Towards Declarative Characterisation and Negotiation of Bindings

Øyvind Hanssen
University of Tromsø
Department of Computer Science
9037 Tromsø
+47 95117457
ohanssen@acm.org

ABSTRACT

This paper addresses negotiation of bindings in open systems, and in particular how to characterise the capabilities of heterogeneous platforms, and communication channels. Based on a middleware architecture supporting policy-governed binding, negotiation is about selecting suitable policies for bindings at run-time. We propose a model for declarative expressions based on a hybrid of declared and rule-based conformance, and composition operators. We also propose a scheme for how the middleware can support automatic characterisation of resources or other relevant capabilities and composition of these, based on the declarative expression model.

Categories and Subject Descriptors

D.2.1 **[Software Engineering]**: Requirements/Specifications -- *languages*. D.2.12 **[Software Engineering]**: Interoperability -- *distributed objects*.

General Terms

Experimentation, Languages.

Keywords

Quality of Service, Negotiation, Binding, Trading, Middleware.

1.INTRODUCTION

In the last decade much attention has been turned towards middleware which supports dynamic adaptation to non-functional application requirements and varying environmental conditions. This is motivated by requirements for e.g. multimedia applications, mobility, dependability, etc. The capabilities of platforms on which to build open and distributed applications are also increasingly diverse. Platforms may offer different types and amounts of resources for computing and communication, as well as different mechanisms to manage them. The approach of middleware providing one single abstraction, hiding implementation details and differences between the various platforms is recognized to be too limiting. Therefore, research has been focusing on opening up and componentising the middleware, to make it more configurable, but still without sacrificing abstraction.

Reflective middleware [1] explores the idea of using meta-level architectures for exposing implementation details, and using meta object protocols for programmatic access to these. It is however less clear how to support automatic adaptation to various QoS requirements and environmental properties. Binding between components would involve a negotiation process, which involves exchanging requirements and offers, to reach agreement on a contract and to find a solution on how to configure the binding accordingly. This means that we need not only platform abstraction, but also platform awareness, which is the ability to characterise and exchange the properties of the platform. Systems which have such expression and negotiation capabilities with respect to non-functional properties are often termed Quality of Service Aware (c.f. [2]). However QoS research has mostly focused on static specification or dynamic negotiation tied to specific architectures.

This paper addresses how to expose varying platform capabilities in a way that facilitates negotiation between heterogeneous platforms on how to select suitable policies for bindings. The main contributions are: (1) A proposed model for declarative expressions. (2) A proposed scheme for how the middleware can support characterisation of resources or other relevant capabilities, as well as composition of these.

In section 2 we give an overview of the main ideas of our approach: Policy bindings negotiation and the need for a language for stating QoS requirements, and properties of the environment, and which supports conformance checking and composition. In section 3 we introduce our profile expression language. In section 4 we introduce our ideas for a negotiation support in our experimental middleware architecture. This includes dynamic profile expressions and a descriptor object framework. In section 5 we relate to current work in the area and in section 6 we conclude.

2. OVERVIEW

The main ideas of our approach, and the basis of our investigations are as follows:

- The concept of *policy* which define contract templates and contract enforcement plans. The concept of *metapolicy*, which define the management of bindings and associated policies [3].
- *Trading* of policy as a principle of negotiation and the use of declared conformance for matching property descriptions [4]. A language for *profile-expressions* used for exchanging requirements and environmental properties, and which support conformance checking and composition.
- Run-time expression support by the middleware.

2.1 Negotiation

We are interested in how to find a contract and a corresponding configuration when establishing a binding. We refer to such a process as negotiation since an overall goal is to reach agreement between possibly autonomous parties and since it may involve exchange of statements (offers and requirements).

Figure 1 illustrates which roles profile expressions play in negotiation based on policy trading. For a given service, application or application domain, there would exist a set of potential contracts, called policies, each stating an offer and an expectation. A contract is a promise, that the properties offered will be provided as long as the expectation is satisfied. The goal of negotiation is to find a policy whose offer (user profile) satisfies the user requirement while its expectation (service-profile) is satisfied by the environment properties (environment descriptor).

