on the client side and *generators* on the server side. FlexiNet allows many binders and protocols to coexist within the same process; hence, there is a need for a means to dynamically select the correct binder to be used for binding. Each layer stack instance would therefore contain references to binders to be used for generating and resolving names or interfaces passed as arguments or results of invocations. It would typically be the responsibility of the *serial layer* (section 5.2.1) to call the resolver or generator, respectively, when attempting to serialise or deserialise remotely invocable interface references.

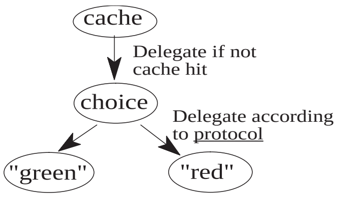

Figure 5.2. Binder delegation hierarchy

Binders can also be arranged into hierarchies, in order to factor out common functionality or to allow dynamic selection of a binder to be used for a particular binding. An actual binder could be a composition of simpler binder components. Figure 5.2 shows a simple (and static) example of a binder setup. There would typically be a cache at the top, which keeps track of recently resolved bindings. More dynamic selection is possible. In addition to the protocol part of the name, applications may also use QoS requirements as arguments to the binder, in order to select the binder which best matches the requirements. Delegating binders can for instance look up another binder to delegate to, or they can negotiate with the server to do the selection.

5.3. FlexiBind - an experimental binding framework

We have designed an experimental framework to demonstrate binding using pluggable enforcement policies and metapolicies (cf. section 3.1, 3.2.2 and 3.3). The prototype implementation of our framework is built upon the FlexiNet architecture. The FlexiBind framework can further be extended to support a family of protocols. Protocols, policies and metapolicies can be "plugged" into the middleware. FlexiBind allows late binding, explicit binding and dynamic adaptation of layer stacks.

5.3.1. Activations

An important principle of the FlexiBind framework is the distinction between bindings and their activations. Binding can then be done without knowing (yet) all about how the binding should be activated. This distinction allows *late* activation (wait until the first invocation is made), *explicit* activation or even *per-invocation* activation. Adaptation can be supported by allowing changes to activation during the lifetime of a binding.

The activation object

To reify meta-level objects involved in carrying out invocations, and to simplify the reasoning about activations in relation to bindings, we introduce the *activation object.* It represents a configuration of layers. Activation objects are in principle constructed at each invocation and is carried with the generic invocation object through the layers. In practise, invocations typically share activations created explicitly or at the first of a series of invocations. Each activation object contains an ordered list of references to the layers of the activation.

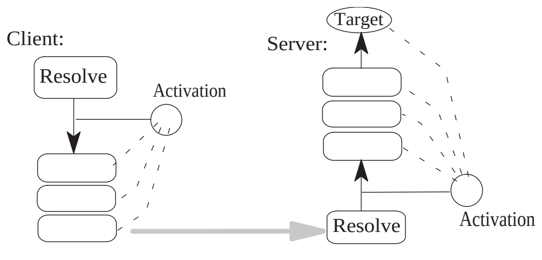

Figure 5.3. Resolving activation objects

Figure 5.3 illustrates how the layer configuration is determined by resolving to activation objects. On the client side, the target name (on the invocation) is resolved to an activation object which is used to perform the invocation. On the server side, a target name (passed over by the protocol) is resolved to an activation object that prepares and invokes the target method.

Channels

When a server exports an interface by generating a name for it, some protocol information must be passed along with the name, such that clients know how to establish bindings. Therefore, the purpose of the *protocol part* of a name would be to identify the minimum protocol support needed to activate a binding to the exported interface.

This observation also suggests that the activation (layer stack) could be divided into two parts at each endpoint: (1) A *protocol dependent part*[20] which is identified in names generated by servers, and (2) a *protocol independent part* which can be resolved/activated dynamically. On the server side, the protocol dependent part would normally be the minimum setup needed to listen for incoming invocations and to initiate dynamic activation. On the client side, this means that we know at binding time what the protocol-dependent part of the activation should look like. However, it does not necessarily have to be activated before the rest of the stack is activated.

[20]This corresponds to the inner binding (nested binding) discussed in section 2.3.1.

There will typically be a small set of such configurations, corresponding to typical protocols and in many cases, listening to specific addresses for incoming connections. They are often installed early, re-used and shared. Therefore, it is useful to encapsulate common protocol dependent configurations as *channel* objects. A *middleware platform* should offer access to one or more default shared channel objects and/or an interface to instantiate channel objects.

5.3.2. Bindings

A *binding proxy* represents a binding which is not necessarily active. When activated, a binding proxy has an *activator* object attached to it. This activator is responsible for managing the layers and other resources forming the activation. It is useful to have activators as separate objects (and components), in order to encapsulate various ways to manage activations and to support adaptation of bindings by installing or replacing the activator. In this way, a policy is represented by an activator component. Each binding proxy implements the interface Binding which defines operations to explicitly *activate*, *passivate,* or to set or replace the activator. Figure 5.4 shows an UML class diagram for the binding framework.

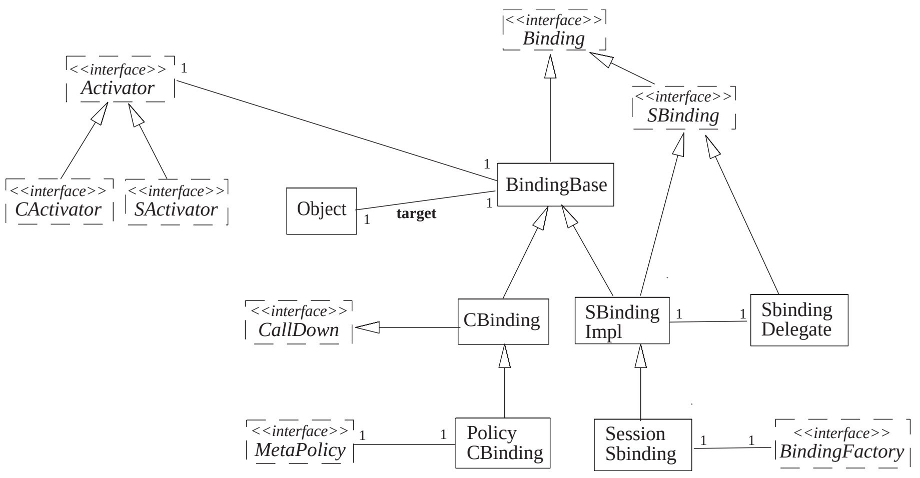

Figure 5.4. Class diagram for binding framework

Client side bindings

Figure 5.5. below illustrates how client side bindings activate and invoke operations. When a remote name is resolved on the client side, it results in a proxy object which can be used in place of the real target object. Proxies automatically generated from interface types are just stubs converting typed invocations to generic invocations. Therefore, the rest of the proxy object containing remote addresses, activation information etc., is placed in a separate object, i.e. the *binding object* (CBinding). This object implements the generic invocation interface (CallDown) and can therefore be regarded as a layer as well.

The scenario of figure 5.5, is as follows: When receiving a generic invocation from a stub (1), the CBinding asks the *activator* for an activation object (3) by calling its getActivation method. The activation object contains references to the layers to which the invocation is sent (4). If necessary (typically at the first invocation), the binding needs to *activate* (2), which means to instantiate the activator. Layers in the activation's list, except the last one, may return to a special *switch layer* which determines to what layer the invocation is sent next (5). Layers may be linked directly which is slightly more efficient but also slightly less flexible.

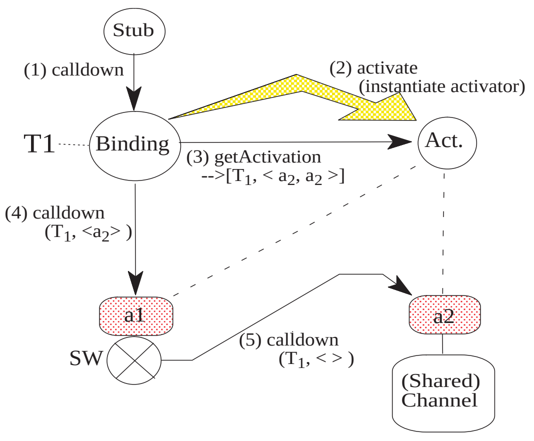

Figure 5.5. Client side binding - Activation, resolving and call scenario

Policy bindings

To support pluggable metapolicies, we introduce the PolicyCBinding class, which is a subclass of the CBinding class. Each instance of this class may contain a reference to a *metapolicy* object. Each metapolicy object implements the MetaPolicy interface which defines the pre_activate, post_activate, pre_passivate and post_activate methods. The PolicyCBinding object will call those methods before and after calling the activate and passivate method. These methods modify the behaviour of activate and passivate, for instance by negotiating with the server to find and install a suitable policy.

Server side bindings

Server side binding objects (SBinding, etc.) are associated with activators and real target objects. Here, we explain the basic design. A *binding layer* (corresponds to the FlexiNet name layer) is always a part of a server channel. At each incoming invocation, the binding layer reads an interface identifier from the input buffer (the id is a part of a name) and uses it to look up an activation object. To do the lookup, the binding layer consults a *binding manager* which maintains mappings from interface identifiers to binding objects representing the targets. When exporting an interface (service deployment), the server creates a binding proxy, an identifier and a mapping between the identifier and the binding proxy (cf. the object adapter in CORBA).

Figure 5.6 below illustrates server activation, by showing a scenario of what happens with an incoming invocation: The binding manager is asked for an activation for a given identifier, the binding manager looks up the binding proxy and asks it for the activation (1). A binding has an activator object if it is active. If not, it needs to be activated (2), before the getActivation method can be used (3). Typically, the activator sets up the layers of the activation when instantiated and return references to them through the return value of the getActivation method. On the server side, the *getActivation* method has the following signature:

```
public ServerActivation getActivation
        (Dictionary session, Object target)
```

The invocation which now contains a valid activation object then goes to the a *switch layer* (4), which calls the first layer in the activation object's list of layers (5). The call goes back to the switch layer which calls the next layer in the list. Before a layer is called, the reference to it is removed from the list. When the list is empty, the operation is finally invoked on the target object (6).

5.3.3. Binder framework

A FlexiBind based ORB can dynamically be configured for different policies and metapolicies by plugging in binder and activator components. These components are implemented as relatively simple extensions of the binder framework.

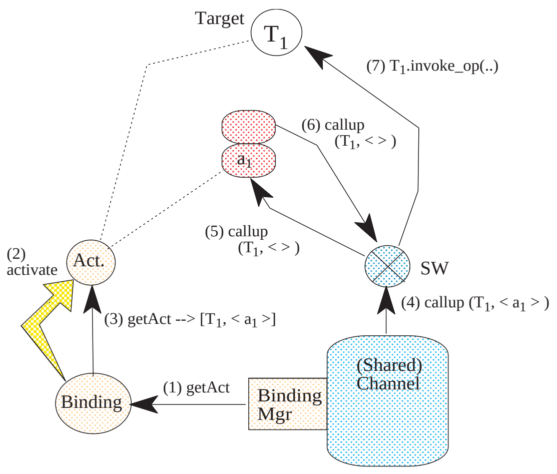

Figure 5.6. Server side binding - activation, resolving and call scenario

Binder components

A *binder* knows the metapolicy and a specific *protocol* (in the FlexiNet sense). In our context this means both how clients bind to the server and how to negotiate an activation policy between clients and servers. A binder is responsible for instantiating and configuring binding proxies, including associated activators or activator selection policies. On the client side, they are usually connected to stubs, and on the server side, it will be registered in the binding manager. Binders act as resolvers or generators (like in e.g. FlexiNet and OpenORB). Generators are involved in service deployment and resolvers in client binding. Client binder classes implement the Resolver interface, and server binder classes implement the Generator interface.

On the client side, a resolver is responsible for resolving names of remote interfaces to binding proxies (and stub objects) that can be used to invoke their operations and possibly before that, to perform negotiation of policy. On the server side, a generator creates names for interfaces to be exported, and it establishes associations between the names and the target interfaces (through binding proxies). A binder component (generator) implements the generateName method. It takes the target object and class as arguments, it creates and configures binding proxies and it generates names for the interfaces. Mappings between the identifier part of names and binding proxies are registered in the binding manager.

```
public Name generateName
  (Object target, Class target_cls, FlexiProps qos)
```

A corresponding resolveName method takes a name as an argument and returns the top layer of the invocation stack (a binding proxy). Note that our framework provides a common base class for resolvers which in addition performs the stub generation, since this is common to all client binders.

```
public CallDown resolveName
  (Name name, FlexiProps qos) throws BadName
```

Activators

An *activator object* is responsible for allocating necessary resources and configuring the activation. A very simple binder could just have the activator set as an object property. More advanced binders may decide more dynamically what activator component to use, or they may set up a binding proxy with a metapolicy. When handling invocations, the binding proxy would invoke the getActivation method of its current activator object to get a reference to the activation (see section 5.3.1). Activator components may configure the activation (layer stack, resources etc.) when instantiated, but it may also change it later. This allows activators to perform a limited kind of adaptation. Adaptation which involves re-negotiation is done by replacing the activator. An activator implementation may choose to re-use some of the configuration of a former activator.

5.3.4. Session management

Given some shared service, different clients or client threads may need to negotiate separate policies for their bindings to it. Furthermore, a client thread may want to share such a policy among a series of subsequent bindings. For instance, objects which are returned from method invocations to an object bound to some moments earlier should often just inherit its policy.

A session is a series of invocations originating from the same client or client thread. In this section, we provide a more detailed discussion of how our middleware is engineered, with focus on how it supports per-session state and per-session negotiation. For a given service, we may want to use separate activations for separate sessions. In some cases, we may want bindings to be established within the same session, in order to share a policy negotiated when opening the session. Therefore our middleware architecture should support management of session specific state on the server side.

Session specific bindings and activations

We may want some binding information to only be visible in the context of a session. To keep track of session specific state on the server side, a session layer keeps track of active sessions. For each active session there is a session object containing naming context (a dictionary). The session object is used by the RMI protocol layer, but is also carried with each invocation object to be available for higher level layers as well. Session specific bindings are registered in a session dictionary instead of in the binding manager (from figure 5.6). When the binding layer asks for the binding-proxy corresponding to some name, the binding manager first looks in the session dictionary (bindings registered here would therefore be visible only by invocations belonging to the same session) and if no binding is found there, the binding manager's global naming context is searched (bindings registered here are visible by all invocations).

When the binding layer asks binding proxies (via the binding manager) for activations, the session object is always an argument. It is up to the attached policy to register session specific bindings. Figure 5.7 illustrates how some bindings can be registered in the session dictionary and therefore are visible only by invocations belonging to the same session, while other bindings are global (visible by all invocations). Different bindings to the same target may have different activators which creates different activations. In a typical RMI setup, a session manager manages session specific state in the RMI layer, but session objects also contain a generic dictionary which could be used by the binding layer.

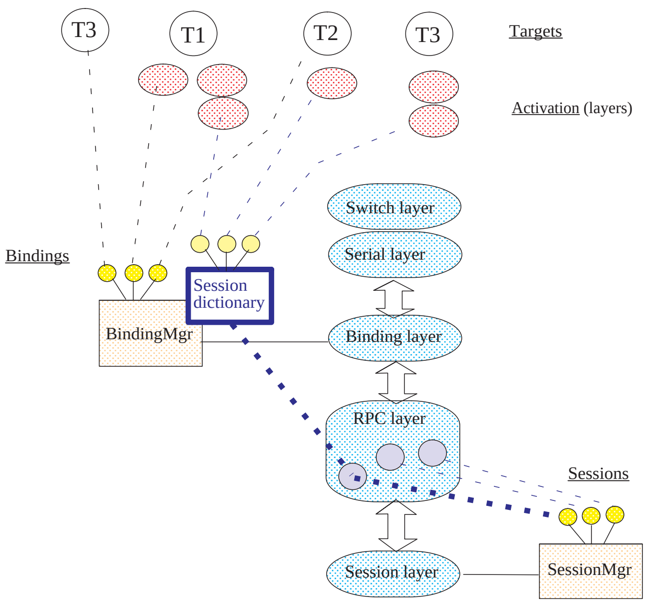

Figure 5.7. Session bindings

Remote binding management

To demonstrate certain aspects of negotiation with our framework, we now describe a simple scheme for client-initiated activation on a server, using sessions and a special control interface. This implies a metapolicy where the client session just selects a policy and installs it on the server. We observe that more sophisticated protocols for negotiating and installing policies follow some of the same patterns.

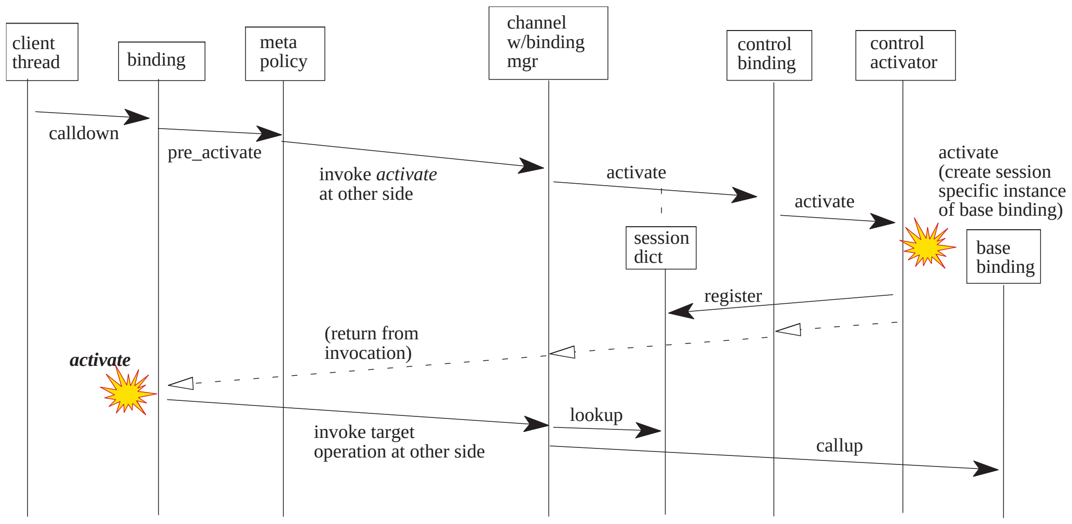

Figure 5.8. Remote binding management sequence diagram

During service deployment, the server side *generator* will in addition to the target interface itself, export the Binding interface (the binding proxy), such that it can be invoked remotely. Generated names contain two identifiers. One for the target itself (base) and one for the Binding interface (control). These two identifiers map to two binding proxies (base-binding and control-binding). Client metapolicies can then invoke the control interface to activate the binding for the target interface. On the client side, the resolver creates instances of the PolicyCBinding class (which is a subclass of Binding). Each such binding proxy is connected to a MetaPolicy object (section 5.3.2) which in the pre_activate method remotely invokes the activate operation on the server side control interface (Binding). *Base* binding proxies are session specific (one per session). The control binding's activator creates such a base binding proxy the first time an invocation asks the control binding for the activation for a given session. This is registered in the session dictionary, and the remote invocation returns. Then, the client side activates and can carry on with the invocation to the actual target. Figure 5.8 summarises this in a UML sequence diagram.

Generators and binding factories

A control binding is a separate binding to an interface which can be used to control or negotiate the behaviour of another binding. In the remote installation scheme described above, there are no session specific instances of the control binding proxy. Therefore the scheme is limited to cases where the client side decides on the policy and performs server side configuration in a single invocation on the server control interface. In general, we may need session state also for the negotiation process itself. To support this, we extend the simple scheme above to one that can produce session specific control bindings.

The idea is to distinguish between *generators* (which are used in service deployment) and *binding factories* (which are used for client binding) and let the generator instantiate a special control binding, which contains a binding factory. This can be used later to create session specific bindings. Note that this scheme still operates with two identifiers in the name and the session specific bindings are handled mainly as described above.

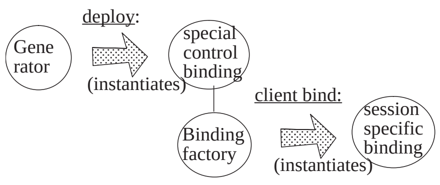

Figure 5.9. Generator vs. binding factory

Figure 5.10 below illustrates this in more detail. At the beginning of the session, the activation of the incoming invocation is looked up (1). The special control binding proxy (created by the generator) asks its binding factory to create a new session binding (2 and 3). The resulting binding proxies (base and control) are registered in the session dictionary (4), using the identifiers produced by the generator. The getActivation call can now be forwarded (5) to the session specific control binding proxy.

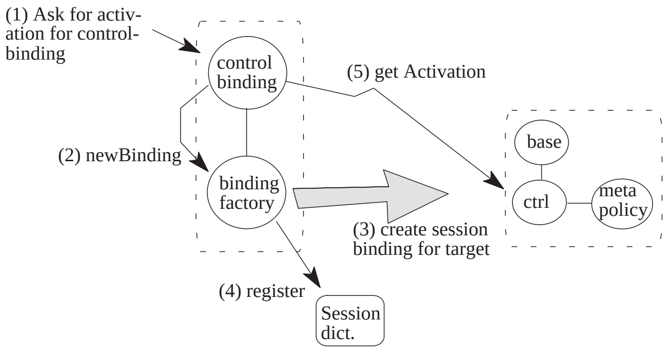

Figure 5.10. Session specific binding scenario

Session shared policy

In many cases, we do not want to negotiate the policy for every binding in a session. We rather want to do this only for the interface initially bound to in a session and let the objects reachable from it share both the metapolicy and the policy. This is consistent with the principle of reachability persistence [Atkinson87].

A simple approach to implementing this is to introduce yet another type of binding proxy (BindingDelegate). This proxy simply delegates policy related operations to another binding proxy. The binder (generator or resolver) for the initial binding can do an extra registration in the session dictionary for the resulting binding, using a distinguished identifier value. Subsequent binding activity (serialising and deserialising interfaces in operation results) within the session will use another generator or resolver which simply looks up the initial binding in the session dictionary (using the distinguished identifier) and re-uses its policy.

5.4. Negotiation aware bindings

Section 5.3 presents the design of our middleware framework and discusses some issues related to session management on the server side. In this section, we discuss how we can support negotiation based on policy trading and profile expressions. Negotiation support will be by extensions to the framework described above.

A main problem addressed here is how to generate environmental descriptions describing various parts of a system and compose them to environment-descriptors. These expressions could be matched with service profiles of candidate policies, using a trading service (as described in section 3.3). The openness of the architecture implies that we can not always know in advance what components are involved and how the components and the platforms are configured. To address this, we propose the use of dynamic profiles and inspector objects, and that these are configured mainly by *binder* components. We demonstrate how negotiation can be realised using metapolicy objects and an example RMI based protocol.

5.4.1. Architecture overview

Negotiation is performed by *negotiator* metaobjects that can be attached to bindings (cf. PolicyBinding framework in section 5.3.2). They implement the pre_activate, post_activate, pre_passivate and post_passivate methods to add negotiation of policy to activation/deactivation. On the server side, a binder would set up a negotiator metaobject. This object offers a special interface to be remotely invoked by client sessions, in order to perform the negotiation (like the control interface in section 5.3.4). Figure 5.11 below illustrates the binding setup where the *target* represents the actual application object at the server, or a stub object on the client side. This is attached to the binding proxy which again is attached to a negotiator, a channel and a dynamic profile (which is essentially an abstract syntax tree of a profile expression). Parts of this tree can refer to *inspector objects* that can generate sub-expressions by inspecting various properties of the platform (for example channels). The dynamic profile can be evaluated at run-time to return complete profile expressions.