The relationships between user requirements and offers, and the relationships between the expectations and the capabilities of the environment, are satisfaction relationships. To facilitate conformance testing, a model should define a *partial order* on such expressions with respect to satisfaction. Then, any pair of expressions may be mechanically evaluated for conformance.

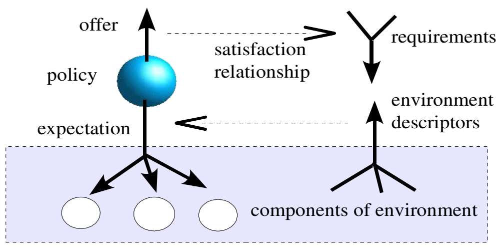

Figure 1. Statements and satisfaction relationships

2.1.1 Declared conformance

It looks appealing to adopt the technique typically used in ODP trading [5, 6] where each requirement or offer is a reference to a type name, and where a type conformance graph is declared a priori. This way of using declared conformance was first proposed in [7]. However, this is too limiting in general, since each declared type will need to capture all aspects relevant for the application. This may easily lead to conformance graphs which are too complex and application specific. Therefore we propose a hybrid model where conformance rules may also be based on simple numeric parameters.

2.1.2 Dynamic composition of statements

Profile expressions and negotiation should support *composition*, since statements from participants which do not necessarily know each other, would need to be combined into one describing the composed system. Given a set of expressions about the behaviour of individual components of a system, it is not obvious how to deduce the behaviour of the whole system. Three different problems should be addressed when it comes to expressing the total behaviour:

- Autonomous users may issue different requirements for the same service and all users should be satisfied.
- We may need to combine expressions regarding the same component but in more than one dimension (e.g. performance and security). We introduce an operator to construct expressions from simpler sub-expressions, meaning that the predicates stated by each part must be true in the same environment (c.f. logical conjunction).
- Open systems are systems interacting with environments neither they or their implementers controls [8]. Expectations towards environments may need to characterise a number of abstract components, for instance, client, server and communication channel, with a separate expectation for each of them. Our model should therefore support dynamic composition of statements about separate components of the environment. We address this problem by introducing an additional composing operator.

The third problem is only partly addressed in QoS specification models like [9], by allowing QoS-characteristics to be defined with composition in mind. Work on formal models has shown that with certain assumptions on the temporal relationships [10] it is possible to make statements about the behaviour of composite systems as conjunctions of statements about each component.

2.2 Binding model

The negotiation scheme discussed above need to be supported by a middleware architecture, it will be a part of the binding establishment process, where an active binding would represent a contract. The basis of our investigation is the family of binding models of ANSA [12], FlexiBind [3], OpenORB [13, 14], etc.

We believe that [14] is suitable as a generic binding model which regards binding-types as pluggable first-class entities. In our current experimental work, we assume a client/server (RMI) special case, and look at how client initiated binding would lead to a session specific end-to-end configuration. We also limit the scope of negotiation to the non-functional aspects. However we believe that the ideas explored here are applicable to other binding types as well.

2.2.1 Binding phases

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

- *Service deployment* (server side binding). A 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 negotiations.
- *Client binding*, i.e. a client associates to the service. This would not necessarily lead to a complete configuration, since there may still be parts which need to be negotiated.
- *Activation*, where binding configuration as a result of negotiation is completed, and associated with necessary resources such that invocation may take place.
- 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 (c.f. CCM, or EJB) and service deployment. Component deployment may involve service deployment.

2.2.2 Policy

A *policy* represents a potential *contract,* i.e. it can be viewed as a mapping from some constraint on the environment *S*, to the satisfaction of an user requirement U. We refer to *U* as the *user profile* and S as the *service profile*. If *P(x)* is the predicate defining a profile *x*, a policy states the following: $P(S) \Rightarrow P(U)$

A policy also constrains how an activation is configured. The configuration part of a policy will depend on the binding type. For RMI bindings it will consist of a client and a server part.

A *metapolicy* represent a way to associate policies with a given binding. A binding will always be associated with a metapolicy which constrain 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 capture how services are set up in the deployment phase as well as in the client binding phase. A metapolicy may therefore involve implementation decisions which constrain the later choice of policy.