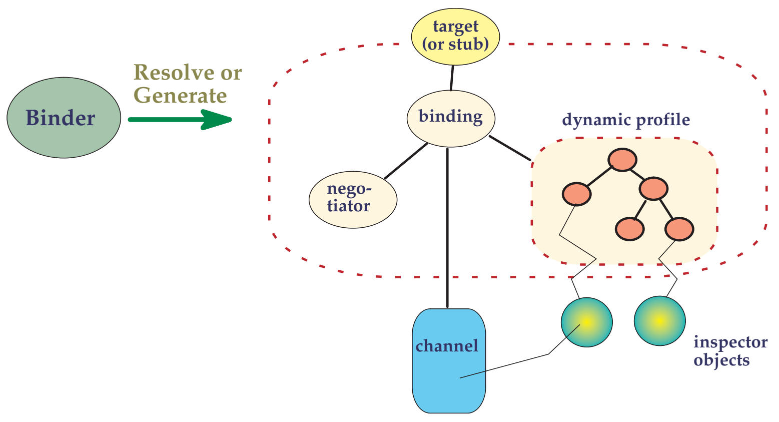

Figure 5.11. Architecture of bindings supporting negotiation[21]

Example protocol and setup

To demonstrate how negotiation can be supported, we provide a relatively simple setup. We assume a client/server model for simplicity. This is an example of an *optimistic* protocol in the sense that it assumes a low probability of an activation failing due to, for example, insufficient resources (failure will require a recovery procedure which is relatively costly).

A policy-trading service is located on the server side. It is used by the server side negotiator meta object to find a set of candidate policies. The trader database contains a set of policies, each having a reference to a client activator and a server activator component (in our prototype we simply reference the corresponding Java classes)[22]. The server side negotiator offers a remote interface to client negotiators where the following operations are offered:

- get_Activation. Start the binding process at the server. It takes the user-requirement and the client side environment description as arguments. It creates a prioritised list of candidate policies, and it returns the client part of the first policy to be successfully activated on the server.
- retry_Activation. Tell the server that the client part of the policy failed and that the server should try another one.
- activation_OK. Tell the server that the binding process has succeeded and that the server may now throw away the list of candidate policies.
- release. Close the binding.

Note that this is a stateful protocol, meaning that a server must keep a per-client record of the negotiation until the activation_OK or release operation is invoked. Figure 5.12 shows a trace of a successful negotiation. A corresponding client side binder would set up a negotiation metaobject (negotiator) which at the first invocation, or when explicitly requested, computes the two profile expressions (application requirements, environment descriptor). These expressions are used as arguments to the get_Activation method. The server will then add its own requirement and environment descriptor and call the trading service which returns a (possibly prioritised) list of policies.

[21]All the thin lines in the figure represent references between objects. The thick arrow represents instantiation of objects.

[22]For multiparty bindings with a non-fixed number of participants it may be appropriate to add another level of indirection; i.e. we may reference types or groups of activators instead of referencing activators directly.

The server attempts to install the first policy on that list. If this activation fails, the next policy from the list is selected, etc. When activation succeeds, the invocation returns with the corresponding client side activator component. The client attempts to activate and if this succeeds, the activation_OK operation is invoked to confirm that the negotiation can be finalised, resources can be released, committed, etc. If client activation fails, the retry_Activation can be invoked to proceed with the next possible policy.

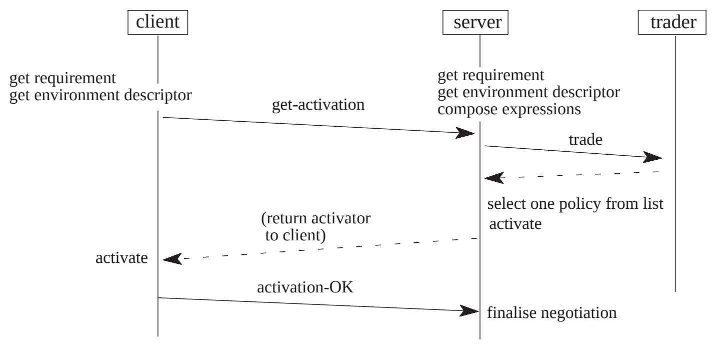

Figure 5.12. Example of successful negotiation

5.4.2. Dynamic profile expressions

As indicated above, the negotiator metaobjects would need to produce and exchange requirements and environment descriptors. The requirement part comes from the application program which deploys services or binds to services. A problem is how to produce the environment descriptors, and we believe that this is a metapolicy issue, since the relevance of the various properties would depend on the application, the binding type, the platform, the channel used, etc. As a consequence, the binder would set up the necessary structures to produce such expressions (see figure 5.13 below) per binding instance (each service deployment or client binding).

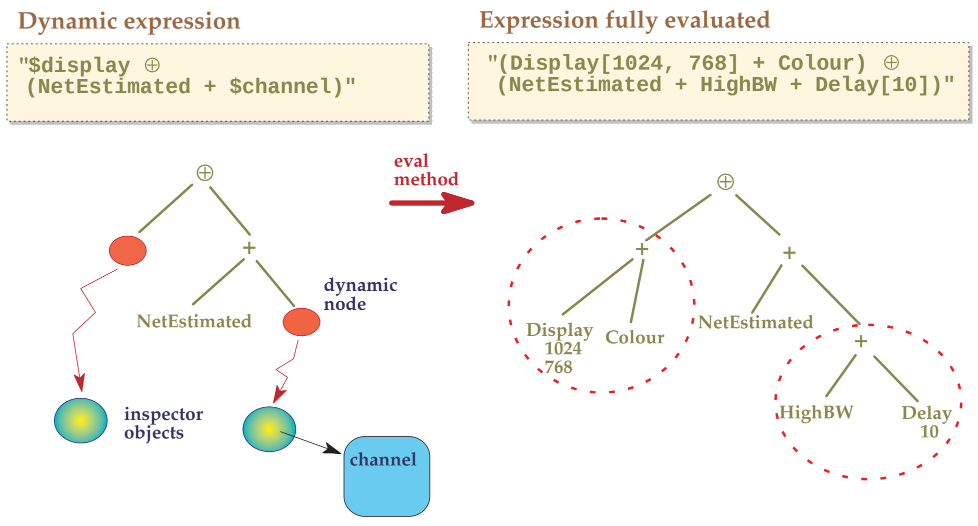

Figure 5.13. Evaluation of dynamic profile expressions

Some parts of the descriptors may be static since they do not change during the lifetime of a binding. This may be the case for platform properties like display resolution or the availability of certain resources or channels. However some properties may change due to varying load etc. Some may also depend on the location of the peer, like for instance estimated end-to-end network delay. Such descriptors cannot be fully provided before the time of negotiation. Therefore, we propose a dynamic profile expression scheme: A binder will set up a *profile expression tree* (corresponds to an abstract syntax tree). Parts of it may be dynamic; i.e. we use a special type of tree node which must be evaluated at negotiation time to get a complete expression.

With this scheme we can set up the composition of expressions from different parts of the system as expressions embedded in the binder code. The construction of a dynamic profile can be done by parsing a textual representation, or by explicitly invoking constructor methods corresponding to either leaf nodes, sum or component sum operators. For instance, an expression may be constructed like this in our prototype:

```
Expr disp =
    Expr.parse("$display ⊕ (NetEstimated + $channel)");
```

We use the special prefix '$' to denote dynamic nodes. In this example we assume that there exist a dictionary where inspector objects are registered. The name following the '$' symbol is used to look up such objects. We further discuss scripting and namespaces in section 5.4.4 below. Other prefixes are possible as well (for example, our prototype implementation offer the prefix '@' for Java classes to be instantiated).

In our prototype implementation, the abstract class Expr represents expression trees and offers much of the functionality related to constructing, composing and resolving dynamic profile expressions. Note that the conformance checker code also operates on Expr objects. The most important methods are :

- parse(String) → Expr. Generate an Expr implementation (essentially an abstract syntax tree) from a textual representation of the expression.
- evaluate(Expr) → Expr. Resolve dynamic nodes and normalise expression.
- SUM(Expr, Expr) → Expr. Construct a sum expression from the two arguments.
- COMPSUM(Expr, Expr) → Expr. Construct a component sum expression.

5.4.3. Inspector objects

To support dynamic profiles, we introduce *inspector* components. Their role is to generate profile expression fragments describing platform specific facts or measurable properties of the system, when requested. They can be viewed as a generalisation of e.g. QuO condition objects. Inspectors offer an interface with a method getProfile which returns expressions. A dynamic profile node contains a reference to an inspector, and inspectors may be shared between profile-expressions. Inspectors may be installed by platform configuration to report properties of platform wide resources, they may be installed by channels, or they may be installed by binders to report properties of individual bindings.

Some inspectors would need to be configured with a *target object* (a reference to a local implementation or to a remote interface reference), since they may produce measured properties which depend on the location of the target. Other inspectors may not need to be associated with a target, but rather with the platform or resources available. Examples of what inspectors can do include:

- Return the actual size of an application window on the screen.
- Estimate end-to-end invocation time by invoking probe operations on the remote system. An inspector could for instance return a profile *"RTT[n]"* (where *n* is a number denoting the round-trip delay time in milliseconds). More sophisticated implementations could use policy

specific interceptors or layers in the invocation chain which monitor the time for real operations, however requiring access to an existing activation.

- Determine by probing, if the remote system is reachable by the UDP protocol (not always the case if end-systems are on different IP-subnets). This can be useful if policies use UDP based invocation protocols or RTP for continuous media streams.
- Estimate the amount of a resource (e.g. network bandwidth) to be available for reservation. This would obviously require some operating system resource management service and a middleware level extension to interface it (see also section 5.5.2). A reservation could for instance be implemented as a scheduling class which guarantees that its members (e.g. threads) get a certain share of the resource.
- Estimate the load on the CPU, network interface or other resources on the platform. This could be combined with (more static) operating system level resource management, implementing performance isolation.

When considering resource reservation, for instance class based resource management, observe that the negotiation scheme cannot guarantee that reservation will succeed, unless the middleware is given exclusive access to the classes of interest by the operating system, and unless the negotiation protocol provides proper concurrency control with respect to resources of interest. Resource management is a large area, mostly outside the scope of this thesis. However, we evaluate to some extent how to interface with resource reservation in section 5.5.1.

Inspector composition

Inspectors can be composed in the sense that one inspector may use the result of other inspectors, and possibly modify or add its own subexpressions. For instance, a metapolicy may wish to specialise the behaviour of the platform's display inspector, to reflect that only a certain part of the display could be used. A binder could for instance install a special inspector which delegates to the platform level display inspector but modifies its output.

5.4.4. Middleware configuration

Different instances of a middleware platform may be configured differently with respect to inspectors, channels and other resources. A binder may add configurations to the platform as well; in particular, it may instantiate inspectors to be used when activating its bindings. A policy component should be written with a range of configurations in mind, rather than a specific one. We therefore need middleware abstractions to represent configurations. The *policy programmer interface* scheme proposed in [Hanssen99] can be used to offer platform services and to check if a policy component can be used on a given platform. Different configurations are expressed as different interface types. Type conformance is used to check if a policy can be used with a particular environment. This is however a rather static scheme which does not easily support components which are dynamically installed by policies, and it does not easily support an arbitrary number of instances of the same type (which would be the case for inspectors). This is addressed by another (complementary) approach, which is to let platform and policy components provide nested namespaces (dictionaries) for objects.

Naming and scoping

Binder components are meant to be pluggable into various platform configurations. Hence, we want to abstract over how inspector objects are implemented and installed. We observe that (1) the platform may set up some, (2) channels may set up some, (3) binders set up some, and (4) some can be metapolicy specific (set up by binders) but shared between the bindings sharing a metapolicy. Binders should be allowed to discover, use and compose such objects. This can be supported by using naming mechanism as suggested above. We also need *scoping*, meaning that a mapping may be defined to work in the context of a policy, metapolicy, binding etc. If a name is not resolved in a local scope, we try a wider scope, etc. Scoping is organised as in figure 5.14. The scope of a binding will also include the scope of the platform. We may want to override a name defined in the platform scope. For instance, a metapolicy may wish to specialise and replace the behaviour of a (platform scope) display inspector. An inspector object may be placed in the scope of a binding and this may again use or modify the behaviour of a platform inspector using the same name. This corresponds to overloading a name in a typical programming language.

This means that a naming context would be associated with each binder, binding, channel and platform and that they are appropriately linked with each other. Note that naming contexts can also be used for other components or resources than just inspectors.

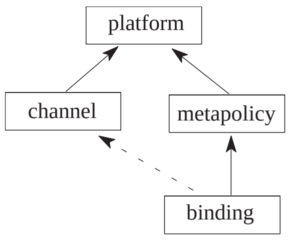

Figure 5.14. Naming contexts (arrows mean inclusion)

Scripting

The use of a run-time naming and scoping mechanism leads us to the idea of defining dynamic profiles as script fragments embedded in binder code[23]. It is convenient for a metapolicy programmer to embed textual representations of expressions and let the middleware evaluate them. In such expressions one could use prefixing to distinguish parts which are expanded at negotiation time, from profile model names. Names with the '$' prefix refer to installed inspectors which are looked up like described above. For example, a client binder sets up an inspector named '*rmi-channel*' which returns the properties of an available RMI channel. An inspector named '*display*' is set up by the platform and returns the properties of the display. The client binder code could contain the following.

```
descriptor = "Client + ($display ⊕ $rmi-channel)"
```

During negotiation, this expression is evaluated: The dynamic profile parts are replaced by expressions returned by the corresponding inspectors, e.g. *$rmi-channel* estimates bandwidth and delay and returns e.g. *"NetEstimated + HighBW + Delay[10]"*.

[23]Scripting using the scoping scheme described here is apparently a useful tool for configuring platform and policies in general.

5.5. Evaluation

In this section, we experimentally evaluate our framework with respect to two selected topics. First, we investigate how resource reservation in the operating system or network can be incorporated into policy. We sketch some middleware abstractions and validate our approach with a proof of concept experiment. In section 5.5.2, we evaluate how our framework can support binding types beyond simple client/server interaction. We design and implement a simple framework for publish/subscribe bindings to demonstrate how to realise negotiable publish/subscribe bindings with our framework.

5.5.1. Resource management

QoS contracts may include assumptions on how resources at the operating system or network level are scheduled to the activities of the binding. This is important if contracts include guarantees on real-time behaviour. For instance, if service invocations are to be performed within time-limits, the CPU must be scheduled to the application threads within certain deadlines, network bandwidth and delay must be within certain bounds etc. In particular, distributed multimedia applications like video conferencing are known to be sensitive to the QoS of the networking and how the CPU, memory and I/O devices on each platform are scheduled.

It is beyond the scope of this thesis to fully analyse resource management at the system level, but we should clarify what overall assumptions we make about that level. We want to test if resource reservations (enforced by various underlying platforms) can be used by our policy concept through a little set of middleware abstractions and plugins. We design and implement some such abstractions on top of our middleware framework. Based on this, we demonstrate how to bound invocation delay by reserving network bandwidth in a situation of heavy load.

Middleware abstractions for resource management

A *platform instance* is a running configuration of our middleware architecture, deployed on a specific machine, running a specific operating system. A problem is that the underlying operating systems and supporting middleware components are rather different with respect to what can be reserved, what abstractions are offered to request reservations, and what characteristics are used to describe such requests. This heterogeneity is hard to abstract away from, due to the wide range of hardware and application QoS requirements. Therefore, we believe that operations for resource reservation should be provided by pluggable extensions to the core middleware architecture.

The following conceptual model seems to be reasonable with respect to how it can be mapped to various resource management implementations. The highlights are:

- *Resource manager* components encapsulate particular resources or resource groups.
- *Reservation domain* objects are instantiated by a resource manager when admitting a reservation request. They represent abstract reservations.
- The concept of *resource consumers* to which resources can be scheduled. In the context of operating systems this could for instance mean processes, threads or communication sockets.

Resource managers and corresponding sets of reservation domains encapsulate and abstract over platform dependent O.S. and network level resource management. Alternatively, they could use middleware level resource brokers (like for instance in [Chu97]). Conceptually, they have some similarities with e.g. resource groups in RT-Java [RTJ], resource containers or reservation domains in Eclipse or Nemesis (cf. section 2.3.1). In principle, this model could be realised as a meta-space [Blair01], where underlying resource properties of platforms, applications or bindings can be reified as these three types of programmatic objects.

Resource managers and reservation domains

Our architecture is designed to allow pluggable *inspector* components (section 5.4.3). Such components could also abstract over resource reservation and act as resource managers. The idea is to extend the *inspector* interface with an *admission test* operation. This operation takes a variable set of parameters (the actual parameters are specific for the resource manager class) and returns a *reservation domain* object if admission is successful.

```
ResDomain canReserve(Object ... parameters);
```

The *reservation domain* object offers operations to associate resource consumer objects to it, meaning that resource consumption by these objects would be charged to the reservation domain. Assuming that all resource consumers implement an interface ResConsumer, the reservation domain interface could offer:

```
boolean addObject (ResConsumer x);
boolean commit();
void release();
```

The commit operation actually performs the reservation. With this framework, one could do admission testing on a set of resource managers before deciding to actually reserve or not. We should also offer an operation to release reservations.

A *resource manager* and its corresponding *reservation domains* may encapsulate one single resource or a combination of multiple resources (e.g. CPU and network). They can be directly mapped to e.g. operating system level resource contexts or resource containers like e.g. CKRM classes, DSRT or RSVP service classes (section 2.3.1) or RT-Java [RTJ] processing groups. It is also possible to implement a resource manager as a composition of other resource managers. Note that the inspector operation can be used to describe the resource manager and what it can do. A policy specification may state the requirement for a certain type of resource, and policy trading ensures that a selected policy would have compatible resource managers in the platform where it is deployed. For certain types of resources (but not necessarily all), inspection may also include the amount of the resource available for reservation.

Usage in policies

It would mainly be the responsibility of the *policies* to specify reservations used for the various parts of the binding (if such are needed). A separation of concerns between the policy implementation component (activator) and the resource management policy is also desirable, to allow the reuse of an activator component with alternative reservation policies. The framework sketched here allows some separation of concerns between activator and resource reservation policy. The figure below illustrates how a policy consists of four aspects: *User profile* (contract obligation), *service profile* (environment assumptions), *implementation* (represented by an activator) and *resource requirements*. Resource requirements can be formulated as a set of references to resource manager objects, each with a list of parameters to be used with the canReserve operation. The activator does not need to know much about resource manager objects; it can be given the resulting resource domain interfaces to pass to the relevant resource consumers without needing to know their implementations.

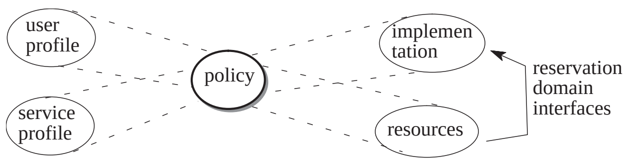

Figure 5.17. Aspects of policy

Resource consumer objects

Resource reservations need to be associated with the activities or data traffic which consume the resources. Operating system reservation services typically use process identifiers (or thread identifiers) for CPU reservations, or file-descriptors (network sockets or files) for network or file system reservations. For network reservations, it may as well be necessary to use IP-addresses and port-numbers to identify a stream. An implementation of a reservation domain could for instance pass such information to a RSVP daemon.

It would depend on the reservation domain type, if and how objects are to be associated with the reservation. Conceptually, reservation domains may offer a method addObject. An activator may call that method for each relevant resource consumer object belonging to the activation or session. Those resource consumer objects would be middleware engineering objects like threads, buffers, certain types of layers, sessions, or channels. The addObject method of each reservation domain would inspect the objects and, if needed, extract the more platform specific information, like thread identifiers, file descriptors or IP addresses. This approach has the advantage of encapsulating platform and resource specific decisions in pluggable resource manager components, while the policy may be written to be portable across some range of platforms. However, activator implementers would need to know how objects are used by activations and may also choose not to call the *addObject* method for a given resource consumer if it knows that it would not be used.

Implementation and experimental results

To prove the feasibility of the concepts and abstractions (using resource managers with policies) described here, we did an experiment. The Linux operating system offer advanced management of the scheduling of packets going in or out of network interfaces. We use this to reserve bandwidth for outbound traffic, which in our case let us test the idea of reserving network bandwidth for individual streams without relying on an *IntServ* or a *DiffServ* architecture. Proper reservations, shaping and policing at each endpoint, possibly combined with SLAs and a *DiffServ* architecture, could provide reasonable network QoS for some cases.

We establish a scheduling class hierarchy based on the HTB (Hierarchical Token bucket) scheduling discipline available in the standard Linux 2.6 kernel [Devara02]. Figure 5.18 shows our scheduler hierarchy[24]. The total bandwidth (2 Mbit/s in this case) available at the root is shared between two classes: The first class reserves all available bandwidth to our middleware resource manager. Packets are placed in the *best effort class* by default, which gets bandwidth not used by the other class (possibly, a minimum amount can be reserved for it to prevent starvation). The

[24]A similar result could also be achieved using a composition of priority queueing dicipline and traffic shapers based on the token bucket model.

resource manager can, as a response to reservation requests, dynamically add subclasses to its class, reserving a fraction of its total available bandwidth. It keeps track of the sum of such reservations to prevent overbooking, in order to perform admission testing and to report bandwidth left available (through the inspector interface).

In our implementation, the *session* object is used as a resource consumer object (see above), The session object (see also section 5.3.4) contains information about the network connection (IP and port of client) and may carry a range of session specific information from the various parts of the layer stack. Just using the session seems to be sufficient for some cases. In general, the implementation of resource managers may need to make assumptions on how middleware abstractions are implemented or mapped to operating system abstractions in addition to how the operating system schedules resources. For instance, there is not an implied one-to-one mapping between Java threads and O.S. kernel threads, and information about such mappings is not available from the Java virtual machine in a standard way.

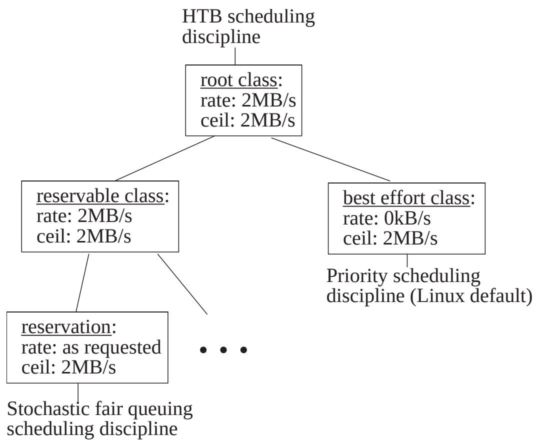
HTB scheduling discipline
root class: rate: 2MB/s ceil: 2MB/s
reservable class: rate: 2MB/s ceil: 2MB/s
best effort class: rate: 0kB/s ceil: 2MB/s
Priority scheduling discipline (Linux default)
reservation: rate: as requested ceil: 2MB/s
Stochastic fair queuing scheduling discipline

Figure 5.18. Scheduling hierarchy for network output

Our reservation domain has a method addSession (instead of the more generic addObject method), which takes a session object, extracts the destination IP address and port of the binding and uses this to add a packet filter to the kernel which places all packets with that destination in the corresponding scheduling class.

Figure 5.19 shows the result of measuring end-to-end invocation delay with and without reservation. We performed 200 invocations which read data blocks of 40 kbytes each, periodically. The server was connected to the internet via a 2MB/s line. We placed an additional 90% load on that line (by continuously requesting large images from a web server), and did the experiment twice: First without any reservation (dotted line) and thereafter with a policy that reserved 200 KB/s of bandwidth to the binding (solid line). The graph confirms that under such heavy load where queuing of packets on network interfaces is significant, bandwidth reservation can effectively bound invocation delay.

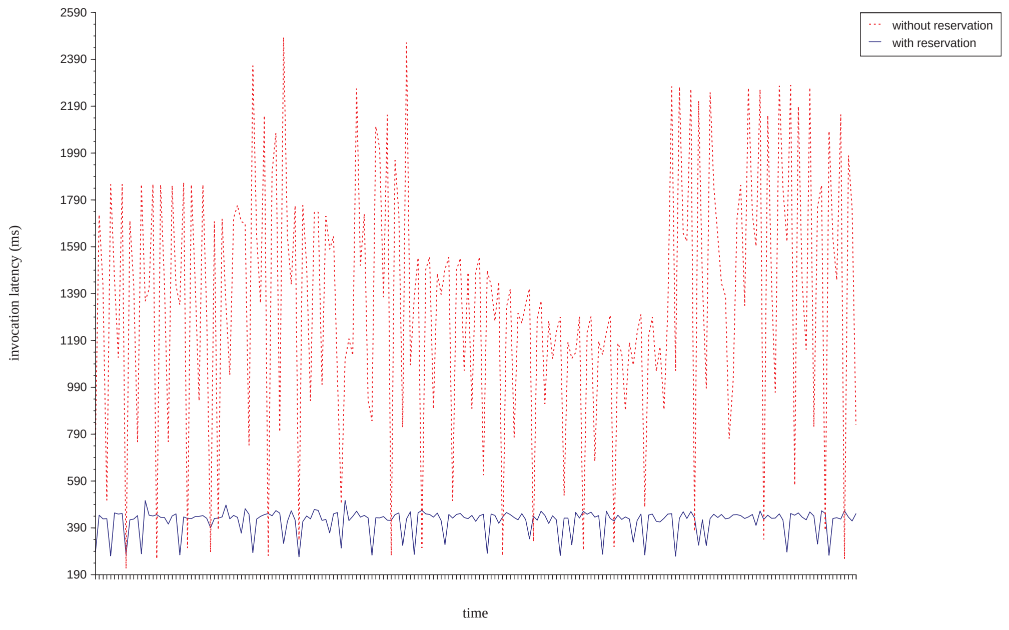

Figure 5.19. Invocation delay with and without bandwidth reservation

5.5.2. Alternative binding types

In our experimental work, we have mostly focused on the client/server architecture and bindings which enable remote method invocations from one client to one server. However, we claim that the proposed architecture is applicable to other binding types as well. The concept of binding types is defined and explored in [Parlav03] where the ideas are validated by applying them to a set of different binding types: The RMI binding type, a binding type representing the publish/subscribe pattern, and an auction binding type.

In this section we investigate how we can support the publish/subscribe binding type. Here, we have two main roles of participants: *Publishers,* which post asynchronous events (or notifications) and *subscribers* which receive events. Publish/subscribe has a relationship with streaming (e.g. media streaming), and can have a wide range of QoS issues, architectures and implementation options. It is more than the observer pattern [Gamma95]. The main strength is the full *decoupling* in time, space and synchronisation. Much research is being performed in the area, and we refer to [Eugster03] for a survey

The figure below illustrates the overall idea of a publish/subscribe service. Publishers and subscribers can also be viewed as clients to a publish/subscribe service. Subscribers invoke it to request subscriptions to event streams, possibly with requirements with respect to content filtering. Similarly, publishers post events.

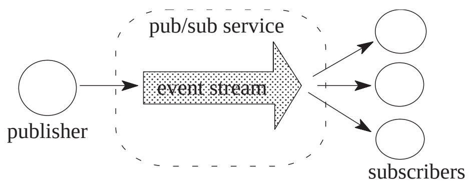

Figure 5.20. Publish/subscribe service

Publish/subscribe as a binding type

Publish/subscribe (pub/sub) interaction can be viewed as a *binding type* that allows multiple participants, each playing the role of either *publisher* or *subscriber*. Different instances of this binding type would represent different event streams, and different QoS properties could be negotiated for each of them. This does not exclude the possibility of negotiating content based filtering or QoS local to individual subscribers. It is useful to also model publishers and subscribers as *clients* to an abstract pub/sub service. Subscribers would use this service to request subscriptions to event streams, possibly with QoS requirements. Similarly, publishers would request interfaces to post events. Realisations of the pub/sub binding type may use the mechanisms of the simpler RMI binding type. Figure 5.21 illustrates how we can model a pub/sub service in the ODP computational viewpoint. The interfaces are as follows:

- The *notify* interface, offering a *notify* method, to deliver events.
- The *pub/sub* interface, offering methods to *publish* (request a notify interface), and to *subscribe* (give the service a notify interface for callback).

Subscribers may want to implement the *notify* interface[25], and publishers may want to invoke it. The pub/sub service would implement *notify* interfaces on behalf of subscribers and invoke the notify interfaces on behalf of publishers. The interfaces could be implemented in different ways, depending on requirements, architecture, resource availability, etc.

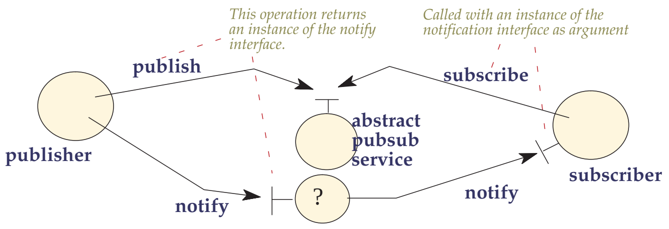
This operation returns an instance of the notify interface.
Called with an instance of the notification interface as argument
publish
subscribe
abstract pubsub service
publisher
subscriber
notify
?
notify

Figure 5.22. Computational model

Engineering of binding infrastructure

It is relatively straightforward to engineer this model with our framework. A generator would *deploy* an abstract pub/sub service by generating a binding proxy and a interface reference for it (see figure 5.23). The setup is rather similar to the simpler client/server case discussed earlier in this chapter. However, there will be no base-implementation of the pub/sub service or the event-stream at this point. This implementation would be completely negotiable at the time of *activating* the binding.

A binding protocol should be supported, for instance as a RMI interface. Publishers and subscribers invoke operations on that interface to join or leave the binding. These operations are implemented by the negotiator object, which may consult a policy trader to find a policy. Information necessary for activation (activator, network addresses etc.) would be returned to the participants.

Given an active binding, the publisher would call a local notify interface proxy to post notifications. Subscribers would implement the same interface. How events from publishers are conveyed to subscribers is however transparent and completely up to the policy.

[25] C.f. the Observer pattern [Gamma95]. An observer is an interface for notifications or events.

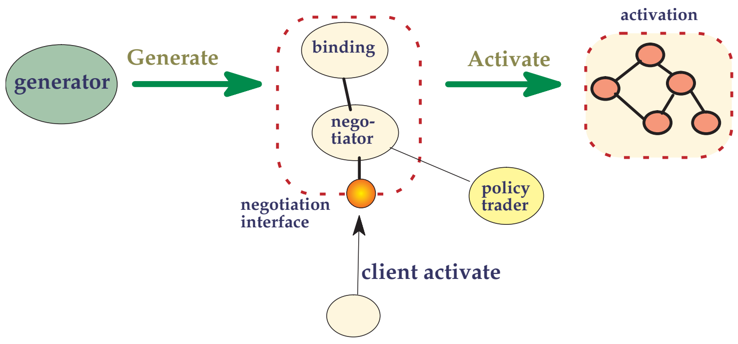

Figure 5.23. Server side binding

Figure 5.24 illustrates the client side (publisher or subscriber). Like in the simpler RMI case, a resolver would set up a binding. This is associated with a negotiator object that is able to contact the negotiation interface of the server. This negotiation may result in activation on both sides. Activations provide implementations of both the pub/sub service interface and the event stream. It should be transparent for the application code if the interfaces are remotely or locally implemented. The FlexiBind framework lets us bind to a service before it is decided how to implement it. Note that we may need to install more than one generator and resolver. In addition to one generator/resolver set for the abstract pub/sub interface, we would need one extra generator for *deploying* notify interfaces at subscribers and one extra resolver at publishers for binding to notify interfaces.

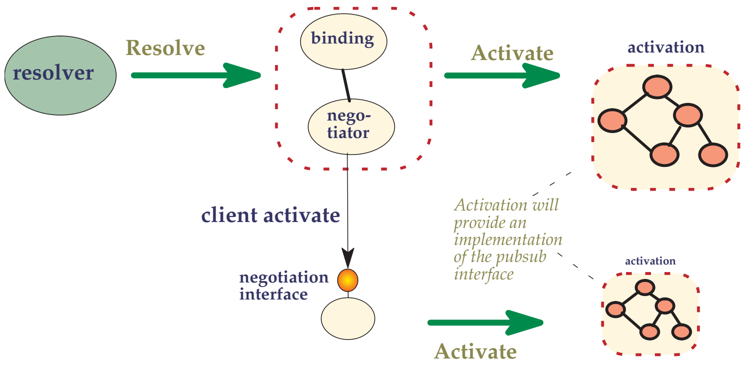

Figure 5.24. Client side binding

Implementation

We implemented a prototype framework for pub/sub bindings as described above, and we tested it with some example policies to validate that our concepts are feasible. The implementation includes a *generator* for the abstract pub/sub service as well as a *resolver* for publishers and subscribers. A negotiator pair realises a simple negotiation protocol (see section 5.4.1). For testing, we provide a simple implementation of a pub/sub interface and an associated *notify* interface. It mainly keeps track of subscribers and copies incoming notifications to notifications on subscribers. This implementation can (by policies) be instantiated in different configurations:

- A simple policy is to install the pub/sub service implementation on a *separate server* and let publishers and subscribers interact with it via RMI. The server activator would instantiate the service and update the binding between the identifier-part of the interface reference and the implementation target. Publisher and subscriber activators can be empty (except for what is needed for basic RMI interaction).

- An alternative is to install the service implementation in the publisher capsule, where the publisher application invokes it locally. Subscribers invoke it remotely. The publisher activator instantiates the service implementations and updates binding between identifier and implementation. The subscriber activator must update the target address in the binding proxy (must be conveyed from the server during negotiation). Obviously, this policy limits the number of publishers to at most one.
- Similarly, we could place the implementation on the subscriber side, which is not very practical since there can then be at most one subscriber.

Discussion

This demonstration shows how pub/sub bindings can be made highly negotiable. We may locate service implementations at publishers, subscribers, servers or combinations. Instead of just duplicating and forwarding RMI invocations, we could use more advanced event stream implementations instead. For example, activations may be based on multicast protocols.

An important issue with pub/sub bindings and multi-party bindings in general is that all participants are not necessarily in the binding at the same time; they should be allowed to come and go over time, without disturbing the operation of other participants. This means that there may be negotiations and resulting policies involving only subsets of participants. Even if we may want to (conceptually) model a multi-party binding as having one single contract, the realisation of a given negotiation should be allowed to be limited to as few participants as necessary. This is obviously important for scalability. In the architecture described here, the arrival of a participant means initiating negotiation with just that participant.

When a participant joins a binding having existing activations, it should (if possible) use a policy consistent with the rest of the binding, both in the contractual sense, and in the sense that the different parts of implementation can interoperate and work correctly. If that is not possible, we may need to change other parts of the binding in order to accommodate the new participant. This raises issues of adaptation and metapolicies. In any case, it is necessary to manage the different parts of binding-implementation. We must for instance ensure that shared state is not accidentally duplicated. There are two alternative ways of joining existing bindings that have shared implementations:

- To define alternative policies. One policy implementation has an extra piece of code which instantiates the shared implementation (and shared state). The other has code to hook into existing implementations instead. Service profiles must state assumptions on existing implementation components to ensure that the right alternative is selected. In our experiments (see also section 6.2), a special *inspector* component (the *channel manager*) reports the existence of event channels (and their QoS properties) which could be re-used by joining participants. The disadvantage of this approach is that we may need to register a number of policies having the same obligations. The advantage is that activator implementations are simpler.
- To use the same policy for both cases, but the activator implementation can detect if it needs to install shared implementations or just connect to one. In our experiments a server activator checks if a pub/sub implementation exists on a predefined name (see section 5.4.4), and instantiates one if not. This approach has the advantage of reducing the number of policies but implementations may be more specialised and may thus be less reusable.

5.6. Concluding remarks

In this chapter, we describe our middleware framework and how it supports dynamic binding configuration by using pluggable and replaceable policies. The distinction between *bindings* and their *activations* represent an extra level of indirection to decouple policy and application programming which may simplify how we understand adaptive bindings. Consequently, we identify two types of pluggable policy components: (1) *Binders,* which represent protocols and metapolicy and (2) *activators,* which dictate how protocol stacks and associated resources would be configured. An activation would essentially be a configuration of layers which represent aspect implementations. This is an untyped approach to reflective interception of invocations in the sense that layers conform to an interface containing a generic invocation method. Bindings are represented by binding-proxies which are attached to an activator (when active) and (possibly) a metapolicy object which defines how activators are selected.

Negotiation is performed by negotiator metaobjects which can be attached to bindings. Negotiation mainly involves collecting environmental properties from the platform, requirements from the applications and that binding participants exchange information with each other to reach a decision on which activator to select for a given binding. It would be up to the metapolicy (binders, negotiators) how system information is reflected as environment descriptors. Each binding instance would be associated with a dynamic profile; i.e. a profile expression with placeholders for parts to be determined by querying at negotiation time. Such querying is done on *inspector* objects which map from platform dependent characteristics to the more abstract profile model. This means that QoS mapping is highly configurable and set up or modified by binder components. This scheme has the advantage of being flexible but requires some conventions for naming of inspectors.

Middleware frameworks should abstract over rather different resource models in operating systems. A simple model is sketched: resource managers, reservation domains, and resource consumers. We demonstrate how this approach can be used for bandwidth reservation to bound invocation delay. Our approach seems to support some level of separation of concerns between activators and resource reservation requirements, which is good for activator code reusability.

Though our framework design has been mostly focused on client/server RMI interactions, we claim it can be used to realise other binding types as well. This can be validated by investigating how alternative binding types, in particular those involving more than two participants. We did this for publish/subscribe bindings. By designing and implementing a framework for this binding type (resolvers, generators, simple negotiators and some simple test policies), we produced some evidence that it is feasible to realise negotiable pub/sub bindings using policy trading and our middleware framework. However, this experiment did not focus on how to design the policies and metapolicies beyond simple negotiation. The design space is so large that we should investigate a particular application domain as well. We do that in the next chapter.

Chapter 6.
Application case studies

In this chapter we apply the policy trading scheme and profile model to two application scenarios in order to test the feasibility of our approach. In section 6.1, we investigate how web applications can exploit policy trading to automatically accommodate varying conditions and requirements. A combination of user requirements and characterisations of the client browser and the web-server (including e.g. display size or server load), lead to the selection among various pre-defined configurations of server components (policies). This experiment indicates for instance that profile expressions may be more compact and composable than e.g. CC/PP profiles [W3C-04a] and that there may be a need for extensions to the core profile model to better support interoperability.

In section 6.2, we investigate how to realise a publish/subscribe binding type and how to make their implementations negotiable. We focus on how to provide a high level of negotiability through partitioning an event stream into a set of event-channels (possibly realised by multicast IP) that subscribers can select from. With this approach, it is for instance possible to stream video to multiple subscribers having different requirements. Here, the main interest is in how we exploit policy trading to manage event-channels.

In section 6.3 we do some performance measurements on some of the policy matchings used in the two cases in this chapter.

6.1. Web application case

The increasing use of web technology for public services, users with various needs, mobility, browsers with varying capabilities, and the use of portable devices, motivate web applications which are able to automatically adapt their behaviour or presentation. A current approach is to promote the separation of concerns between content and presentation aspects. The use of cascading style sheets (CSS) and scripting can help in adaptation of web content on the client side. Some earlier research, propose techniques for filtering content in proxy servers, for instance to better fit into smaller screens [Bickmore97], or to give better response times in low bandwidth situations [Fox96].

In some cases, it may be more efficient to adapt the behaviour and presentation on the server side as well. This may also enable a wider range of adaptations, and both client and server properties may be taken into consideration. This implies negotiation in the sense that client sends its preferences and description of browser capabilities to the server. The server uses these to make decisions on how to adapt the content. W3C standards (CC/PP) for user agent characterisation exist [W3C-04a], which illustrates that this is not a new idea. However, the current standard has some problems and has not yet reached a wide acceptance. For instance, it is complicated to implement and has weak support for composition of profiles (see section 6.1.5).

Examples of issues which could be negotiated and handled by policies include:

- Scaling of images or other media content on servers, e.g. resizing or reducing quality, to save bandwidth. There is a trade-off between media-quality and response time when bandwidth is limited.
- Converting formats (or scale) of media content to be compatible with client capabilities, such as supported formats, display resolutions etc.
- Reorganising of layout and behaviour due to different display sizes. For small displays, one may use more pages/requests with less information on each, and unnecessary illustrations may be removed.
- Skipping unnecessary navigational information and offering short-cuts and advanced options for experienced users.
- Adapt behaviour (degrade quality of service) due to server overload. This can be combined with admission control on client sessions as discussed in [Cherka02].

6.1.1. Architecture

We focus on how to decide on adaptations mainly on servers. This does not exclude the possibility of adapting web-applications on clients or proxy-servers as well[26]. Figure 6.1 illustrates how we may exploit policy trading in a web server context. Negotiation can happen at each request or when initiating a client session[27]. Negotiation results in a decision on how to present the content components (software components, document fragments) and (possibly) how to configure implementation resources, backend-services etc. We may associate a server or a server application with a set of policies, each which dictates a possible configuration.

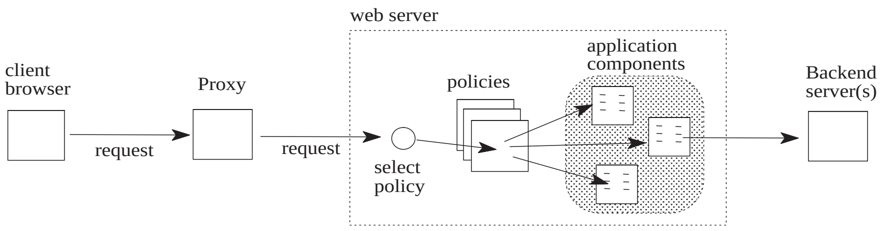

Figure 6.1. Web server architectural model

[26]Client scripts can for instance adapt by replacing stylesheets or programmatically manipulating the abstract syntax tree of the document or other properties of the browser.

[27]Though the HTTP protocol is stateless, it is often desirable to establish client sessions, in order to maintain application specific state shared between subsequent requests. Sessions are typically implemented by using cookies. A session can be started when a user logs in or (more transparently) at the first request (use timeout to end sessions).

Exchange and composition of profile expressions

At each request, or at the start of each client session, the browser would need to pass two profile expressions over to the server: One which represents user requirements (resulting from user preference choices) and one characterising the browser's capabilities and possibly other environmental properties like network QoS. These can for instance be conveyed by using HTTP request headers (in an extended version of the HTTP protocol), or by using request parameters. The easiest way to do that, is to add the expressions as extra (hidden) parameters in a HTML form (e.g. login). This approach is limited to session negotiation and is not completely transparent, but it is sufficient for testing our ideas.

The profile expressions can be composed as discussed in chapter 3 and 4; i.e. the proxy and the server may add their own environment descriptors by using the component sum operator.

Implementation

We use the implementation of a profile model compiler, a conformance checker (described in chapter 4) and a policy trader demonstrator to evaluate the example described in this section. In addition, based on the Java Servlet architecture [JSR154] and on the experiences from reflective middleware (chapter 5), we are able to compose and configure web content components and components which can intercept requests and responses, in order to modify the behaviour of the application. In [WComp] we develop a component framework and a scripting language for such composition. An interceptor for HTTP requests can collect profile expressions coming from clients, and (possibly) consult a policy trader to select a policy which is then invoked. A policy would denote a particular configuration of components; it may be represented by a script fragment, and it may present an interface to be called to serve the request. Different policies may for instance share application components, but do the composition differently.

6.1.2. User profile expressions

User profile expressions represent application or user requirements (or preferences) which should be understood as constraints (or offers to satisfy constraints). If a profile model for the web application domain is defined, such expressions could be generated by a web browser (based on user preference selections). Sometimes, it can be useful to extend a generic web browsing profile model with more application specific parts (for example for e-commerce or telemedicine applications). In that case, the user preference selections and negotiations could be better supported by the application specific forms and/or scripts. Examples of profile-names include:

- *'Inexperienced'* - Suited for inexperienced users. This may be interpreted as if generated content should contain more explanations, or simplify certain choices/questions.
- *"Advanced"* - Give access to advanced features (for advanced/experienced users).
- *"Frame"* - Navigational information, headings and footings or similar should be removed, such that the content is suitable for presentation in a frame. A subprofile of this cold be *"Frame-Noimg"*. This means that images should be removed from the content as well.
- *"Accessible"* - The content should be strictly conformant with the W3C accessibility guidelines, such that it is possible for browsers to extract the essential information. Note that *"Frame"* also may serve such a purpose and may be subprofile of *"Accessible"*.

- "HighQuality" - Client requires high quality images (or possibly other aspects). In our example, this means that images should be high resolution and high colour which could be particularly useful in a scientific/medical image browsing/rendering scenario. It could be up to the application to put a specific meaning into this.
- "QuickResponse" - The user should not have to wait too long for responses. This could mean that most responses come within a second and/or that the average round trip time is better than e.g. 0.5 s.

6.1.3. Service profile expressions

Environment descriptors from clients would mostly denote browser capabilities. A typical issue is the display size[28], the actual window size or other display characteristics. A related issue is image or multimedia viewing capabilities[29]. A browser environment descriptor could also include other relevant characteristics like network QoS or the presence of special interfaces, protocols or the existence and properties of plugins. *Server side* environment descriptors could for instance describe resources (including availability of certain software components) used to convert/re-scale multimedia content.

Other components like proxies between the client and the server may as well add sub-expressions to the environment descriptor, for instance the capability to transform or re-scale media content to different formats, size, colour etc. Such components may compute the environment descriptor string dynamically, by looking at the supported media types and formats from the client and including the types/formats which it is able to transform. A simple example of this is policy #11 of table 6.1 where the proxy announces its ability to convert HTML into WML by adding an expression which satisfies *"(Proxy + WML-Filter)"*.

Media support in browser

Browsers may have different capabilities with respect to displaying different media formats (e.g. images). If this is reflected in a browser service profile, a server could decide to convert or omit images if the format is unsupported by a browser. Figure 6.2 below shows a possible profile graphs for some typical image formats.

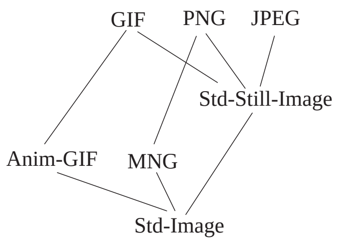

Figure 6.2. Image format support profiles

[28]The current HTTP standard allows a request header for conveying the client display size but this is seldom used in practise.

[29]The current HTTP standard defines a request header containing a list of supported Mime types.

In our model, we can define subprofiles which represent certain combinations, thus allowing simpler expressions for common situations. For instance, most browsers could simply say "*Std-Image*" instead of listing all supported Mime types. Note that if we have established that *"Std-Still-Image"* satisfies *"GIF+PNG+JPEG"*, the model does not say that the opposite is true as well. If the browser states: *"GIF+PNG+JPEG"* while the service profile of a policy states "*Std-Still-Image",* we may want these to match. Unfortunately, the core profile model does not allow sums in conformance rules (see also the next paragraph). The rules are:

```
Std-Image ≤ Anim-Gif ≤ GIF                                    (rule 1)
Std-Image ≤ MNG ≤ PNG                                         (rule 2)
Std-Image ≤ Std-Still-Image ≤ PNG                             (rule 3)
Std-Still-Image ≤ JPEG                                        (rule 4)
```

Display properties

One could also express if the display is capable of colour. We may use subprofiles to reflect that a display which is capable of displaying full colour would also satisfy requirement for greytones or monochrome, etc. It is also straightforward to express display resolution as parametrised profiles. Note that we want to describe the size actually available for the application, which often would be the window size rather than the display size. The rules are:

$\texttt{Colour} \le \texttt{GreyTones} \le \texttt{Monochrome};$ (rule 5)

$\texttt{Res}[x_1, y_1] \le \texttt{Res}[x_2, y_2]$, **if** $x_1 \ge x_2$ **and** $y_1 \ge y_2$; (rule 6)

It is more flexible but somewhat more space consuming to define separate profiles for each dimension: E.g., *XRes[x]* and *YRes[y]*. This illustrates how our model can address the problem that different communities may use different notions for the same phenomenon (see also section 4.5). Defining a proper set of mapping rules allows the trader to resolve the differences automatically. For instance, one may use a profile definition using both dimensions, while another may use separate profiles for each dimension. We may define rules for that:

$\texttt{Res}[x_1, y_1] \le \texttt{XRes}[x_2]$, **if** $x_1 \ge x_2$; (rule 7a)

$\texttt{Res}[x_1, y_1] \le \texttt{YRes}[y_2]$, **if** $y_1 \ge y_2$; (rule 7b)

With these rules, the expression *"Res[100,100] ≤ XRes[100] + YRes[100]"* will evaluate to true. However we may also want *"XRes[100] + YRes[100] ≤ Res[100,100]"* to be true as well, which is not the case. This means that it would have been useful if our profile model could be extended to allow rules to be defined for sums:

$\texttt{XRes}[x_1] + \texttt{YRes}[y_1] \le \texttt{Res}[x_2, y_2]$, **if** $x_1 \ge x_2$ **and** $y_1 \ge y_2$; (rule 7c)

Network QoS

Network QoS characteristics are typically latency and bandwidth, and it is straightforward to define rules for those. However, profiles would need to be used in some context to define their meanings more completely, for instance if a latency is estimated or guaranteed. One could define completely different profile names with similar rules, but it may be better to define different contexts as separate profile names. In this example, we define the following basic profiles representing network entity descriptors. They are meant to be used with a latency and/or a throughput basic profile:

- *"NetConnection"*: Characteristics of the connection to the internet. We cannot really deduce the end-to-end QoS from this kind of description. However, it can give some indication[30] of what can be expected under normal load and it can be provided without knowing the address of the opposite end.
- "*NetEstimated*": The QoS is estimated end-to-end. It could be estimated by measuring the round trip delay or the actual throughput. One could also deduce an estimated QoS from knowledge of the path to the other end or the type of network in between, for instance if both ends are on the same LAN. *"NetEstimated"* could be defined to be a subprofile of *"NetConnection"*. Observe that while latency would be partly meaningless for *"NetConnection"*, it would make sense for *"NetEstimated"*.
- "*NetGuaranteed*": The QoS is pre-negotiated or reserved end-to-end (it represents certain guarantees). This would be a subprofile of *"NetEstimated"*.

For this example we simplify somewhat and define three classes of latency and bandwidth respectively. This should be sufficient for many web applications. The network related rules are:

```
NetGuaranteed ≤ NetEstimated ≤ NetConnection ;                (rule 8)
VeryLowLatency ≤  LowLatency ≤ NormalLatency ;                (rule 9)
VeryHighBW ≤ HighBW ≤ NormalBW ;                              (rule 10)
```

In this example, *"Normal"* may mean what normally can be expected for internet connections (for instance max 1s latency or min 100Kb/s bandwidth). *"High"* may mean QoS normally only found in LANs or shorter paths (e.g. max 0.50ms latency or min 5Mb/s bandwidth), *"VeryHigh"* could mean quality found only in dedicated networks or LANs (max 1ms latency or min 100Mb/s bandwidth). This example illustrates that a profile name may be given different interpretations (what they mean in terms of measurable characteristics) in different contexts[31], i.e. it may make sense to interpret for instance the bandwidth profile *"HighBW"* differently in the context of *"NetConnection"* than in the context of *"NetNegotiable"*. Also observe that such abstract profiles allow for changing interpretations over time as technology changes without changing the profile models.

Other properties

It could be useful (when related to sessions rather than single requests) if we could report and use dynamic properties like server load. This could be reported by some application-specific inspector component (section 5.4.3). It is of course not trivial how to use such measurements in general to guarantee QoS while being resource efficient. A simpler approach is that certain policies require that a system is moderately loaded or better in order to reduce the probability of overloading the system.

```
LowLoad ≤ ModerateLoad ≤ HighLoad ;                             (rule 11)
```

[30]In some cases, it can be useful just to able to distinguish between users with a normal broadband internet connection and those using a slow and expensive mobile phone link.

[31]It is outside the scope of our model to define or manage such meanings, but contractual QoS languages like CQML seem to be suitable for that.

Expression examples:

A browser which can only display still images, except for animated PNG, and 200x100 display:

```
Std-Still-Image + MNG + XRes[200] + YRes[100]
```

A browser like in the first example, but with a high bandwidth network connection. If the server is detected to be in the same high quality LAN this will lead to an estimated QoS:

```
(Std-Image + Res[200, 100]) ⊕ (NetConnection + VeryHighBW)
  ⊕ (NetEstimated + VeryHighBW + VeryLowLatency)
```

To characterise a device which is capable of 200x100 display in colour mode *or* 400x800 in monochrome mode an environment descriptor can be like the following. This corresponds to *parallel-or* composition (section 4.5.4). It can be hard to express such composition in CC/PP (section 6.1.5).

```
(Res[200, 100] + Colour) ⊕ (Res[400, 800] + Mono)
```

It is possible to use component-sums to express more than one type of capability in the same expression, for example that the *connection* has certain properties and that the end-to-end QoS is estimated (statistically from round-trip measurements) to be of a certain level.

```
(NetConnection + ModerateBW) ⊕ (NetEstimated + LowLatency )
```

6.1.4. Negotiation scenarios

In this section, we sketch a set of example policies and how a trader may select from these, based on a request from a client.

Policy examples

Table 6.1 below shows some examples of policies. We assume that the resulting web pages contain text, images and typically some navigational information.

#	Policy description	User profile	Service profile
1	"Standard configuration": normal size images, full colours, navigation menus.		HTML + Res[800, 400] + Std-Image + Colour
2	"High quality configuration": high resolution and high quality images (e.g. for medical/scientific use). Higher level of configurability.	Experienced + HiQuality	(HTML + Res[1200, 900] + Std-Image + Colour) ⊕ (NetConnection + NormalBW)
3	Scale down images (to JPEG format), remove unnecessary parts of lay-out.		HTML + Res[600, 300] + JPEG + Colour
4	Convert all images to medium quality JPEG (high level of compression)		HTML + Res[800, 400] + JPEG + Colour
5	Same as P#4, but with a certain level of network and server QoS, it can satisfy a requirement for quick response times.	QuickResponse	(HTML + Res[800, 400] + JPEG + Colour) ⊕ (NetEstimated + NormalBW + LowLatency) ⊕ (Server + ModerateLoad)
6	Like P#4 but for smaller display. Images are not scaled down but the removal of explanations and parts of navigation structure will free some space which may leave some more space to other parts of the layout.	Experienced	HTML + Res[400, 200] + JPEG + Colour
7	Like P#1 but skip explanations, simplify navigation structure, add some advanced option to the menu.	Experienced	HTML + Res[800, 400] + Std-Still-Image + Colour
8	Show only text part (and possibly images in the text).	Frame	HTML + Res[800, 400] + Std-Still-Image + Colour
9	Show only text part.	Frame-noimg	HTML

#	Policy description	User profile	Service profile
10	Suitable for very small displays, remove most images, remove (or minimise) navigation part of page and place them in separate pages.		HTML + Res[200, 100] + JPEG + Colour
11	Same implementation as P#6 on the server, but in addition tell proxy to do translate into WML.		(WML + Res[200, 100] + Std-Still-Image + Colour) ⊕ (Proxy + WML-Filter)

Table 6.1. Policy examples

Negotiation examples

Example 1: The client is a PDA connected via a wireless LAN having a 500x250 pixel display, and which supports HTML and still images. The corresponding environment descriptor is

$$E_c = \mathit{(HTML + Std\text{-}Still\text{-}Image + Res[500, 250] + Colour)} \oplus \\ \mathit{(NetConnection + NormalBW + NormalLatency)}$$

This would satisfy P#6, P#9 and P#10. P#6 should probably be preferred since it better utilises the display (more information in one page, fewer clicks to navigate). Note that the list of policies is ordered such that the trader would select it first.

Example 2: The client requires *QuickResponse* and has a 1000x800 pixel display, and it supports the standard set of image formats. The client is close to the server, such that the network QoS would be estimated to be very high bandwidth and low latency. The client sends the request to the server which has a low load at the moment of negotiation. The client and server environment descriptors are combined: $E = E_C \oplus (\text{'Server'} + E_S)$. The resulting expressions would match P#5.

$$R = \mathit{QuickResponse}$$

$$E_c = \mathit{(HTML + Std\text{-}Image + Res[1000, 800] + Colour)} \oplus \\ \mathit{(NetEstimated + VeryHighBW + VeryLowLatency)}$$

$$E_s = \mathit{LowLoad}$$

Example 3: If R is set to empty in example 2, the request would match all policies, except P#2 and P#11. A policy trader could for instance select the first found, i.e. P#1, and also here, it is necessary to order the set of policies according to what we want to be preferred. However, different orderings may be suitable for different clients. For instance, if resource economy is important, we may prefer P#1. If user-QoS is more important, we may prefer P#5 instead. This example illustrates that though a single ordering may be satisfactory in many cases, there may be cases where the ordering is dependent on the client.

6.1.5. Related standardisation

CC/PP (composite capabilities/preference profiles) is a W3C standard [W3C-04a], defining abstract syntax and transfer syntax for exchange of client device characteristics and user requirements, to support context aware or adaptable web-applications. CC/PP is based on RDF [W3C-04b] which constitutes a metalanguage (with at least one corresponding XML encoding) in which CC/PP profiles can be defined. In CC/PP, profiles are essentially name/value pairs. A profile is structured as a set of *components*, each of which can be described by a set of attributes. The types for attribute values can be simple or complex; i.e. they can be declared as sets or sequences as well as atomic types.

A CC/PP *vocabulary* is a defined as a set of allowed names and types for attributes plus their associated meanings. Vocabularies are meant to be application- or application domain specific. However, the W3C recognise the interoperability problem (section 4.5.3). Vocabularies correspond in some sense to concrete profile models. However, a CC/PP vocabulary does not formally define semantics to facilitate conformance checking or profile validation. Semantics may be defined *informally* within vocabularies, but this is not required or encouraged by the standard. In essence CC/PP leaves the interpretation of the values used to the processing application.

Figure 6.3 below shows a simplified picture of a CC/PP profile corresponding somewhat to the environment descriptor in example 1 (A full representation of the CC/PP profile would be too big and complicated to be illustrative here). We assume that we are allowed to define our own vocabulary, but our choice of component types is similar to UaProf[32]. Note that we can use the collection type (BAG) to represent the supported image formats and languages. It is possible to use references to e.g. service-level names, but the definition of such names would be outside the scope of CC/PP.

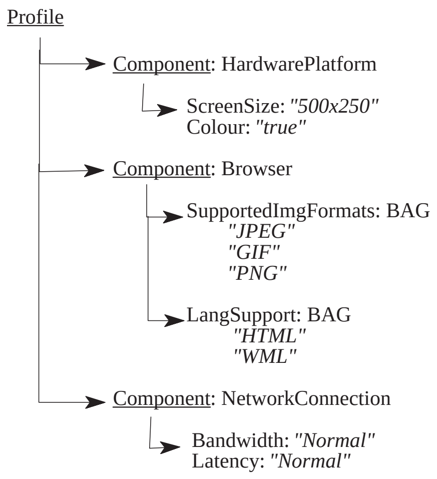

Figure 6.3. CC/PP example (simplified)

Matching cannot be facilitated by just associating profiles with policies. In general, it is necessary to specify rules for how parameter values are matched with policies.

This illustration shows how our model can simplify the expression of capability and preference expressions to be exchanged between negotiation partners. A core model defining conformance matching rules, along with concrete defining conformance rules, contributes to this. CC/PP is on its side criticised (cf. e.g. [Butler02]) for being complex, but also for its limited expressiveness and potential interoperability problems[33]. It is difficult to combine profiles from independent actors, in particular if those profiles describe the same types of components (for instance, it is not clear how to merge two profiles each with a hardware platform component). This is also one of the reasons why it is hard to express certain types of composition.

[32]The UaProf [OMA03] also includes a software-platform (operating system) component type, but this is not needed in our exampe.

[33]The profile resolution in particular, may lead to interoperability problems since it is not standardised how this is done. Profile resolution is necessary since profiles may be split into fragments, possibly describing the same concepts.

6.1.6. Discussion

In this section, we investigate how policy trading and our profile model can be applied to web applications, such that they can be made adaptable to different user requirements, different browser capabilities, network connections and dynamic properties like server load. Browsers may add requirements and environment descriptors to each request or at the initiation of a session.

Policies may share implementations; i.e. in some cases, the only differences between policies are their profile expressions. For instance, the implementation of policy #4 above may satisfy a *"QuickResponse"* requirement if its environment satisfies a service profile with additional constraints on network QoS and server load. Another example is that low or expensive network bandwidth may be another reason (in addition to small browsing devices) to select policy #3 or #4 instead of policy #1. For simplicity, this is not reflected in table 6.1, but is possible.

We observe that more than one policy may match during a given negotiation, but only one of them should be selected. We do not focus on how to order the alternatives to maximise user satisfaction, cost, resource utilisation, or some combination. We adopt a simple (but somewhat ad. hoc.) approach which is to order the policies in the trader repository according to the policy designer's knowledge on what should usually be preferred. This is satisfactory in many applications but not necessarily all. As indicated in example 3 above, goals may be conflicting and different situations may require different orderings. Approaches like worth based negotiation or utility functions (see section 2.2.3), would not be easily adaptable to our profile mode since they are based on parameter values.

A related issue is that negotiation will fail if no matching policy is found (our model implies full satisfaction or failure). In many cases, there are acceptable alternatives. For instance, if a user requires *'Experienced'*, policies which cannot satisfy it would not be selected. It may however be acceptable for the user to select an alternative, if no such policies are found. A simple and effective solution is that clients can provide an ordered list of requirement expressions instead on a single one. This also illustrates that when designing profile models, we should keep in mind that an user-profile describes requirements to the policy rather than characteristics of the user. The profile '*Experienced*' illustrates this, and is problematic. For example, users *not* requiring *'Experienced'* can get policies for experienced users, which may not be what we intended. It may be better to use a profile '*Inexperienced*' which requires that the service should be suitable for inexperienced users.

We have also seen that declared conformance rules are limited to pairs of basic profile types. We may want to extend our profile definition language to allow rules involving a basic profile and a sum. Our model should be extended to accommodate this.

6.2. Multi-subscriber streaming

In section 5.5.2, we discussed how to support publish/subscribe bindings with our middleware architecture. Here, we have two main roles of participants: *Publishers* which post asynchronous events (or notifications) and *subscribers* which receive events. Publish/subscribe can be used for streaming (e.g. media streaming) and can have many QoS issues, architectures and implementation options. A publish/subscribe *binding type* could for example be used to stream video to multiple subscribers, each having different capabilities and requirements. The transport and filtering of events representing video frames could be done by negotiable components. In this section, we apply our policy trading approach to such a case where the main problem is to manage a set of shared event channels. By analysis and a proof-of-concept experiment, we find that the most obvious way of using policy trading is not generally feasible with multiple event-channels. However, the architecture can be extended to better support scalability. In particular, a two-level trading approach is investigated.

6.2.1. Introduction

In Eide et. al. [Eide03, Eide05], it is demonstrated how to exploit content based publish/subscribe networking to address the challenge of providing video streaming clients with fine grained selectivity along different quality dimensions, while maintaining efficiency with respect to resource consumption. This approach is claimed to allow a more dynamic and flexible streaming solution compared to more traditional approaches. A prototype is implemented as a proof of concept.

A video stream can be modelled a number of events representing different layers or aspects of the stream. Receivers can subscribe to subsets of the stream, and they are thereby allowed to independently trade-off between video quality characteristics and resource requirements. For example, it is possible to subscribe to a lower frame-rate than available from the publisher by filtering out the wanted frames. For scalability and resource efficiency, it is important to do most of the filtering close to the source, exploit multicast networking, and to share multicast channels when possible. Figure 6.4 illustrates how this may look like. A video server publishes events which are mapped to different multicast channels. Clients would select channels for subscriptions based on attributes of the events they carry. For example, client #2 subscribes to a reduced quality while client #2 subscribes to full quality. It can use the reduced quality channel (share it with client #1), but it needs an additional channel for the remaining notifications. The filtering and mapping to channels are based on attributes attached to each notification.

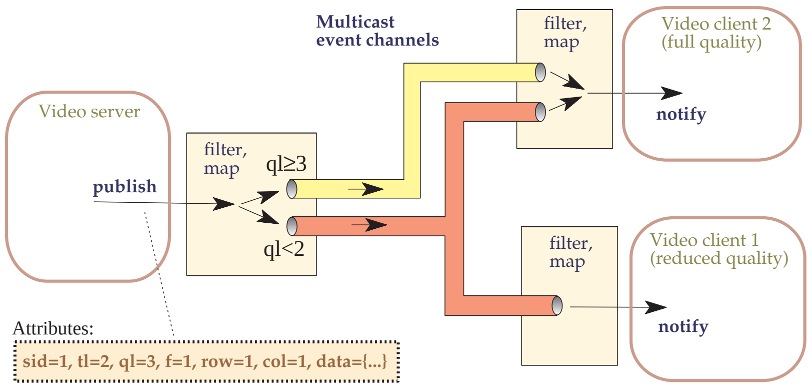
Multicast event channels
filter, map
Video client 2 (full quality)
notify
Video server
publish
filter, map
ql≥3
ql<2
filter, map
Video client 1 (reduced quality)
notify
Attributes:
sid=1, tl=2, ql=3, f=1, row=1, col=1, data={...}

Figure 6.4. Video subscription case

Dimensions and complexity of mapping

The video stream encoding scheme allows the stream to be split into "*layers*", according to a set of dimensions: time, quality, colour and picture-blocks (to select sub-areas of the frames). Each frame can be represented as a series of notifications; one per layer. The dimensions are as follows (we refer to [Eide03] for a detailed explanation):

As discussed in [Opyrchal], exploiting multicast in content-based publish/subscribe systems can in general be a rather complex problem. The number of possible event-channels may grow exponentially with the number of participants or dimensions (depending on the application). With *n* channels, there are 2^n-1 possible ways to subscribe to a selection of those channels. However, due to application specific semantics, only a subset of the possible subscriptions are meaningful. For example, we would not subscribe to quality layer 3 without also subscribing to layer 1 and 2. With the dimensions in table 6.2, and if we set x and y to 3, there can be up to 72 notifications per frame. If we add the time dimension, there are 4×72=288 different types of notifications, and we may need up to 288 event channels to support full selectivity without redundancy (some events are filtered out on the subscriber side, or some events are transmitted on more than one channel). There will be up to 25×4×4×2=800 different meaningful ways to subscribe if selected blocks are limited to *rectangular* sub-areas of the picture (without that constraint the number is higher). In practise, we would need fewer channels if channels could carry more than one event-type. Furthermore, we can use heuristics or manually defined mappings (to cover expected typical cases). This means that we may accept some redundancy.

Name	Range	Description
tl	0-3	Time dimension. Adding a higher layer corresponds to a doubling of the frame rate
ql	0-3	Quality dimension. Adding a higher layer increases the quality (signal/noise ratio).
f	0-1	Luminance (0) and chrominance layers (1) are separated. f>= 1 means full colour.
col	0-x	coloumns (we choose x = 3).
row	0-y	rows (we choose y = 3).

Table 6.2. Dimensions

6.2.2. Our approach

We now investigate how we can address a similar problem using our policy trading approach (chapter 3), profile model (chapter 4) and some of the concepts of chapter 5. Like in [Eide03], we assume that the application is run within a local area network. In our design, publishers and subscribers would need to contact a pub/sub service located at a server[34] in order to establish bindings. This server keeps track of the channels (using a channel manager), and uses a policy trader to pre-register and select suitable policies for each case. A policy would subscribe to existing channels and/or create new ones. Figure 6.5 illustrates this.

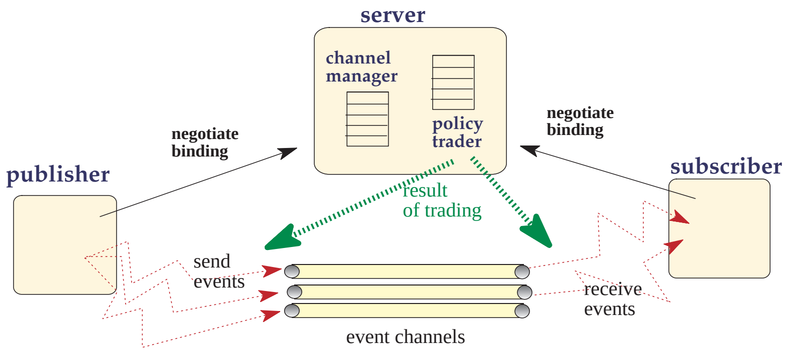

Figure 6.5. Publisher/server/subscriber architecture

Focus and assumptions

We make some assumptions and simplifications: First, we assume that there is at most one publisher per binding. In many cases, it is reasonable that different publishers require different bindings if the content differs and the subscribers would want to distinguish between them. Second, we do not support the row/coloumn dimension. Adding the row and coloumn dimensions (for

[34]This is strictly for the purpose of negotiation. The actual notifications may go directly between publishers and subscribers or via multicast channels (depending on the policy).

selection of a sub-area of interest) would be straightforward, if we assume that the areas are continuous. It can also be argued that sub-picture selection is not a QoS issue. Like [Eide03] we operate within a single LAN. We also assume that there exists a single server per LAN. We also focus on the QoS dimensions that are directly associated with events (like those shown above), though it is relevant to at least discuss how other dimensions can be added. For instance, network bandwidth, delay and jitter could be associated with multicast channels.

Characterising subscriptions and event-channels

Our profile expression model (chapter 4) can be used to characterise subscriptions and event channels. This is used in policy trading and in the channel manager. We first define a simplified model for the user level requirements (to characterise subscriptions). The profile type *'TQ'* has two parameters: an upper constraint of the time dimension and the quality dimensions of table 6.2. *'TQ[3,3]'* would for instance imply time and quality layers from 0 to 3. In addition the profile *'Colour'* indicates that the video stream carries colour information. The rule is:

$TQ[t_1, q_1] \leq TQ[t_2, q_2]$, **if** $t_1 >= t_2$ **AND** $q_1 >= q_2$; (rule 1)

For describing event channels, we define separate profile types for the time and the quality dimension, and we use two parameters for each: A lower and an upper limit. This is reasonable since an event channel may carry any sub-interval of the available layers. For user level subscriptions, we need only the upper limit (we use all layers up to a certain level). The profile name *'VChan'* is used to indicate that we describe event channels, *'Chrom'* denotes the presence of a chrominance (colour) layer and *'Lum'* denotes the presence of a luminance layer. The rules are:

$tl[x_1, y_1] \leq tl[x_2, y_2]$, **if** $x_1 >= x_2$ **AND** $y_1 >= y_2$; (rule 2)
$ql[x_1, y_2] \leq ql[x_2, y_2]$, **if** $x_1 >= x_2$ **AND** $y_1 >= y_2$; (rule 3)

Channel configurations

Now, it is an issue how to partition the event stream into *event channels* (preferably multicast), each carrying a subset of the events. We refer to a particular way of partitioning the stream into event-channels as a *channel configuration*[35]. Given such a configuration having *n* channels, there are 2^n-1 ways (in theory) to subscribe to (subsets of) those channels. In practise, only a subset of the possible subscriptions would be useful (as discussed above). If we focus on a subset of the dimensions introduced above; *tl*, *ql* and *f*, there are 4×4×2=16 different types of notifications. This is also the size of the largest reasonable channel configuration.

Based on assumptions about usage and the resource consumption of events, we may find acceptable configurations having fewer channels. For instance, let us assume that most clients would request at least *tl* and *ql* layer 0-2, and that some clients request *tl* and *ql* layer 3 in addition. Furthermore, assume that it is common to request video without colour (the degradation policy may be to trade off the colour layer first when there is lack of resources). Then the configuration shown in table 6.3 below is sufficient most of the time. It has four channels. Observe that there are dependencies between channels. For instance, we do not use channel 2 if not also subscribing to channel 1, etc. (the chrominance layer is not useful without the luminance layer and tl and ql layer 3 are not useful without layer 1-2). Profile expressions of table 6.3 are used to describe the

[35]Instance of channel config corresponds to *mapping specification* in [Eide03].

channels. In the following we will use such descriptions in different ways, for example as part of service profiles or environment descriptors.

#	Event channel description	Profile expression
1	Time layer up to 2, quality layer up to 2, luminance only (intensity information, no colours)	VChan + tl[0,2] + ql[0,2] + Lum
2	Time layer up to 2, quality layer up to 2, chrominance only (colour information)	VChan + tl[0,2] + ql[0,2] + Chrom
3	Time layer 3, quality layer 3, luminance.	VChan + tl[3,3] + ql[3,3] + Lum
4	Time layer 3, quality layer 3, chrominance.	Vchan + tl[3,3] + ql[3,3] + Chrom

Table 6.3. Channel examples

Channels of a configuration are not necessarily all active at a time in the sense that resources (including network addresses, sockets, etc.) are allocated to them. This may be more important with configurations having a high number of channels, and where it is not feasible to activate all at the same time. A related issue is how and when to activate (or de-activate) the channels. The simplest approach is to define and activate at the time of deploying a service. However, a given channel-configuration may have a high number of possible channels, and it may not be desirable (or even possible) to activate all at startup time (or at any time). We may want to activate and deactivate event-channels dynamically. We refer to this as *tactical channel management*.

In the following, we focus on tactical management. However, there may some cases where the channel-configuration itself should be adapted, for instance by splitting or merging channels. For instance, if there is just one subscriber, we may start with one channel (carrying all events) and split it later as more subscribers arrive. The process of coordinating such adaptation could be part of a global adaptation process governed by *strategic managers* [Ecklund02]. We refer to such global channel-reconfiguration as *strategic channel management*.

Channel manager

A channel manager manages a channel-configuration; i.e. it keeps track of which channels are active, and it helps in activating and de-activating channels. There will be one instance per (publisher) binding. Channel managers are located at the binding-server. In the two-level trading approach described below, the channel manager can find already active channels (using profile expressions), it can activate channels by using an activators (resulting from policy trading) and it can de-activate channels not used by any subscribers.

6.2.3. Simple trading

We need to address how the configuration is defined and how channels are activated. A possible approach is to encapsulate all this into *policies.* This is tempting, since channel activation is an application domain specific task rather than a generic middleware functionality. Policies would be used to define the subscriptions to the channels necessary. The service profiles (of policies) state what channels are needed. This is matched with information about existing channels, which could be provided by an *inspector interface* (see section 5.4.3), which is provided by the channel manager. The information about *existing* active channels may also be used to select a policy that also activates the remaining set of channels.

Example policies

If we apply this approach to the channel-config example above (4 channels with dependencies), we need at least 4 policies for subscribing, and 10 policies for activating. 14 policies in total. Table 6.4 below shows the profiles and informal descriptions of these policies. Note that ordering is significant (we assume it selects the first matching policy in the order). It is essential that policies that subscribe to existing channels are placed before policies that create the same channels. For instance, policy #1 re-uses the channel created by policy #5, #7, #9 or #14. The service profile of policy #1 will ensure that it is selected if the channel exists.

#	Policy description	User profile	Service profile
1	Subscribe to channel 1.	TQ[2,2]	VChan + tl[0,2] + ql[0,2] + Lum
2	Subscribe to channel 1 and 2	TQ[2,2] + Colour	VChan + tl[0,2] + ql[0,2] + (Lum ⊕ Chrom)
3	Subscribe to channel 1 and 3	TQ[3,3]	VChan + ((tl[0,2] + ql[0,2] + Lum) ⊕ (tl[3,3] + ql[3,3] + Lum))
4	Subscribe to channel 1, 2, 3 and 4	TQ[3,3] + Colour	VChan + ((tl[0,2] + ql[0,2] + (Lum ⊕ Chrom)) ⊕ (tl[3,3] + ql[3,3] + (Lum ⊕ Chrom)))
5	Add channel 1.	TQ[2,2]	null
6	Subscribe to channel 1, add channel 2	TQ[2,2] + Colour	VChan + tl[0,2] + ql[0,2] + Lum
7	Add channel 1 and 2.	TQ[2,2] + Colour	null
8	Subscribe to channel 1, add channel 3	TQ[3,3]	VChan + tl[0,2] + ql[0,2] + Lum
9	Add channel 1 and 3.	TQ[3,3]	null
10	Subscribe to 1, 2 and 3, add channel 4.	TQ[3,3] + Colour	VChan + ((tl[0,2] + ql[0,2] + (Lum ⊕ Chrom)) ⊕ (tl[3,3] + ql[3,3] + Lum))
11	Subscribe to 1 and 3, add 2 and 4.	TQ[3,3] + Colour	VChan + ((tl[0,2] + ql[0,2] + Lum) ⊕ (tl[3,3] + ql[3,3] + Lum))
12	Subscribe to 1 and 2 add 3 and 4.	TQ[3,3] + Colour	VChan + tl[0,2] + ql[0,2] + (Lum ⊕ Chrom)
13	Subscribe to 1, add 2, 3 and 4.	TQ[3,3] + Colour	VChan + tl[0,2] + ql[0,2] + Lum
14	Add all	TQ[3,3] + Colour	null

Table 6.4. Policy examples

Limitations

The simple trading scheme described here has serious limitations with respect to scalability. In general, it is infeasible, except for a small number of channels. The main reason is that when specifying policies that activate channels, every possible combination of already active channels needs to be handled explicitly. If there are no dependencies among channels, the minimum needed number of policies obviously grows exponentially with the number of channels. To illustrate this problem, try to add policies to support *'TQ[2,3] + Colour'* (with subscriber side frame dropping) to the example (table 6.4) above. We will need 6 additional policies to cover all possible channel-activations. It may be also a tedious and error-prone task to construct each such alternative manually. For instance, our experiments indicate that if we fail to cover all alternatives, there is a danger of producing duplicate channel instances.

In addition, using environment-descriptors to describe all available event-channels is not feasible for a high number of event-channels. Recall that conformance testing using our profile model (chapter 4) are computationally efficient for a low or moderate number of components. Furthermore, long expressions are not very readable for humans and would probably contain redundant subexpressions.

6.2.4. Two-level trading

Because of the limitations of the simple policy trading scheme described above, we consider an alternative negotiation approach where the *channel manager* can activate channels on demand (i.e. when subscribed to) and where policy-trading is applied at two levels: One for subscribing to sets of channels, and one for activating each individual channel. With this approach, we can reduce the minimum number of "create-channel" policies to *one* per channel.

Our experimental design uses *activator components* (see section 5.3.3) both for subscriber bindings and for individual channels. The first level involves activators for subscriber-bindings (to be used by the subscriber and the server). These activators would ask the *channel-manager* for the channels they need. The channel manager consults a policy trader to get policies for channels. These policies contain activators to be deployed on the server (and possibly the publisher). If trading is successful, the channel manager activates the channel and returns the result. It also stores the activator(s) in its own repository, in order to allow subsequent binding to re-use the channel.

Unlike the simple trading approach, the first level policies do *not* specify the channels needed in service-profiles. A policy does not need to assume that certain channels are active. The policy implementation contains a list of needed channels, which are still described using profile-expressions. Expressions are used by the channel manager, and in second-level trading, in order to get to existing channels or channel-policies that satisfy the requirements. They are matched with the *user-profiles* of channel-policies (since they describe the resulting behaviour of channels). The service profile can be used to describe requirements for the environment the channel is to be activated in, e.g. properties of the server platform, the publisher platform or the network.

Example policies

If we apply this approach to the channel-config example above (4 channels with dependencies) we now need 4 policies for subscribing and 4 for activating; 8 policies in total. Table 6.5 shows the policy configuration. Note that adding support for *'TQ[2,3] + Colour'* (with subscriber side frame dropping) would now require only one extra policy. Note also that service profiles are empty since this is a minimum policy configuration focusing only on channels. However, we can use service profiles for other things.

#	Policy description	User profile	Service profile
1	Subscribe to channel 1.	TQ[2,2]	-
2	Subscribe to channel 1 and 2	TQ[2,2] + Colour	-
3	Subscribe to channel 1 and 3	TQ[3,3]	-
4	Subscribe to channel 1, 2, 3 and 4	TQ[3,3] + Colour	-
5	Add channel 1.	VChan + tl[0,2] + ql[0,2] + Lum	-
6	Add channel 2	VChan + tl[0,2] + ql[0,2] + Chrom	-
7	Add channel 3	VChan + tl[3,3] + ql[3,3] + Lum	-
8	Add channel 4	VChan + tl[3,3] + ql[3,3] + Chrom	-

Table 6.5. Policy examples

Environmental properties

To illustrate that environment and service profiles are important, let us extend our focus somewhat, to consider properties other than the events themselves. First, let us assume that the highest quality layer requires a subscriber display with a certain minimum resolution, using the profile *'Display + HiRes'* (high resolution). Let us assume that not all displays are capable of handling colours, hence the profile *'Display + Colour'*. Let us also assume that the policies of channel 3 and 4 need to reserve outbound bandwidth on the publisher node. We refer to section 5.5.1 for an example on how we can do such reservation and how a resource manager can report the amount of reservable bandwidth as a profile expression (through an inspector interface). The addition to the rule base is straightforward:

`BwReservable[`x_1`] ≤ BwReservable[`x_2`], `**`if`**` `x_1` >= `x_2`;` (rule 4)

These assumptions affect policy #2, #3, #4, #7 and #8. See table 6.6 below (*'Sub'* means subscriber and *'Pub'* means publisher).

#	Policy description	User profile	Service profile
2	Subscribe to channel 1 and 2	TQ[2,2] + Colour	Sub + Display + Colour
3	Subscribe to channel 1 and 3	TQ[3,3]	Sub + Display + HiRes
4	Subscribe to channel 1, 2, 3 and 4	TQ[3,3] + Colour	Sub + Display + HiRes + Colour
7	Add channel 3	VChan + tl[3,3] + ql[3,3] + Lum	Pub + BwReservable[800]
8	Add channel 4	VChan + tl[3,3] + ql[3,3] + Chrom	Pub + BwReservable[1000]

Table 6.6. Adding environmental assumptions

Negotiation scenarios

Example 1: Subscriber 1 wants a stream with at least the lowest rate, quality level 2 and colour. It sends a request to the *binding server*. The server may add to the requirement and the environment descriptor before the policy trader is consulted.

R = TQ[1,2] + Colour
E = Sub + Display + Colour + HiRes

The expressions would match policy #2 and #4, and policy # 2 is chosen. The server side activator for this policy consults the *channel manager* to get channel 1 and channel 2. Since the channel manager does not have existing channel activations matching these requests, the trader is consulted again, using the expressions of the requested channels as requirements. Before doing that, it composes an environment descriptor by consulting the inspector of the publisher (and possibly other inspectors).

R_1 = *VChan + tl[0,2] + ql[0,2] + Lum*
R_2 = *VChan + tl[0,2] + ql[0,2] + Chrom*
E = Pub + BwReservable[2000]

The trader returns policy #5 and #6. The corresponding activators are deployed on the publisher which actually activates the channels. The request from the subscriber returns with information about the active channels (IP-addresses), and the subscriber side activator of policy #2.

Example 2: A subscriber wants a stream with the highest rate, quality level 2 and colour. The first-level trading request is:

R = TQ[3,3] + Colour
E = Sub + Display + Colour + HiRes

The trader returns policy #4, and its activator will request channel 1, 2, 3, and 4 from the channel manager. Channel 1 and 2 are already active so they are just returned. Channel 3 and 4 must be traded for, and two requests are made, including an environment descriptor from the publisher, which reports that only 1500 kB of bandwidth is available (due to a high load).

```
R = VChan + tl[3,3] + ql[3,3] + Lum
E = Pub + BwReservable[1500]
```

Policy #7 is successfully traded and activated including reservation of bandwidth. When trading for the other channel, the environment descriptor has changed.

```
R = VChan + tl[0,2] + ql[0,2] + Chrom
E = Pub + BwReservable[700]
```

This trading request will fail since there are no matching policies. Now, the first level activation would also fail, meaning that the whole binding will fail, and the channel manager should be notified to de-activate unused channels. Alternatively, the subscriber could give an alternative requirement, for example by dropping the *'Colour'* part of the requirement. In that case, policy #3 could be used instead, which does not require channel 4. It would succeed by just getting the already active channels from the channel manager.

6.2.5. Implementation

The two designs described above are explored and evaluated by means of a proof of concept implementation. The main goal is to test a *negotiation process* based on using policy trading. Our implementation is based on the design described in section 5.5.2. A channel manager, a slightly specialised version of the server negotiator and a set of demonstrator activators are added. Instead of the real event-channels and video streaming we use placeholder stubs. We first explore the simple trading scheme and quickly discover its limitations. Then we investigate the 2-level trading scheme. Except the policies themselves, the main differences between the two approaches, are in the channel manager.

Channel manager

The *channel manager* keeps track of available event channels and may initiate creation of such. An event channel would (in a full multicast based implementation) be a multicast IP address and a filter configuration on the publisher side, which make sure that events are transmitted using the right network destination addresses. Similarly, subscribers need to know what addresses to listen on and how to handle incoming data. In our experiment, each channel is described by a unique identifier (to simulate address allocation) and a *profile expression* to characterise the events it carries (and possibly other QoS properties). The channel manager also acts as an *inspector* to be used in the server negotiator dynamic profile expressions (see section 5.4.2 and 5.4.3).

In the case of using simple trading, the channel manager offers methods to *add*, *find*, *or remove* channels. The *inspector* operation gets a list of profile expressions describing each available channel, and it combines those expressions using the *component sum* operator.

In the case of two-level trading, the channel manager offers methods to *get* or *remove* channels. The manger keeps *activators* representing *active* channels in a local repository. The *get* method first checks this repository, and if an activator is found there, it is asked for an activation to be returned as the result. If not found, a *policy trader* is consulted to find new activators. It returns *policies* satisfying the profile expression describing the wanted channel. Each policy contains activator classes for the server and the publisher. The manager tries to activate the policies in turn. As soon as activation succeeds for a policy, its server-activator is stored in the repository and the result is returned. For publisher activation, the manager would invoke a method on the publisher to install the publisher-side activator. The manager needs to keep track of the participating publishers (in this case, only one). The inspector method may return properties of the publisher(s).

Binders, negotiators and dynamic profiles

The *binders* (resolver and generator classes) used in section 5.5.2 are re-used with minimal modifications. Each binding instance is set up with an instance of the *channel manager* (channels are not shared between different bindings). The server side *negotiator* (which implements the binding operation) would set up the profile expressions to be used to invoke the policy-trader, as follows:

```
ureq = Expr.SUM(Expr.parse(up), _localUP);
envd = Expr.COMPSUM(Expr.parse(sp), Expr.evaluate(chanmgr.getProfile()));
plist= _ppi.getTrader().lookup( ureq, envd );
```

The user requirement is a sum of a local requirements, *_localUP* (local requirement) and *up* (the requirement coming from the client). The environment descriptor (*envd*) is a component sum of the environment descriptor coming from the client and the result of calling the inspector operation of the channel manager.

Demonstrator activators

A set of activator components are implemented, corresponding to the example policies presented above. We focus on server activators. They interact with the channel manager to find channel-information and to create new channels when necessary. We implement them as a generic class which takes three parameters: (1) A profile expression describing the channels to subscribe to, (2) a set of channels to be created in the case of simple trading and (3) an informal description of what happens. This generic class is subclassed into concrete activators by just adding the needed parameters. Channel-activators for the two-level trading approach are very simple. On the server side, they just allocate a unique identifier (to simulate the allocation of a multicast address). All activators share this functionality.

Publisher- or subscriber side activators establish local implementations of the pub/sub service interface like in section 5.5.2. In a full implementation, *publisher activators* would attach the notify implementation to a mapping and filtering function that either drops events (if no subscribers) or routes them the appropriate event-channel implementations. The mappings would in the case of two-level trading be updated by publisher side channel activators.

Subscriber implementations would need to collect incoming events from a set of multicast addresses and decide if to deliver them to the application's notify interface (if they match the subscription). Server activators would produce the set of multicast addresses for the channels which are passed over to clients by the negotiation protocol. Except that they would follow the same pattern as our demonstrator activators. Concrete policies are made by just parametrizing the generic class by a description of channels to be re-used or created.

6.2.6. Discussion

A goal of this investigation is to see how our policy trading approach and profile model can be applied to a binding type with multiple participants. We study a case where video is distributed to multiple receivers using a publish/subscribe binding type, and where QoS selectivity is approached through subscription to proper sets of event channels (possibly multicast), carrying different aspects of the video stream. Channels may be activated, deactivated (and possibly defined) dynamically.

Using multiple multicast channels like this is appealing for several reasons. For instance, doing event-filtering close to the source, rather than close to the consumer, can save resources of network and receiver nodes. This approach can however be complex in terms of the number of event channels needed to support full selectivity with minimal redundancy, as well as the number of possible subscriptions. In practise, we can exploit knowledge of applications and QoS dimensions to reduce this complexity.

Our policy trading and conformance matching scheme can be attractive in this context, in the sense that meaningful configurations and subscriptions can be defined statically as policies and channel configurations. Therefore, the search for solutions at run-time can be rather simple. Our experiment negotiate subscriptions and channel-activations based on static channel configurations. This is possible and feasible if we trade subscriptions and channel-activations separately. A second level trading operation can be skipped by letting the channel manager create channels directly. Making channel-activation negotiable is slightly more complicated but more flexible: The system can more easily adapt to heterogeneous and changing environments (publisher environments in particular), and QoS requirements. Furthermore, support for new or changing platforms can be added by just adding new policies to the trading service.

Our experiment is based on static channel configurations. In general, we may want to change configurations dynamically, for instance by splitting or merging channels. This will require changes to the set of policies in the trader repository, which raises some issues. First, the order of searching policies is significant, meaning that if a policy is replaced by a set of policies, they may need to be inserted in different places in the order, depending on the other policies. Second, there is a many-to-many relationship between channels and subscription policies, which complicates the management of policies. Third, replacing a channel which is in use may require the clients using it to stop and re-negotiate.

Our experiment indicates that profile expressions are suitable for matching both policies and channels (chanmgr). There are limitations though. We may view channels as components and use the component sum operator to form expressions describing a number of channels. This seems attractive, but only with a limited number of components. Both the readability of expressions and computational complexity may suffer if there is more than a handful of components. Though denormalising may help[36], it can also be somewhat inefficient to describe a number of similar components. This raises the question if it is useful to extend the profile language with iteration or recursion operators, in order to allow composition of possibly unknown numbers of similar components.

[37]Properties that are shared among components, have to be expressed repeatedly for each component when expressions are in the normal form (section 4.3.3).

6.3. Performance measurements

Based on the experiments in this chapter we did some measurements of the time it takes to match and find policies. Table 6.7 below shows the results in microseconds. Using the rule base of section 6.2 (web case) we tested with null requirements and environments for which no policies exist. We then measured the time for example 1 and example 2. Using the example policies of section 6.2.3 (single level pub/sub case) we traded for a full colour full quality binding assuming either two active channels or four active channels. We did the tests several times, on a 2.8 GHz Pentium 4 computer running Linux kernel 2.6.9 and Sun Java 1.6.0, and we computed the mean values for matching each policy and for finding the solution, respectively.

#	Trader config	Test	Resulting policy	Total time	Time/match
1	Section 6.1	R=null E=null	none	-	51 µs
2	"	Example 1 (2 components in E)	policy #6	1224 µs	165 µs
3	"	Example 2 (3 components in E)	policy #5	700 µs	60 µs
4	Section 6.2.3	Channel 1 and 2 (2 components in E)	policy #12	2656 µs	211 µs
5	"	All 4 channels (4 components in E)	policy #4	1269 µs	187 µs

Table 6.7. Measuring time of policy matching

These results indicate that the trading approach is feasible at least for the types of application scenarios we tested. Note that we did not attempt to optimise our implementation with respect to performance. It seems that matching and finding a policy would not be dominant compared with time used for exchanging messages over the network, loading and instantiating activators (including disk access), etc.

We do not observe a significant increase in matching time by increasing the number of components in the environment descriptor from 2 to 4. There is a slight increase in the matching time when going from the web case to the pub/sub case. This is expected since many of the service profiles are longer. Note also that user profiles tend to be shorter and if matched first, it may shorten the search time significantly. In two of the tests (1 and 3) the user profile does not match in most policies. This simple experiment seems to confirm that there are many factors that influence search overhead and that the length of expressions may have little impact as long as they are within manageable limits. We expect that if the length of expressions does pose a significant matching overhead, they are impractical of other reasons as well.

6.4. Concluding remarks

In this chapter we provide some evidence that our approach can be used for application cases like the web or some architectures for multi-subscriber video streaming. First, web applications can exploit policy trading to adapt to changing conditions and requirements. A combination of user requirements and a characterisation of the client browser/web-server (e.g. display size or server load), lead to the selection amongst pre-defined configurations of server components (policies). This experiment indicates that profile expressions may be more compact and composable than e.g. CC/PP profiles [W3C-04a] and that it could be useful if the profile model definition language could be extended to allow conformance rules involving sums.

Second, our approach may support negotiability for publish/subscribe bindings with heterogeneous participants, at least in an architecture where negotiation is facilitated by a single server entity in a LAN environment, and where the application is multi-subscriber video streaming. Our experiment assumes pre-defined sets of possible channels which are selected and activated through selection of proper policies. A policy maps from QoS requirements to event-channel selections. Change to the channel-configuration could lead to change of the policy-configuration and replacement of running policies. We observe that a simple approach to policy trading where activation and subscription are mixed is not feasible except for a small number of channels. A better solution is to do policy trading in two levels: One for subscribing to bindings, and one for activating event-channels. We also observe that it is useful to characterise channels using profile expressions, though it is not feasible to describe the availability of a high number of channels that way. It could be a topic for further research if it is feasible and useful to extend the profile model with constructs for iterative or recursive expressions.

Chapter 7.
Discussion

In this chapter we summarise and discuss the results of this thesis. The overall theme is how to support QoS aware binding at run-time. This involves negotiation of *contracts* and configuration of the implementation of bindings. The problem is approached as follows: (1) We propose an overall model and architecture for supporting negotiable bindings. (2) We develop a language for QoS profile expressions based on declared conformance rules and composition operators, and (3) we investigate how all this can be supported by a middleware level infrastructure.

In section 7.1, we summarise and discuss the main results of our overall architecture, profile language and infrastructure framework respectively. In section 7.2, we discuss our results with respect to some selected topics like scalability, adaptation, composition and generality. In particular we discuss if our results are general in the sense that they are usable outside the context in which they were produced and not limited to particular application domains, technologies, platforms or binding types. This is also somewhat related to interoperability meaning that the independently developed components should be able to bind.

7.1. Results summary

In this section we summarise the results of our work. Early ideas of policy binding and policy trading have been developed and generalised over a period of time. In particular, the simple declared conformance approach has been developed and generalised to a language and a tool to define and evaluate concrete models as rule bases.

7.1.1. Binding and negotiation model

A binding would correspond to a contract and realisation of interaction between the participants such that the contract is enforced. Our approach to contract negotiation is based on *trading* [Hanssen00] of some policy component that defines the contract and configuration of the binding to enforce it. A policy consists of a QoS profile which consists of two parts (expectation and obligation), one or more implementation components, and possibly a set of associated resource requirements (as discussed in section 5.5.1).

Policies (including profiles) are specified with some binding type in mind, and they are managed by a trading service. The negotiation (trading) process match profiles with a complete environment description and a requirement which are composed from parts coming from components taking part in the binding. How parts are composed may depend on the binding type and metapolicy. Figure 7.1 below illustrates how this would look for a client/server binding type. Since profiles are pre-composed and directly associated with enforcement policies, some of the complexity of matching and selecting profiles at a number of different components can be avoided.

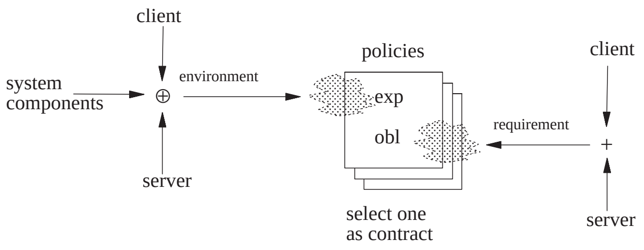

Figure 7.1. Policy trading model

In most related contractual approaches, profiles are specified for interface types or component types and associated with interfaces or components. Each interface or component may have a set of alternative profiles to support adaptation. Figure 7.2 below illustrates how we may view this in a client/server binding type. Negotiation may involve complex matching of many profiles of many components.

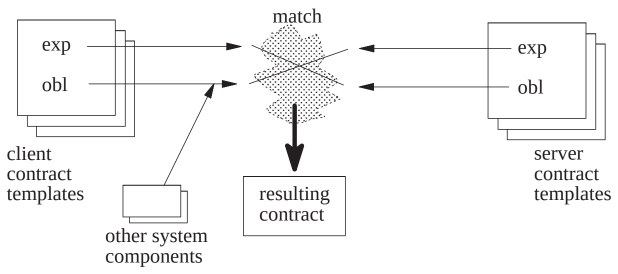

Figure 7.2. A more traditional negotiation model

We also propose a distinction between bindings and their *activations*. This distinction may simplify the understanding of QoS contracts and adaptation, as well as late binding, since contract negotiation (or re-negotiation) is decoupled from binding establishment. A meta-policy can specify how a binding activates, how it negotiates contracts, or how it adapts to changing environments or requirements.

7.1.2. Profile model

An important contribution of our work is a language for expressions to be used in profiles, requirements and environmental descriptions involved in negotiation. It supports evaluation at run-time for conformance. This language is defined and evaluated in chapter 4. We define how expressions can be constructed from atomic QoS statements termed *'basic profiles'* using composition operators. Two such operators are defined: The *sum* ('+'), which corresponds to conjunction and *component-sum* ('$\oplus$'), which implies separate contexts for the operands (which therefore must be satisfied separately). To define the semantics of any expressions using these operators, we define a distributive law and a normal form. Based on these definitions and conformance theorems, we are able to develop an algorithm for conformance checking expressions.

Concrete profile models explicitly establish conformance relationships between basic (atomic) profile expressions. These are typically defined for specific application domains and are defined as *rule-bases* which are essentially sets of axioms from which we can infer conformance between any pair of basic profiles. From an axiom set, we may derive a full rule set, covering any pair of profile-types. Such a rule set can be directly mapped to executable code which allows efficient conformance checking at run-time. As a proof of concept we implemented a *profile model compiler* (which can also be useful in analysing consistency and performance issues). The compiler performs a basic semantic check of rules, computes the derived rule-set by using a *transitive closure algorithm* and outputs conformance checking code.

Discussion

An initial idea [Hanssen98] was to base dynamic QoS statement on *declared conformance,* meaning that a concrete profile model would simply be a directed acyclic graph of profile names where the edges define conformance relationships. This simple approach was extended [Hanssen00] since it is too inflexible for the general case, where we want to involve environmental properties of multiple components involved in the binding. First, the sum operator was introduced to compose expressions describing different QoS dimensions. Second, the component sum operator was introduced to compose expressions about separate environments. The model was formalised and we developed an algorithm for testing expressions for conformance. Furthermore, in some cases (like e.g. the latency time in section 5.1), models may still be too inflexible if we allow simple declared conformance and sum operators only. To cover a sufficiently wide set of situations, we may need to define a high number of profiles, leading to more complicated models than necessary. Therefore we propose to allow profiles with parameters and use rule bases to define models [Hanssen05a].

The result may therefore be viewed as a *compromise* between declared conformance and parameter value comparison. It may be tempting to view the definition of such profiles as user defined QoS characteristics, but our profiles are meant to be simpler and more abstract than the QoS characteristics of e.g. CQML. Also, semantic rules defining conformance are more explicit and general compared with languages like QML or CQML (section 2.4.1).

We evaluate our model in section 4.5. There are important issues in how concrete models (rule bases) are defined. First, our definition of conformance rules means that models do not need to be complete in the sense that all possible conformance relationships are covered by the rule base. We require that models do not produce false positives. This is based on an assumption that we always want to avoid selection of incorrect policies, and that there may exist conformance relationships that we do not need in practice. Models should be *sufficiently complete*, meaning that they should cover needed conformance relationships. Second, models should be *consistent,* meaning that we should not be able to infer contradicting results from a rule base. Since models are not complete, we cannot prove that they are consistent, but we can prove that inconsistencies exist, which is useful. We proposed a definition of inconsistency (based on which ranges of profile parameter values evaluate to true) and proved that it is sound.

This means that we have some tools to evaluate concrete profile models. There are inconsistencies and potential problems that can be detected and reported. It is useful to view models as graphs where edges are rules and nodes are profile types. We observe that cycles in such a graph may indicate derived rules that are not easily foreseen by the model designer and that may be imprecise (not sufficiently complete). Cycles can be problematic except for symmetric rules or equivalence and should be reported by profile model compilers.

7.1.3. Infrastructure support

Our binding model and negotiation scheme should be supported by an infrastructure, mainly a middleware platform offering proper abstractions and services. The infrastructure should (1) support negotiable and adaptable behaviour (of bindings); i.e. it should be able to reconfigure itself at run-time to change its internal behaviour, and (2) support the process of negotiating such behaviour. In chapter 5, we describe an experimental framework which serves as a proof of concept and as a tool for exploring our concepts.

In an early phase of our work, we designed the FlexiBind framework [Hanssen99] which was based on the ANSA FlexiNet framework[37]. These experiments represent early contributions to reflective middleware research. The focus was how to support dynamic configuration of bindings by pluggable and replaceable policies. The distinction between bindings and their activations may help decoupling policy and application programming. We identify two types of pluggable policy components: (1) *Binders* (generators and resolvers) which represent binding protocols and metapolicy and (2) *activators* which define how negotiable aspects are configured. In principle, each invocation is *resolved* (by a resolving mechanism), not only to a target object but to a configuration of negotiable aspects (layers or other resources). Bindings (also non-active) are represented by *binding proxies*, which can resolve such activations. When activated, a proxy is attached to an activator and (possibly) a metapolicy object which defines how activators are selected.

In later work [Hanssen05a], we explore further how middleware can support profile expressions and negotiation based on policy trading. Negotiation mainly involves collecting environmental properties from the platform and requirements from applications. Binder- and negotiator components define how environment descriptors are collected and composed. The prototype framework supports dynamic profile expressions which can contain placeholders for subexpressions to be determined by querying the platform at negotiation time. Such querying is done on inspector components which can map from platform dependent characteristics to the more abstract profile model. Such mapping can be highly configurable and set up or modified by binder components. However, this scheme requires some conventions for naming of inspectors.

Experimental evaluation

An experimental framework is implemented and serves as a proof of concept and a tool for exploration. It is further evaluated with respect to two cases. First, we investigate how resource reservation in the operating system or network can be incorporated into policy binding. Due to platform heterogeneity we believe that resource management should be done by pluggable components, which fit into a framework defining *resource managers*, *reservation domains* and *resource consumers*. An idea is to extend the inspector interface with an admission test operation. Our approach also implies a possible separation of concerns between policy implementation and resource requirements. An experiment (using Linux packet scheduling) demonstrates that this approach may work in practise, for instance to bound invocation delay.

Second, we evaluate how our framework can support binding types beyond simple client/server activation, and in particular those having more than two participants. As a proof of concept, we design and implement a framework for negotiable publish/subscribe bindings. We observe that it can be made highly negotiable how event streams can be implemented, in particular since our

37. We also participated in developing the FlexiNet framework itself.

framework allows binding to a service before it is decided how to implement it. This experiment indicate that we can realise publish/subscribe bindings, but it raises issues about the relationship between the binding and associated contracts, as well as how to manage implementations or state shared between different participant bindings. How this is handled would be more application domain specific. In section 6.4, we further evaluate our approach by applying it to a specific application case using subsets of shared multicast channels to stream video. Here, we need an additional entity to manage channels and channel configurations (channel manager) and we observe that we can benefit from doing policy trading in two levels: (1) policies for activating channels and (2) for subscribing to them.

7.2. Evaluation and discussion

In this section, we discuss and evaluate our results with respect to some selected topics: Composition (section 7.2.1), orthogonality (section 7.2.2), adaptation (section 7.2.3), performance and scalability in trading (section 7.2.4), and generality (7.2.5).

7.2.1. Composition

A service may be realised by a *composition* of services or components. A component which implements a service may have expectations which involve not only the client behaviour but other components of its environment as well. These components may have their own expectations, meaning that a contract validation may in general involve finding a transitive closure of component dependencies. In typical related work (see figure 7.2), when negotiating binding between a client and a service, there will be a partial match between their profiles. The expectation of the server profile would be satisfied by a combination of the obligation of the client and obligations of other components in the environment, etc. The matching process may be complex since it involves profiles of several components, but also since it involves evaluation of several QoS parameters which may interact in various ways.

Our approach contributes to simplifying composition in QoS modelling and QoS negotiation. There are two reasons for this: first in our architectural approach as discussed in section 7.2.1, *profiles* are associated with *policies* for binding, instead of service interfaces or components. A policy specifies a mapping from a sum of environmental properties to resulting QoS. It *encapsulates* non-trivial interactions between environment components. Second, this idea is strongly supported by our profile expression language, which is based on declared conformance rules and composition operators (chapter 4). The scope of profile expressions/profile models is to describe what components are needed, what components are available and constraints on how policies can combine them, rather than what compositions result in.

Profile expressions represent constraints on behaviour, but these are not directly stated. Evaluation at run-time can therefore be simplified. If we view profile expressions as constraints, composition can be understood as logical conjunction or disjunction. For instance, a profile may assume behaviour A <u>and</u> behaviour B, or one may assume behaviour A <u>or</u> behaviour B. Behaviour A and behaviour B may be in the same or in different contexts. Composition operators represent conjunctions and component-sum also imply *separation of context*. If we register multiple policies sharing implementation and user profile, this may be viewed as disjunctive composition in the sense that one of the alternatives may be selected.

In chapter 4, we show that the model defining the semantics of profile expressions and composition operators is sound. In section 4.5.4, we discuss how composition operators are used and how some known composition patterns can be supported. It is however necessary to define some pragmatics for how compositions are expressed (section 4.5.4), and it is useful to check models for consistency- and interoperability problems (section 4.5.2 and 4.5.3). In chapter 6 we apply our approach to certain application cases and we validate that our model can be used for the compositions needed there.

Nested composition

In general, composition can be nested, and in the context of our model, there are two different cases to be considered. First, a component of an environment may be realised or supported by using other components. This could imply that we do binding again, recursively. Second, a description of an environment may consist of components, and the contexts of some components appear inside the contexts of other components. Our profile model seems to support nesting of expressions, but as pointed out in section 4.5.4, this is in a somewhat limited sense. In particular, there are cases where we may wish to express constraints on the identity or location of containing contexts, and this does not necessarily follow from the expression nesting. It is possible to identify constraints more explicitly by adding additional constraints. A more abstract and portable approach may be add a kind of *labelling* to the profile model but it remains to be investigated if this is feasible.

Nested composition seems to be solvable in our application cases in chapter 6. In section 6.2 we test both ways to do nested composition. First, we try to do only one level of binding and to describe the appearance of multiple channel components in service profiles and environment descriptors. We observe that this is not generally feasible. The problem is however not related to the limitation mentioned above, since the identity of each channel is not significant in negotiation. The main problem is that mixing channel subscription and channel activation in the same policies lead to an unnecessary high number of policies. Furthermore, there is a practical limit to the length of profile expressions. An approach doing policy trading in two levels, seems to be more scalable and would also better support heterogeneity of channel implementations.

7.2.2. Orthogonality

Orthogonality means that extra-functional behaviour should not be tied to specific service interface types. This is motivated by the desire for reusable components in open systems, as well as the desire for automatic adaptability. Any type should in principle be allowed to negotiate a given extra-functional behaviour. We also distinguish between *abstract* extra-functional behaviour and the policies or implementations which enforces them. The orthogonality requirement was meant for the abstract behaviour and the abstract interface, but it is relevant to discuss orthogonality with respect to policy implementations as well.

Abstract behaviour orthogonality

We claim that our contract and policy concepts support the reasoning about extra-functional behaviour to be orthogonal to functional types. In our architecture we introduce policies and their corresponding software components (activators) as an extra level of indirection; i.e. we do not apply policy implementation components directly but rather via a policy component. This represents a separation of concerns between policy and the resulting behaviour, and would together with policy trading, abstract over differences in how a given extra-functional behaviour is realised. An important part of our approach is that policy implementations should be associated with explicit assumptions, which may include the type of target interfaces. We claim that our orthogonality goal is reached since (1) one can define an abstract behaviour which is meaningful for a range of different service interfaces, (2) since one can create alternative policies for different environments and target types, but which satisfy the same abstract behaviour and (3) since we can do transparent selection and installation of policy based on an abstract requirement. This is also supported by the binding type concept. A binding type does not have to include the type of participant interfaces, but rather the *roles* they play in the binding.

Orthogonality for a given behaviour may however be limited in the sense that policies are not available for all target types. It may be a question of making policies for all needed cases. However, all abstract behaviour would not be meaningful for all possible service types. For instance, transactional behaviour (ACID) is not relevant for stateless or read-only services, colour is not relevant for audio streams, frame rate or jitter is not relevant for services which are not continuous media, etc. This issue can be resolved in two ways: (1) define that a property is *never* satisfied for the type, (2) define that a property is *always* satisfied for a type. For instance, a jitter requirement would always be satisfied for a still-image.

Policy implementation orthogonality

Though extra-functional behaviour can be viewed as orthogonal to functional behaviour, their *implementations* are not necessarily so, and sometimes such behaviour would be related to some of the underlying syntax or semantics of the functional interface. For instance, the implementation of transactional behaviour may need to distinguish between methods which read and operations which update the state of the object. For instance, a mean latency time requirement should be equally meaningful for all operational interfaces and individual methods. However, the implementations (e.g. use of caching) may need to discriminate read and write operations, especially when consistency or recoverability is an additional requirement. Some properties cannot be implemented without making assumptions about target implementations or the semantics of their individual methods. A goal may be to minimise assumptions or to make assumptions which hold for a wider range of targets. Assumptions should be expressed explicitly in service profiles to aid the negotiation. Trading can ensure that policies are not deployed in incompatible environments.

However, our observations suggest that simplicity and generality of policy implementations may be conflicting goals. Policy implementations which reduce assumptions tend to be more complex since they must test for many cases at run-time and (possibly) use reflective techniques to expose implementation issues. Furthermore, reducing assumptions often implies making *meta-level* assumptions, for instance that certain naming conventions are used to identify operation semantics.

7.2.3. Adaptation and extensibility

Adaptation is run-time modification of active bindings. In our context, there are two approaches to this: (1) that the policy is explicitly re-negotiated and replaced and (2) that the policy itself adapts its behaviour. In the first case, the contract is changed such that the new service profile is satisfied by the environment. In the second case, the adaptation stays within the constraints of the contract. Since a service profile denotes a *range* of states the environment can be in, we assume that the policy itself can deal with variations within that range. We do not focus strongly on adaptive policies or metapolicies themselves, but our architecture is designed to support such policies and extensibility of adaptation range through adding new policies. This needs to be discussed.

Our architecture addresses renegotiation in the sense that activators may be *replaced* within the lifetime of a binding. Adaptation can conceptually be understood as switching between activators. Activations (layer stacks) are constructed when an activator (a policy) is instantiated (activate), though they can in principle be constructed at invocation time (see section 5.3). Even if re-activation means replacing the activator, it does not necessarily mean a full re-construction of the layer stack and re-allocation of resources. Different activators may share or re-use parts of the activations. In effect, re-activation may result in adjustments to existing stacks (and associated resources). For instance, frame dropping may be added to the sender stack in response to a drop in network bandwidth. In our current design, policy programmers would need to make assumptions about the previous activator, in order to re-use parts of it or to transfer state consistently from one version of the binding to the next. Such assumptions may be explicit in negotiation, and activator instances may implement the inspector interface (section 5.4) to describe themselves.

Safe reconfiguration

When adapting by replacing activators, it is important to ensure that the transition does not lead to violation of call semantics or other inconsistencies (like those identified in [Goudarzi99]). Adaptable bindings should therefore be able to keep track of ongoing interactions and to drive the binding into a *safe state* before reconfiguring. In addition, the interruption of ongoing invocations should be as little as possible or within bounds (possibly specified as a QoS requirement).

Our prototype implementation protects the activation using locks. For typical RMI interfaces, our binding objects uses a classical read/write lock which ensures that the reconfiguration does not happen while invocations are in progress, and that it eventually happens. Invocations require a read lock and there may therefore be multiple concurrent invocations (read lock as long as an invocation is active), but changing the activation or activating/passivating/re-activating it requires exclusive access (write lock). This is a simple approach which is not necessarily suitable for all cases. For instance, if bindings support transactions consisting of several causally related invocations, and if the negotiated QoS is with regard to the execution of the transaction as a whole. In that case, one may need to hold a read-lock as long as the transaction is active, and the simple locking scheme may be problematic with respect to performance since the safe state cannot be reached until the transaction is committed. For stream bindings, simple locking per stream would prevent any adaptation, and it may be a unnecessary overhead to request a lock per frame. For such cases it could be beneficial to use more advanced synchronisation schemes (see e.g. [Wegdam03]).

Adaptation extensibility

Our architecture allows embedding some adaptation in the policy implementation instead of adapting by invoking explicit re-negotiation of the contract. This makes sense in cases where an adaptation method itself is application domain- or technology specific, or where the adaptation method is more efficient than re-trading policies. For instance, it may use feedback loops to continuously adjust various parameters to maintain the contract in response to changing measured QoS or environmental properties. It could also make sense in the case of section 6.2, where a policy determines the number of multicast channels subscribed to (carrying different quality levels of a video stream). A policy may be made to adapt to smaller variations in e.g. load. Larger variations may require changes of channel subscriptions and thus replacement of policy.

Therefore, adaptation is not strictly a metapolicy issue, but sometimes also a policy issue. We would like extensibility in the sense that one running policy can be automatically replaced with another which the implementer of the first one did not foresee. A specific adaptation scheme can be embedded in a policy working for a range of environment properties. New policies can be dynamically plugged into a running system to extend the adaptability range. When a contract cannot longer be enforced, this may trigger explicit adaptation: The environment descriptor is reformulated and a search for a new policy and contract is initiated. If a policy is not found, we may try to negotiate with users to see if they can accept a degraded user profile.

7.2.4. Scalability

It is of interest how binding performs when the size of the system grows in size. There is no simple answer to this, since there is more than one dimension in which to characterise the size of a system, and since it may depend on the binding-type, metapolicy, negotiation protocol, etc. We may consider the number of participants that can be involved in a binding, or the number of possible policies. Multi-participant binding in particular is discussed in section 7.2.5 below. In the rest of this section we discuss the following issues:

1. Architecture of the policy trading service; i.e. dependence on a central trader service can represent a bottleneck.
2. The number of candidate policies to search, or more specifically, the number of expression matchings (conformance checks) needed to find a contract.
3. The length and structure of profile expressions used in negotiation. These are related to the number of components which are involved in a negotiation.

Trading architecture

Dependence of a centralised trader component could be a scalability problem, and the traditional trading approach has been criticised for it [Vasud98a]. In our approach, the policy-trader should be regarded as a *conceptual* entity, and it may be distributed and/or replicated over many locations. We find that it can be convenient to have one instance of a trading service per service instance and co-locate the trader with the server. More generally it may be desirable to share policies between different services and even applications. Like in traditional trading, it is also possible to federate separate trading domains. Therefore, an issue is to what extent the trader information can be partitioned such that lookups between trader federations are minimised. This problem is related to the indexing problem briefly discussed below.

Candidate policy search complexity

If in policy trader implementations, we use *linear* search to find matching policies, this may limit the scalability with respect to the number of candidate policies available. It may however be possible to *index* or order the database of candidate policies according to conformance relationships, but this is not straightforward since conformance defines partial orders. Furthermore, the order of policies may be significant for other reasons (as observed in chapter 6).

It is beyond the scope of this thesis to explore the use of indexing on policy files. However, we can identify it as a partial-order indexing problem. This is investigated e.g. in the context of object oriented databases [Bertino95]. If we restrict the profile expressions to sums only, indexing profile expressions is equivalent to the problem of indexing on *set attributes*. The search for a conformant user profile corresponds to finding subsets, and the search for a service profile would correspond to finding supersets. [Goczyla97] outlines a possible solution which is based on *partial-order trees*. This could be used in our context as well (with some adaptations). However, further research is needed on efficient implementation as well as how to incorporate component sum expressions.

Worst case complexity

When discussing scalability and performance of trading policies (see also section 4.5.1 and section 6.3), we should analyse the worst case complexity of policy trading with respect to the number of components involved in the negotiation and the number of possible policies available. If the number of components involved in the negotiation is denoted C and the number of candidate policies is denoted M, the worst case scenario is $O(MC^3)$, since it is possible to implement the conformance checking with an $O(C^3)$ worst case performance (section 4.5.1).

Our trading approach is not easily comparable with e.g. CQML due to the different architectures. If we negotiate by matching profiles specified on each component or interface to participate, and the number of profiles at each component is denoted N, the worst case complexity seems to be $O(N^K)$ where K is the number of components involved as active participants of the binding (e.g. client and server). A problem in such a comparison is how we can characterise M in terms of N, and C in terms of K. If we assume that the scope of a given trader database is limited to the involved components, a theoretical upper limit for M would be all possible combinations of profiles from these components. However, we can assume that most combinations are meaningless and can in our approach be eliminated statically. This indicates that our approach may be more scalable in some cases. Also, the cost of profile comparison itself is not easily comparable. It would be lower in our approach because of our use of declared conformance and the simpler structure of profile expressions.

7.2.5. Generality

We claim that our results are *general* in the sense that they are usable outside the context in which they were produced, and that they are not tied to a particular application domain, technology or platform. As a part of validating this, we have developed an infrastructure framework that is highly configurable and where extra-functional aspects of bindings are highly negotiable. This is related to the orthogonality claim discussed above. Our approach was successfully applied to different application cases. In both cases, resulting profile models were simple and manageable. Certain extensions to the infrastructure were necessary in the publish/subscribe case and the most obvious and simple way of using trading was not scalable. However, a two-level approach was found to be effective, and the use of a channel manager does not invalidate a claim that our results are general.

Our profile language is general purpose in the sense that it is not restricted to specific QoS categories or application domains. This claim is supported by analysis and by applying it to several examples. It supports defining application specific concrete models and compositions. However, there are issues concerning the generality and interoperability of concrete profile models. The scope of our language is rather narrow. It supports conformance matching, but each participant may need to interpret profile expressions in terms of measurable characteristics, and they should do so in a consistent way. In practise, this means that implementers of policies, inspectors and metapolicies would need to have some knowledge of the meanings of expressions that cannot be derived from the rule bases themselves. This would need to be specified elsewhere. Advantages of this separation of concerns include that we may define models that are general and reusable; we may define hierarchies of rule bases where more application-specific parts may be based on more generic parts and mapping rules between such parts may be defined to support interoperability. However, allowing interpretations to be context-dependent may also lead to subtle interoperability conflicts if not careful (see section 4.5.3).

Another issue related to our choice of focus is that our model does not define degrees of conformance or QoS satisfaction. In contrast, related work based on utility functions or worth based negotiation (see chapter 2) may define total ordering. By matching profiles we can only determine if a policy is satisfactory or not. The ordering of policies in the trader repository is a way to define what policy to be selected first (based on policy designers knowledge on the application domain), a metapolicy may define a way to order candidate policies returned from a trader (for instance based on resource requirements), and clients may specify ordered alternative requirements to support degradations. We suspect a limitation to generality in the sense that negotiation based on strict conformance is well suited for certain types of applications, while *not* well suited for other types.

Our approach is not platform specific, but there may be challenges in defining profile models and policies which are portable across platforms. Profile models should abstract over platform specifics to support interoperability between components running on different platforms. On the other hand, platform specific capabilities may be exposed through environment descriptors to allow them to be exploited. This means that policies may be used to encapsulate and abstract over platform dependent solutions.

Binding type and scalability

We claim is that our results are usable for other binding types than RMI and also bindings with more than two participants. To validate this, we investigate how we can design and implement a publish/subscribe binding type as was also done in [Parlav03]. We perform two experiments: one to see how we can implement infrastructure support (section 5.5.2) and one to see how we may apply this to an application case (section 6.2). We are able to re-use most of the infrastructure used for RMI bindings, in particular the binding-proxy and the binder framework. It is convenient to use operational interfaces (and RMI) in parts of the implementation (binding, event delivery). Implementations of event streams are completely negotiable though.

These investigations support a claim that our approach is general enough to be used for multi party binding types, at least in some application scenarios. However, there are still issues which deserve more discussion, in particular when it comes to scalability.

A contract for a multi-party binding can be viewed as composite in the sense that it can be described in terms of contracts for individual participants. In our policy model, an expectation should be satisfied by a composition of participant environments, and an obligation should satisfy all requirements of participants. Each individual participant may have requirements which are satisfied separately, without involving other participants. For instance, different subscribers of a publish/subscribe binding may require different filtering of the content. All participants are not necessarily in the binding at the same time and they should be allowed to come and go over time. Furthermore, it can be impractical to involve a large number of participants in a single negotiation operation. A multi party binding may in our conceptual model have a single contract, but in practise, a negotiation should not involve more participants than necessary. This is obviously important for scalability. In our experiments, negotiations typically involve two participants, the joining subscriber (or publisher) and a negotiation service. However, in section 6.2, the subscriber negotiation, *publisher side inspectors* may be invoked when trading for channel (second level) policies.

The concept of contracts being composite (or even hierarchically composed) could be a topic for further research. For example, it would be interesting to see if it fits into the architecture of [Ecklund02] where the authors propose hierarchically composed domains of QoS management. In related research on reflective middleware, there are global and local bindings, but local bindings are assumed to be trivial. Our investigation indicates that it can be convenient to negotiate bindings separately for individual participants, but we may require them to be consistent with some global contract (for the multi party binding as a whole). We may require the component sum of local expectations to satisfy the expectation of the global contract and we may require the obligation of the global contract to satisfy the sum of all local obligations.

7.3. Summary

This chapter is a summary and discussion of our results. We describe how ideas of policy binding and policy trading have been developed and generalised. A declared conformance approach has been developed and generalised to a more general language for QoS profiles. An infrastructure framework has been developed and evaluated over some time.

# Chapter 8.
Conclusions

The main goal of this thesis is to develop the foundations of a QoS aware binding facility, which maps from application requirements plus descriptions of the environmental properties to suitable bindings between components. We investigate the use of *policies* as architectural entities to encapsulate potential QoS contracts and their implementation, the use of *policy trading* as a principle for negotiation and the development and evaluation of a language for profile-expressions, based on conformance rules and composition operators. In this chapter we summarise what are the main results of our investigations (section 8.1), we present some critical remarks (section 8.2) and we discuss research trends and possibilities for further research (section 8.3).

8.1. Main results

The main contributions in this thesis are related to (1) an alternative binding and negotiation model, (2) a language for profile expressions and (3) infrastructure support.

8.1.1. Negotiation model

We propose a scheme for QoS aware binding where potential contracts are predefined. A *policy* is an entity consisting of an obligation (*user profile*), an expectation of the environment (*service profile*) and an enforcement policy (typically an implementation component). Policies (including profiles) are specified with some *binding type* in mind, and they are managed by a *trading service*. Policy trading is essentially to search for policies where the user profile satisfies a QoS requirement, and where the service profile is satisfied by an *environment descriptor* (a composition of environment descriptions coming from the components taking part in the binding). *Metapolicies* dictate how to manage policies, i.e. how to negotiate and (possibly) how to react to changing environmental conditions by re-negotiating.

This means that contract templates (profiles) are decoupled from interfaces or components and associated with a policy trader instead. This has a potential to simplify negotiation, since it is reduced to a linear search, since there is a clear and simple semantics of matching profiles and since composition can be handled by just adding expressions before trading is invoked. Solutions (possibly end-to-end) can be defined statically instead of e.g. matching a number of alternative profiles on each participating component against each other to find usable combinations.

Separating functional and extra-functional behaviour is not a new idea. Since we started our work, much progress has been made in aspect oriented programming and dynamic aspect weaving, so this is not a focus here. However, QoS contracts are typically more or less associated with interfaces or components. In our approach, the concepts of *contract*, *policy*, and *activation* of bindings support the reasoning about extra-functional behaviour to be *orthogonal* to functional types or service instances. Our approach adds a level of indirection between negotiation and the implementations, it let us describe behaviour in an abstract way, and it supports alternative implementations of the same abstract behaviour in a transparent way. We may support a high level of negotiability; we have for example demonstrated how we may bind to a service and where its implementation may be decided on later (when activating).

8.1.2. Profile expression model

A main contribution of this thesis is a language (profile model) for profile expressions, designed towards conformance checking and composition. This language is founded on declared conformance rules and composition operators: *Sum* (which corresponds to logical conjunction) and *component sum* (which combines expressions in separate contexts). Expressions are composed from *basic* (atomic) *profiles* which are simple names or names associated with numeric parameters. Concrete profile models (for applications or application domains) are defined as *rule-bases*, defining conformance between basic profiles. We demonstrate how we can compile such rule-bases into efficient conformance checking code and that a general conformance checking algorithm can be made. We evaluate the model with respect to performance, consistency and completeness problems, interoperability problems and composition. We find that viewing rule-bases as graphs and using a profile model compiler provide us with tools to help identifying some of those problems.

Our language implies a separation of concerns between conformance relationships and the interpretation of profiles in terms of measurable characteristics. This allows expressions to be simpler, more general and more abstract than more traditional approaches based more directly on measurable characteristics. This claim is supported by analysis and by applying the approach to application examples. However, since participants may need to define how profile expressions are to be interpreted, and since the meaning of a profile may be context dependent, care should be taken to avoid interoperability conflicts when defining models.

Our profile language supports composition by having generic composition operators and clearly defined composition semantics. Service profiles and environment descriptors would capture the availability of components and constraints of different contexts, possibly including constraints on how components may be combined. The policy concept is meant to encapsulate the effect of composition, i.e. how component properties are combined (possibly in non-trivial ways) to achieve a result. This means that the use of policy trading and composed service profiles can simplify composition (in negotiation), but (as observed in section 6.2) it does not necessarily eliminate the need for invoking the negotiation process recursively in the case of nested composition.

8.1.3. Infrastructure

To validate our approach to negotiability, we explore a middleware level binding framework. Some of these experiments represent early contributions to reflective middleware research, and much has been done since by others. An important contribution is the distinction between bindings and activations of bindings. Activations are defined and managed by *activators*, which could be part of the traded policies. Furthermore, bindings are set up by pluggable *binder* components (resolvers and generators) and managed by *metapolicy* components.

We demonstrate how to incorporate support for policy trading, and in particular, how to setup, produce and evaluate profile-expressions in an infrastructure. Binders and negotiator components compose expressions (environment descriptors and requirements) and expression parts can be collected from *inspectors* that can be dynamically installed in the middleware platform or in the bindings themselves. We demonstrate that it is possible to incorporate support for system level resource management, and that such support can be effective in constraining e.g. invocation time. We also demonstrate that our framework can be used to negotiate publish/subscribe bindings. This indicates that we can support binding types beyond RMI and in particular, multi-party bindings. In our demonstration, each individual participant initiates negotiation individually, but our negotiation scheme supports incorporating constraints or components of a more global binding level. This is explored in section 6.2 where bindings activate sets of event channels which are shared between participants. It is useful to describe such channels using profile expressions

8.2. Critical remarks

The work reported in this thesis has been done over a long time span and much work has been published, especially in the area of reflective middleware, since we started. However, our early contributions were published and presented to the research community, and they are not invalidated by later research. Our core concepts are rather general and we do not see fundamental problems in implementing them on more recent reflective or component oriented middleware like OpenCom. Also, the acceptance of a paper on the Adaptive and Reflective middleware workshop in 2005 [Hanssen05] should indicate that we have produced some results which are still regarded as relevant by the research community.

One may argue that the usability of our approach is somewhat limited since the solution space for bindings (an thus, possible compositions) is a fixed number of statically defined policies. One may also argue that the policy approach is based on an assumption that it is possible for a component developer to know (in advance) what resources or configurations are necessary to achieve a particular QoS level. This can be a non-trivial problem in open systems. However, our approach does not exclude policies that make some of the decisions themselves (within the constraints of their profiles). A policy may for example use a feedback loop to continuously adjust various characteristics of the binding. Furthermore, our approach does not exclude using policy trading recursively as demonstrated in section 6.2. At last, we do not exclude the possibility of generating policies automatically by using compilation and run-time probing techniques like in 2KQ+ (see section 2.4.2). Policies may be added or replaced in an evolving system.

One may argue that policy trading is of limited use since it is based on strict conformance. We present a rather fixed view of application QoS requirement, and negotiation does not deal with optimisation of resource utilisation. However, our approach does not exclude the possibility of (during negotiation) ordering satisfactory policies according to a metapolicy (and possibly additional information attached to policies), and it does not exclude the possibility of attaching alternative requirements to client requests to be considered by the trading service if the first priority requirement cannot be satisfied. In our cases (chapter 6) we show that it is sometimes necessary to order policies in the trader search according to the designer's knowledge on what is "best", and that this (somewhat ad. hoc.) approach can be useful.

8.3. Further research

At last we briefly summarise the research trends in our area of interest the last decade and identify some areas of current and future research. Based on our own results we identify some issues for further research.

8.3.1. Research trends and challenges

During the progress of this thesis, research on reflective and adaptive middleware has been very successful. It has been demonstrated [Blair04] that reflection can be a more open and flexible approach to middleware, that it does not necessarily incur a performance overhead, and that it can be platform independent. It has been demonstrated how to provide very configurable middleware as component frameworks [Coulson02, Coulson04]. [Blair04] identifies some future research directions for middleware where focus is on how to extend the scope of such middleware, both *in depth* (going further down towards the system level) and *in width* (investigating new application areas). Current research indicates that such middleware can successfully be tailored to support interaction paradigms like peer-to-peer, publish/subscribe etc., or used in application areas like the GRID, mobile computing or sensor networks. Furthermore, reflection and QoS awareness seem to support the autonomic computing vision [Anane07].

Traditionally, research on QoS has been focusing on specific architectures, technologies or application areas like networking or multimedia. In the last decade, there has been an increasing interest in more general QoS architectures and QoS support at the middleware level. Though significant progress have been made in QoS modelling [Aagedal01], or middleware architectures [Nahrstedt00, Ecklund02] the area of QoS support for open and distributed systems can still be regarded as immature and many questions remain both in QoS modelling, QoS negotiation and in QoS management. Current research trends include how to integrate QoS in component infrastructures, how to support automatic composition based on QoS requirements [Staehli04], and how to support QoS web service composition [Zeng04, Thi06] as well as mobility, etc. We highlight the following as an important research challenge:

QoS aware composition in open systems where we can assume very little about the implementation of components involved. We would depend on a suitable notion for describing capabilities in an abstract way and to reason about composition of those. Modelling, static analysis and run-time expression (and negotiation) are important issues. Reflection has proved to be an useful principle and it would also be interesting to see how this can be used in this context. For instance, mirror based reflection may be an interesting approach [Eliassen06].

8.3.2. Profile model

This thesis has laid the theoretical foundations for a profile expression language for contract negotiation and identified its potential to capture conformance relationships and composition. The language features investigated here are still just the core features. This means that we may extend the core model in order to simplify expression, to improve expressibility, to help making correct models or to better support interoperability. We have identified several issues which could be interesting topics for further investigation:

Section 4.6.1. identified some ways to extend the profile model; some of them were also justified in chapter 6. Many of these suggested extensions relate to (static model) definition and include allowing equivalence rules, symmetry rules to capture typical patterns and arithmetic operators to support scaling of parameters (which is important for interoperability). We may also consider additional semantics, for instance to constrain the values of parameters.

Possible extensions to the (run-time) expression language include labelling of expressions to provide abstract identity for instances in nested compositions. We may also benefit from some higher order syntax that capture common composition pragmatics or that increase the readability of expressions.

A more fundamental issue is if the language could be extended with recursion or iteration operations (e.g. Kleene closure) to capture (possibly infinite) sequences of similar components.

Furthermore, it is relevant to investigate how to define mappings between profile expressions and more measurable characteristics of the participating components. Implementers of policies, inspector components, etc. may need some knowledge on the meanings of expressions that cannot be derived from the rule-bases themselves. This must be defined elsewhere, for instance by using QoS modelling languages like CQML. One may also envisage tools that generate code for inspector components from specifications. Mappings may be somewhat dependent of platforms and may not be trivial. It is also important that the meanings of expressions are consistent and compatible amongst heterogeneous and autonomous participants. As discussed in section 4.5, there are interesting interoperability issues and it may be relevant to see if ontologies as used with the semantic web can be of use in defining profile-models.

8.3.3. Other issues

When considering our results in general, there are several interesting issues for further research. Some of these are obviously related to validating and generalising our results. It is particularly relevant to investigate alternative binding types or interaction paradigms like publish/subscribe, peer-to-peer, etc. Applying the results to different application domains would strengthen a claim that they are general purpose or identify limitations. Possible issues for further research include:

- *Transactions.* Some researchers propose that extra-functional properties of transactions (traditionally referred to as ACID) should be negotiable (e.g. [Karlsen03, Arntsen05]). We may take this idea further and view transaction as a unit of computation for which several QoS dimensions are negotiable. Here, we may evaluate a transactional binding type supporting multiple servers.
- *Scalability.* Further investigate the idea of nested contracts as discussed in section 7.2.5 where a binding with multiple parties corresponds both to a global contract for all parties, and local contracts for each party. This may be further generalised to a hierarchy. We may investigate how our approach can be combined with the hierarchical management model of [Ecklund02].
- *Policy concept.* Evaluate how to use adaptation in policies or automatic generation of policies as discussed in section 8.2 above.

Bibliography

Aagedal01	J.Ø. Aagedal, *Quality of Service Support in Development of Distributed Systems*, Ph.D. Thesis, University Oslo, 2001.
Abadi93	M. Abadi, L. Lamport, *Conjoining Specifications*, Digital Systems Research Center, Report 118, Palo Alto, 1993.
Abadi94	M. Abadi, L. Lamport, Open Systems in TLA, *Proc. ACM Symposium on Principles of Distributed Computing, August 1994*, p. 81-90.
Almedia04	J.P.A. Almeida, M.J. van Sinderen, L. Ferreira Pires. The role of RM-ODP computational viewpoint concepts in the MDA approach. *Workshop on ODP for Enterprise Computing WODPEC 2004*, pp. 28-35, September 2004.
Anane07	R. Anane, Autonomic Behaviour in QoS Management, *Proc. International Conference on Autonomic and Autonomous Systems*, IEEE, 2007.
Andersen01	A. Andersen, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulø, W. Yu, *Arctic Beans: Configurable and Reconfigurable Enterprise Component Architectures*, IEEE Distributed Systems Online, Vol. 2, No. 7, 2001.
Andersen02	A. Andersen, *OOPP - A Reflective Middleware Platform including Quality of Service Management*, Ph.D. Thesis, Dept. of Computer Science, University of Tromsø, 2002.
Anderson91	D.P. Anderson, R.G. Herrtwich, C. Schaefer, *SRP: A Resource Reservation Protocol for Guaranteed Performance Communication in the Internet*, Internal Report, University of California at Berkeley, 1991.
Arntsen05	A-B. Arntsen, R. Karlsen, ReflectTS: A Flexible Transaction Service Framework, in *Proc. Reflective and Adaptive Middleware' 05*, ACM, December 2005.
ASJ	*AspectJ home page*: http://eclipse.org/aspect.
Atkinson87	M.P. Atkinson, O.P. Buneman, Types and Persistence in Database Programming Languages, *ACM Computing surveys, Vol. 19, No. 2*, June 1987.
Aurre96	C. Aurrecoechea, A.T. Campbell, L. Hauw, A Survey of QoS architectures, *Multimedia Systems Journal, special issue on QoS architectures*, 1996.
Banga99	G. Banga, P. Druschel, J.C. Mogul, Resource Containers: A new facility for resource management in server systems, in *proc. 3rd USENIX Symposium on Operating Systems Desing and Implementation*, pp. 45-56, February 1999.

Bearman93	M.Y. Bearman, ODP Trader, in *Proc. ICODP'93*, September 1993.
Beaton97	G. Beaton, A Feedback-Based Quality of Service Management Scheme, in *Proc. HIPPARCH Workshop*, June 1997.
Becker97	C. Becker, K. Geihs, MAQS - Management for Adaptive QoS-enabled Services, in Proc. *IEEE Workshop on Middleware for Distributed Real-Time Systems and Service*s, San Fransisco, 1997.
Bergmans00a	L . Bergmans, A. van Halteren, M. van Sinderen, M. Aksit, A QoS-Control Architecture for Object Middleware, in *Proc. 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pp. 117 - 131, LNCS, 2000.
Bergmans00b	L. Bergmans, M. Aksit, Ascpects and crosscutting in layered middleware systems, in *Proc. IFIP/ACM Workshop on Reflective Middleware* (Middleware 2000), April 2000, pp. 23-25.
Bergmans01	L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, *Communications of the ACM, October 2001*, Vol. 44, No. 10.
Bertino95	E. Bertino, P. Foscoli, *Index Organizations for Object-Oriented Database Systems*, IEEE Transactions on knowledge and data engineering, vol. 7, no. 2, April 1995.
Beugnard99	A. Beugnard, J-M. Jézéquel, N. Plouzeau, D. Watkins, Making Components Contract Aware, *IEEE Computer,* July 1999.
Bickmore97	T.W. Bickmore, B.N. Schilit, Digestor: Device-independent Access to the World Wide Web, in *Proc. 6th International World Wide Web Conference*, April 1997.
Black87	A. Black, N. Hutchison, E. Jul, H. Levy, L. Carter, Distribution and Abstract Types in Emerald, *IEEE trans. on Software Engineering,* Vol. SE-13, No. 1, January 1987.
Blair97	G. S. Blair, J-B. Stefani, *Open Distributed Processing and Multimedia*, Addison Wesley 1997.
Blair98	G.S. Blair, G. Coulson, P. Robin, M. Papathomas, An Architecture for Next Generation Middleware, in P*roc. Middleware '98*, Springer 1998.
Blair00	G.S. Blair, A. Andersen, L. Blair, G. Coulson, D. Sánchez, Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform, in *IEE proceedings, Software*, 148(03), June 2001.
Blair01	G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, K. Saikoski, The design and implementation of Open ORB 2, *IEEE Distributed Systems Online 2* (2001), no. 6.
Blair04	G.S. Blair, G. Coulson, P. Grace, Research Directions in Reflective Middleware: the Lancaster Experience, in *Proc. RM 2004*, October 2004.
Blazewicz86	J. Blazewicz, W. Cellary, R. Slowinski, J. Weglarz, Scheduling under resource constraints - Deterministic Models, *Annals of Operations Research*, Volume 7, 1986.

Bruno98	J. Bruno, E. Gabber, B. Ozden, A. Silberschatz, The Eclipse operating system: Providing quality of service via reservation domains, in *Proc. USENIX annual technical conference*, June 1998, pp. 235-246.
Butler02	M.H. Butler, *CC/PP and UAProf: Issues, Improvements and Future Directions*, Hewlett Packard Labs Technical Report, HPL-2002-35.
Campbell96	A.T. Campbell, *A Quality of Service Architecture, Ph.D. thesis*, Computing Department, Lancaster University, 1996.
Candea98	G.M. Candea, M.B. Jones, *Vassal: Loadable Scheduler Support for Multi-Policy Scheduling*, Microsoft Research Technical Report MSR-TR-98-30, August 1998.
Capra03	L. Capra, W. Emmerich, C. Mascolo, CARISMA: Context-aware reflective middleware for mobile applications, in *IEEE transaction on software engineering*, Vol. 29, No. 10, 2003, pp. 929-945.
Cherka02	L. Cherkasova, P. Phaal, Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites, In *IEEE Transactions on computers*, Vol. 51, No. 6, June 2002.
Chiba00	S. Chiba, Load-time Structutal Reflection in Jav*a*, in Proc. ECOOP'00, LNCS 1850 pp. 313-336, Springer 2000.
Chu97	H. Chu, K. Nahrstedt, A Soft Real Time Scheduling Server in UNIX Operating System, in *Proc. European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, September 1997.
Chu99	H. Chu, *CPU Service Classes: A Soft Real Time Framework for Multimedia Applications, PhD thesis, Department of Computer Science*, University of Illinois at Urbana Champaign, August 1999.
Clarke01	M. Clarke et. al., An Efficient Component Model for the Construction of Adaptive Middleware*, in Proc. Middleware '01*. 2001.
Coulson02	G. Coulson, G.S. Blair, M. Clark, N. Parlavantzas, The Design of a Configurable and Reconfigurable Middleware Platform, in *ACM Distributed Computing Journal*, Vol 15, No 2, pp. 109-126, April 2002.
Coulson04	G. Coulson, G.S. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, OpenCOM v2: A Component Model for Building Systems Software, in *Proc. IASTED Software Engineering and Applications*, November 2004.
CUP	*CUP: LR Parser Generator in Java*. http://www2.cs.tum.edu/projects/cup/
Daniel99	J. Daniel, B. Traverson, S. Vignes, Integration of Quality of Service in Distributed Object Systems, in *Proc. DAIS'99*, Helsinki 1999. pp. 31-43.
Denning89	P. J. Denning, D. Comer, D. Gries, M. C. Mulder, A. B. Tucker, A. J. Turner, P. R. Young. Computing as a discipline, *Communications of the ACM* (CACM), 32(1):9, 1989.
Deschrevel93	J.P. Deschrevel, *The ANSA Model for Trading and Federation*, ANSA Phase III Technical Report, APM.1005.01, July 1993.

Devara02	M. Devara, D. Cohen, *HTB Linux queuing discipline manual - user guide*, http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm (Last updated 5.5.2002).
Dini97	P. Dini, A. Hafid, Towards Automatic Trading of QoS Parameters in Multimedia Distributed Applications, in *Proc. ICODP'97*.
Donaldson98	D. Donaldson, et. al., DIMMA - A Multi-Media ORB, in *Proc. Middleware '98*, Springer Verlag, 1998.
Dumant98	B. Dumant, F. Horn, F. Dang Tran, J.B. Stefani, Jonathan: an Open Distributed Processing Environment in Java, in *Proc. Middleware '98*, Springer Verlag, 1998.
Ecklund01	D. Ecklund, et. al., QoS Management Middleware: A Separable, Reusable Solution, in *Proc. 8th Int. Workshop in Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS 2001)*, pp. 124-137, LNCS, 2001.
Ecklund02	D.J. Ecklund, V. Goebel, T. Plagemann, E.F. Ecklund, Dynamic end-to-end QoS management middleware for distributed multimedia systems, *Multimedia Systems 8*, pp. 431-442, 2002.
Eide03	V.S.W. Eide, F. Eliassen, O.Lysne, O-C. Granmo, *Extending Content-based Publish/Subscribe Systems with Multicast Support*, Simula Research lab. Technical Report 2003-03, July 2003.
Eide05	V.S.W. Eide, F.Eliassen, J.A. Michaelsen, Exploiting Content-Based Networking for Fine Granularity Multi-Receiver Video Streaming, In *Multimedia Computing and Networking 2005*, SPIE Vol. 5680, 2005.
Eliassen98	F. Eliassen, S. Mehus, *Type Checking Stream Flow Endpoints*, in *Proc. Middleware '98*. Springer Verlag, September 1998, pp. 305-322.
Eliassen99	F. Eliassen, A. Andersen, G.S. Blair, F. Costa, G. Coulson, V. Goebel, Ø. Hanssen, T. Kristensen, T. Plagemann, H.O. Rafaelsen, K.B. Saikoski, W. Yu, Next Generation Middleware: Requirements, architecture and prototypes, in *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems* (FT-DCS '99), December 1999.
Eliassen06	F. Eliassen, E. Gjørven, V.S.W. Eide, J.A. Michaelsen, Evolving Selv-Adaptive Services using Planning-Based Reflective Middleware, In *Proc. Adaptive and Refletive Middleware 06*, November 2006.
Eugster03	P.Th. Eugster, P.A. Felber, R. Guerraoui, A.M. Kermarrec, The Many Faces of Publish/Subscribe, in *ACM Computing surveys*, Vol. 35, Issue 2, 2003, pp. 114 - 131
Evans04	J. Evans, C. Filsfils, Deploying Diffserv at the Network Edge for Tight SLAs (part 1 and part 2), *IEEE Internet Computing,* January/February 2004 and March/April 2004.
Fitzpatrick98	T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, P. Robin, Supporting Adaptive Multimedia Applications Through Open Bindings, in *Proc. International Conference on Configurable Distributed Systems*, 1998.

Foster99	I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy, A distributed resource management architecture that support advance reservation and co-allocation, in *Proc. IEEE/IFIP Intl. Workshop on QoS* (IWQoS), 1999.
Foster00	I. Foster, A. Roy, V. Sander, A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation, in *Proc. IEEE/IFIP Intl. Workshop on QoS (IWQoS)*, 2000.
Fox96	A. Fox, E. Brewer, Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation, in *Proc. 5th International World Wide Web Conference*, May 1996.
Frølund98a	S. Frølund, J. Koistinen, Quality-of-Service Specification in Distributed Object Systems, in *Distributed Systems Engineering Journal.* Vol. 5. 4, pp. 179-202.
Frølund98b	S. Frølund, J. Koistinen, *Quality of Service Aware Distributed Object Systems,* Hewlett Packard Software Technology lab. report: HPL-98-142.
Frølund98a	S. Frølund, J. Koistinen, Quality-of-Service Specification in Distributed Object Systems, *Distributed Systems Engineering Journal*. Vol. 5. 4, pp. 179-202.
Frølund98b	S. Frølund, J. Koistinen, *Quality of Service Aware Distributed Object Systems,* Hewlett Packard Software Technology lab. report: HPL-98-142.
Gamma95	Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison Wesley 1995.
Geihs01	K. Geihs, Middleware Challenges Ahead, *IEEE Computer, Vol 36, No 6*, pp. 24-31, June 2001.
Goczyla97	K. Goczyla, The partial-Order Tree: A New Structure for Indexing on Complex Attributes in Object-Oriented Databases, in *Proc. Euromicro 1997*, IEEE press.
Goel99	A. Goel, D. Steere, C. Pu, J. Walpole, *Adaptive Resource Management Via Modular Feedback Control*, Oregon Graduate Institute, Computer Science and Engineering, Technical Report 99-03, January 1999.
Goh97	C.H. Goh., *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Sources*, Ph.D. thesis, MIT, 1997.
Gourdarzi99	K. Moazami-Goutdarzi, *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, Ph.D. thesis, Imperial College, London, March 1999.
Hanssen98	Ø. Hanssen, F. Eliassen, Towards a QoS aware Binding Model, In *Proceedings of SYBEN '98*, Zurich, Spie press, 1998.
Hanssen99	Ø. Hanssen, F. Eliassen, A Framework for Policy Bindings, In *Proceedings of Distributed Object and Applications '99*, IEEE press, 1999.
Hanssen00	Ø. Hanssen, F. Eliassen, Policy Trading, In *Proceedings of Distributed Objects and Applications '00*, IEEE press, 2000.
Hanssen05a	Ø. Hanssen, Towards Declarative Characterisation and Negotiation of Bindings, in *Proc. Adaptive and Reflecive Middleware 2005*. ACM, 2005.
Hanssen05b	Ø. Hanssen, *A Declarative Profile model for QoS negotiation,*Technical report 2005-54, University of Tromsø, Computer Science Department, 2005.

Hauck01 F.J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, M. Steckermeier, AspectIX: a quality aware, object-based middleware architecture, in *Proc. DAIS'2001*, Kluwer, September 2001.

Hayden97 M. Hayden, *The Ensemble System*, Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1997.

Hayton99 R. Hayton et. al. *FlexiNet Architecture report*, ANSA Phase III Technical Report, Feburary 1999.

Hayton00 R. Hayton, A. Herbert, FlexiNet: A Flexible, Component-Oriented Middleware System, in *Lecture Notes in Computer Science*, 1752, p. 497 ff., Springer Verlag, February 2000.

Hutchinson88 N. Hutchinson, L. Peterson, Design of the X-kernel*, ACM SIGCOMM Computer Communication Review*, Vol. 18, 4. August 1988.

ISO95 *QoS - Basic Framework*, Report: ISO/IEC JTC/SC21, N9309, 1995.

ISO95b *Open distributed processing reference model, part 1: Overview,* ITU-T X.902 - ISO/IEC 10746-1, 1995.

ISO97 *ODP Trading Function,* Report: ITU-T X.950 - ISO/IEC 13235.

ISO98 *Working Draft for Open Distributed Processing - Reference Model - Quality of Service*, Report: ISO/IEC JTC 1/SC 21 N 10979 Ed 6.4, Berlin.

Jaeger04 M.C. Jaeger, G. Rojec-Goldmann, G. Mühl, QoS Aggregation for Web Service Composition using Workflow Patterns, in *Proc. 8th IEEE Enterprise Object Computing Conference* (EDOC 04), 2004.

Jin04 Jin, K. Nahrstedt, Classification and Comparison of QoS Specification Languages for Distributed Multimedia Applications, in *IEEE MultiMedia,* Volume 11, Issue 3, pp. 74-87, July 2004.

Jing99 J. Jing, A. Helal, A. Elmagarmid, Client-Server Computing in Mobile Environments, *ACM Computing Surveys*, Vol. 31, No. 2, June 1999.

Jones83 C.B. Jones, Specification and Design of (parallel) programs, in *Proc. Information Processing, 1983*, IFIP 9th World Congress, pp. 321-332.

Jones96 M. Jones, J. Alessandro, F. Paul, J. Leach, D. Rou, M. Rou, An overvivew of the Rialto real-time architecture, in *Proc. 7th ACM SIGOPS European Workshop*, September 1996.

J2EE Sun Microsystems, *Java 2 Enterprise Edition 1.4*. Documentation, (http://java.sun.com).

Karlsen03 R. Karlsen, An Adaptive Transactional System - framework and service synchronization, in *Proc. Distributed Objects and Applications '03*, November 2003.

Kelly00 F. Kelly, P. Key, S. Sachary, Distributed Admission Control, in *IEEE Journal on Selected areas in Communications*, 2000.

Kiczales91 G. Kiczales, J.J. des Rweres, D. Bobrow, *The art of the metaobject protocol*, MIT press 1991.

Kiczales96a	G. Kiczales, Beyond the Black Box: Open Implementation, *IEEE Software* 1996.
Kiczales96b	G. Kiczales, J. Irwin, J. Lamping, J-M. Loingtier, C.V. Lopes, C. Maeda, A. Mendhekar, Aspect-Oriented Programming, in *ACM Computing Surveys 28* (4es), 1996.
Kim91	W. Kim, J. Seo, Classifying schematic and data heterogeneity in multidatabase systems, in *IEEE Computer, 24(12)*: 12-18, 1991.
Koistinen98	J. Koistinen, A. Seetharaman, Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures, in *Proc. 2nd Enterprise Distributed Object Computing Workshop* (EDOC'98). San Diego, 1998. pp. 239-249.
Kon98	F. Kon, A. Singhai, R.H. Campbell, D. Carvalho, R. Moore, F. Ballesteros, 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments, in *Proc. ECOOP'98 Workshop on Reflective Object Oriented Programming and Systems*, July 1998.
Kon00	F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, R. Campbell, Monitoring, Security and dynamic configuration with the dynamicTAO reflective ORB, in *Proc. IFIP/ACM Middleware 2000*, pp. 121-143.
Kon02	F. Kon, F. Costa, G. Blair, R.H. Campbell, The Case for Reflective Middleware, *Communications of the ACM*, June 2002, Vol. 44, No. 6.
Kumar92	V. Kumar, Algorithms for Constraint Satisfaction Problems: A Survey, *AI Magazine*, Vol 13, No 1, pp. 32-44, 1992.
Lazar94	A.A. Lazar, S. Bhonsle, K.S. Lim, A Binding Architecture for Multimedia Networks, in *Proc. COST-232 Conf. on Multimedia Transport and Teleservices*, Vienna, 1994
Lee99	C. Lee, J. Lehoczky, R. Rajkumar, J. Hansen, A scalable solution to the multi-resource QoS problem, in *Proc. IEEE Real-Time Systems Symposium* (RTSS'99), 1999.
Li94	G. Li, D. Otway, *ANSA Real-Time QoS Extensions*, ANSA Phase III technical report APM.1094.00.06.
Li99	B. Li, K. Nahrstedt, A Control-Based Middleware Framework for Quality of Service Adaptation, *IEEE JSAC, Special Issue on Service Enabling Platforms,* vol 17, no. 9, September 1999, pp. 1632-50.
Li05	G. Li. H. Jacobsen, Composite Subscriptions in Content-Based Publish/Subscribe Systems, in *Proc. ACM/IFIP Middleware 2005*, LNCS 3790, Springer.
Loughran05	N. Loughran, et. al., *Survey of Aspect-oriented Middleware*, AOSD-Europe. WP9/D8 report, version 1.0, June 2005.
Loyall98a	J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.E. Bakken, Specifying and measuring Quality of Service in Distributed Object systems, in *Proc. ISORC'98*, IEEE, April 1998.

Loyall98b	J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, K. R. Anderson, QoS Aspect languages and Their Runtime Integration, in Proc. LNCS'98, Springer 1998.
McAffer96	J. McAffer, Meta Level Architecture Support for Distributed Objects, in *Proc. Reflection'96*, 1996, pp. 39-62.
Miguel02	M.A. de Miguelm J.F. Ruiz, M. García-Valls, QoS-Aware Component Frameworks, in *Proc. IEEE/IFIP Intl. Workshop on QoS* (IWQoS), 2002.
Mercer94	C.W. Mercer, S. Savage, H. Tokuda, *P*rocessor Capacity Reserves: Operating system support for multimedia applications, in *Proc. IEEE international Conference on Multimedia Computing and Systems*, May 1994, pp. 90-99.
Molenkamp02	G. Molenkamp, H. Lutfiyya, M. Kachabaw, M. Bauer, Diagnosing Quality of Service Faults in Distributed Applications, in *Proc. 21st IEEE International Conference on Performance, Computing and Communications 2002*, pp. 375-382. April 2002.
Nahrstedt95	K. Nahrstedt, J. Smith, The QoS Broker*, IEEE Multimedia*, spring 1995.
Nahrstedt00	K. Nahrstedt, D. Wichadakul, D. Xu, Distributed QoS Compilation and Runtime Instantiation, in *Proc, 8th IEEE/IFIP international workshop on QoS*, June 2000, pp. 198-207.
Nahrstedt01	K. Nahrstedt, D. Xu, D. Wichadakul, B. Li, QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments, *IEEE Communications Magazine,* November 2001.
Nagar04	S. Nagar et al., Improving resource control using CKRM, In *Proceedings of the Linux Symposium, Volume Two*, Ottawa, July 2004.
Najm95	E. Najm, J-B. Stefani, A Formal Semantics for the ODP Computational Model, *Computer Networks and ISDN Systems*, Vol. 27, 1995, pp. 1305-1329.
OMA03	Open Mobile Alliance, *User Agent Profile Version 2.0*, OMA-UaProf-v2_0-20030520-C, May 2003.
OMG96	Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0. 1996.
OMG00	Object Management Group, *Trading Object Service*, Version 1.0, 2000-06-27
OMG02	Object Management Group, *CORBA Component Model*, Revision 3.0. 02-06-65.
Opyrchal	L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, D. Sturman, Exploiting IP Multicast in Content-Based Publish-Subscribe Systems, In *Proc. of Middleware 2000*, LNCS 1795, pp. 185-207, Springer 2000.
Otway87	D. Otway, E. Oskiewicz, REX: a remote execution protocol for object-oriented distributed applications, in *Proc. ICDCS 1987*, IEEE Press 1987, pp. 113-119.
Parlav03	A. Parlavantzas, G. Coulson, G.S. Blair, An extensible Binding Framework for Component-Based Middleware, In proceedings of EDOC 2003, pp. 252-263.

Plagemann94	T. Plagemann, *A Framework for Dynamic Protocol Configuration*, Ph.D. thesis, ETH Zurich, no. 10830, 1994.
Plagemann00	T. Plagemann, F. Eliassen, B. Hafskjold, T. Kristensen, R.H. Macdonald, H.O. Rafaelsen, Managing Cross-Cutting QoS Issues in MULTE Middleware, (extended abstract), *Workshop on Quality of Service in Distributed Object Systems* (QoSDOS), in association with 14th European Conference on Object-Oriented Programming (ECOOP 2000), June 2000.
Rafaelsen00	H.O. Rafaelsen, F. Eliassen, Trading and Negotiating Stream Bindings, in *Proc. IFIP/ACM Middleware'2000*, April 2000.
Rafaelsen02	H.O. Rafaelsen, F. Eliassen, Design and performance of a media gateway trader, on *Proc. International Symposium on Distributed Objects and Applications* (DOA'02), October 2002, pp. 675-692.
Raikumar98	R. Raikumar, K. Juvva, A. Molano, S. Oikawa, Resource kernels: A resource-centric approach to real-time systems, in *Proc. SPIE/ACM conference on Multimedia Computing and Networking*, pp. 150-164.
Rajahalme97	J. Rajahalme, T. Mota, F. Steegmans, P.F. Hansen, F. Fonseca, Quality of Service Negotiation in TINA, in *Proc. Global Convergence of Telecommunications and Distributed Object Computing*, TINA'97, pp. 278-286.
Regehr01	J. Regehr, J.A. Stankovic, HLS: A Framework for Composing Soft Real-Time Schedules, in *Proc. 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, December 2001.
RFC1633	B. Braden, S. Schenker, D. Clark, Integrated Services in the Internet Architecture. an Overview, RFC 1633, IETF IntServ Working Group.
RFC2475	S. Blake et. al., An Architecture for Differentiated Services, RFC 2475, IETF DiffServ Working Group.
Roscoe95	T. Roscoe, *The structure of a multi-service operating system*, PhD thesis, Computer Laboratory, Cambridge University, April 1995.
Rothermel97	K. Rothermel, G. Dermler, W. Fiederer, *QoS Negotiation and Resource Reservation for Distributed Multimedia Applications*, Proc. IEEE International Conference on Multimedia Computing and Systems'97, pp. 319-326.
Ruttkay98	Z. Ruttkay, Constraint Satisfaction, - a survey, *CWI Quarterly*, Vol 11, 2-3, pp-163-214.
RTJ	G. Bollella et. al., *The Real-Time Specification for Java*, Addision Wesley, 2000, ISBN 0-201-70323-8.
Schmidt97	D.C. Schmidt, A. Gokhale, T.H. Harrison, D. Levine, C. Cleeland, *TAO: a High-Performance Endsystem Architecture for Real-time CORBA*, RFI response to the OMG Special Interest Group on Real-time CORBA, 1997.
Sedgewick	R. Sedgewick, *Algorithms in Java*, third edition. Part 5, Pearson 2003.

ShLa90 A. Sheth, J. Larson, Federated Database Systems for managing Heterogeneous and Autonomous Databases, *ACM Computing Surveys,* Vol. 22, No. 3, September 1990.

Siqueira99 F. Siqueira, V. Cahill, Delivering QoS in Open Distributed Systems, in *Proc, 7th IEEE Workshop on Future Trends in Distributed Computing Systems*, 1999.

Siqueira00 F. Siqueira, V. Cahill, Quartz: A QoS Architecture for Open Systems, in *Proc, 20th IEEE International Conference on Distributed Computing Systems*, April 2000.

Smith82 B.C. Smith, *Procedural Reflection in programming languages*, Ph.D. thesis, Massachusetts Institute of Technology, 1982.

Solberg04 A. Solberg, S. Amundsen, J.Ø. Aagedal, F. Eliassen, A Framework for QoS-Aware Service Composition, in *Proc. 2nd. ACM Intl. Conference on Service Oriented Computing*, ICSOC 2004.

Sommer02 N.L. Sommer, F. Guidec, A Contract-Based Approach of Resource-Constrained Software Deployment, *IFIP/ACM Working Conference on Component Deployment*, 2002.

Sommer04 N.L. Sommer, Towards a Dynamic Resource Contractualisation for Software Components, *IFIP/ACM Working Conference on Component Deployment*, 2004.

Staehli02 R. Staehli, F. Eliassen, *A QoS Aware Component Architecture*, Simula Research Laboratory report, 2002-12.

Staehli04a R. Staehli, F. Eliassen, S. Amundsen, Designing Adaptive Middleware for Reuse, in *Proc. Reflective and Adaptive Middleware' 04*, ACM, October 2004.

Staehli04b R. Staehli, F. Eliassen, Compositional Quality of Service Semantics, in *proc. SAVCBCS '04*, ACM, October 2004.

Szyperski98 C. Szypersk*i, Component Software: Beyond Object Oriented Programming*, Addison Wesley, 1998.

Thi06 D. Thißen, P. Wesnarat, Considering QoS Aspects in Web Service Composition, in *Proc. 11th IEEE Symposium on Computers and Communications*, IEEE, 2006.

Tina94 *Quality of Service framework*, TINA-C, Report: TP_MRK.001_1.0_94.

Tina96 *TINA Object Definition Language Manual*, TINA-C, Report: TP_NM.002_2.2_96.

Topolcic90 C. Topolcic, *Experimental Internet Stream Protocol*, *Version 2 (ST-II)*, Internet Request for Comments, RFC-1190, 1990.

Vasud98a V. Vasudevan, *A Reference Model for Trader-Based Distributed Systems Architectures*, Object Services and Consulting Inc. http://www.objs.com/survey/trader-reference-model.html

Vasud98b V. Vasudevan, Augmenting OMG traders to handle service composition, Object Services and Consulting Inc.
http://www.objs.com/survey/compositional-trader.html

Wache01	H. Wache. T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, S. Hübner, Ontology-Based Integration of Information - A Survey of Existing Approaches, in *Proceedings of IJCAI-01 Workshop, Ontologies and Information*, pp. 108-117, Sharing, 1997.
Waddington97	D. Waddington, C. Edwards, D. Hutchison, Resource Management for Distributed Multimedia Applications, In *proc. ECMAST'97*, May 1997.
Wang00	N. Wang, M. Kircher, D.C. Schmidt, Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation, in *Proc. COMPSAC 2000 conference*, October, 2000.
WComp	Ø. Hanssen, *Web Component Scripting Framework*, http://www.wcomp.cs.uit.no.
Wegdam03	M. Wegdam, *Dynamic Reconfiguration and Load Distribution in Component Middleware*, Ph.D. thesis, University of Twente, Telematica Instituut Fundamental Research Series, No. 009 (TI/FRS/009)
Wichadakul01	D. Wichadakul, K. Nahrstedt, X. Gu, D. Xu, 2KQ+: An Integrated Approach of QoS Compilation and Reconfigurable Component-Based Run-time Middleware for the Unified QoS Management Framework, in *Proc. Middleware '01*, November 2001.
W3C-04a	*Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0*, W3C Recommendation 15 January 2004, http://www.w3c.org/TR/2004/REC-CCPP-struct-vocab-20040115/
W3C-04b	*Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C Recommendation 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
Xu01	D. Xu, K. Nahrstedt, D. Wichadakul, QoS and Contention-Aware Multi-Resource Reservation, *Cluster Computing 4*, 2001, pp. 95-107.
Yokote92	Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, in *Proc. OOPSLA'92*, ACM, October 1992, pp. 414-434.
Zeng04	L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang, QoS-aware middleware for web service composition, *IEEE Transactions on software engineering, 30(5)*, pp. 311-307, 2004.
Zhang93	L. Zhang, S. Deerling, D. Estrin, S. Schenker, D. Zappala, RSVP:A New Resource ReSerVation Protocol*, IEEE Network*, vol. 7, no. 5, september 1993.
Zinky97	J.A. Zinky, D.E. Bakken, R.E. Schantz, Architectural Support for Quality of Service for CORBA Objects, *Theory and Practice of Object Systems*, Wiley, April 1997.