3. PROFILE MODEL

In our approach statements about offers, expectations, etc. are formulated as *profile expressions* which can be evaluated for conformance. In this section we describe the idea of *basic profile models* and how more complex expressions can be composed from *basic profiles* by using *sum* or *component-sum* operators.

3.1 Defining basic profile models

A *basic profile* is an identifier and is associated with zero or more numeric parameters (parameters are enclosed in square brackets). A *profile model* define a set of rules for how basic profiles are related by conformance. If a profile x (implicitly) denotes a predicate $P(x)$, a conformance relationship exist: $x \leq y$, if $P(x) \Rightarrow P(y)$.

Since profile models only need to state conformance relationships, the actual meaning of a basic profile may be implicit in a profile model. Profiles can be abstractions over measurable properties like e.g. timing constraints, amounts of memory, but also structure of implementations etc. A policy programmer may however need a specification defining the actual meaning. For instance that `ModerateDelay` means average delay less than 500 milliseconds.

A concrete profile model is specified as a set of *axioms*. To define axioms we propose a simple notation like shown in the example below. Each axiom declares conformance between pairs of basic profiles using the '≤' operator. A predicate for when conformance is true is placed after the **'if'** keyword. Variables in the predicates are bound to the parameters given inside brackets. Omitting the predicate in a rule means **'true'** (corresponds to simple declared conformance).

```
NetGuaranteed ≤ NetEstimated ≤ Net;
LowLoad ≤ ModerateLoad ≤ HighLoad;
LowDelay ≤ ModerateDelay ≤  AnyDelay;
Delay[x] ≤ Delay[y], if x <= y;
Delay[x] ≤ LowDelay, if x <= 100;
XRes[x] ≤ XRes[y], if x <= y;
Disp[x1,y1] ≤ XRes[x], if x1 >= x;
Disp[x1,y1] ≤ Disp[x2,y2], if x1>=x2 AND y2>=y2;
```

From the rules above, we can for instance infer that the expression `Delay[10]` satisfy `ModerateDelay` and that `Disp [2000,1000]` satisfy `XRes[500]`.

As a proof of concept we have implemented a profile model compiler which checks the correctness of the definition, computes a set of additional rules which can be derived from the axioms. It generates code which facilitates efficient testing of conformance between any pair of basic profile expressions.

3.2 Composing expressions

3.2.1 Sum operator

Profile expressions can be combined using the '+' operator. The semantics of this operator is logical conjunction. If a profile expression x denotes (implicitly) a predicate $P(x)$, A profile expression $x+y$ denotes a predicate $P(x+y) = P(x) \wedge P(y)$.

From this definition it is straightforward to infer conformance. For instance $(x+y) \leq x$. Furthermore, $z \leq (x+y)$, if $z \leq x$ and $z \leq y$.

3.2.2 Component sum operator

The '⊕' (component sum) operator is used to state expressions regarding separate environments. To satisfy a component sum $x \oplus y$, both x and y must be satisfied, but x and y cannot be satisfied by the same profile instance. For a profile z to conform to $(x \oplus y)$, z must itself be a component sum $(a \oplus b)$ where $a \leq x$ and $b \leq y$.

A profile expression $(x \oplus y)$ denotes a predicate $P(x \oplus y) = P_1(x) \wedge P_2(y)$. where P is a composite of P_1 and P_2.

3.2.3 Expressions in general

From the definitions above we have developed a complete syntax and semantics of profile expressions formed by these operators. Based on this, we have developed conformance rules which can be used to match any expressions in this language. We refer to [11] for a more complete set of definitions and proofs.

A conformance testing algorithm has been implemented as a proof of concept. Conformance testing software (which will be part of policy trading software) will link in code generated by the profile model compiler.

3.3 Example

Consider an application for interactive browsing of graphics representing large and complex models (e.g. GIS). Clients initiate sessions to a server, and may have requirements for presentation quality and average response-time. Network connectivity and client device capabilities may vary, and the graphics rendering may put a high load on servers. The choice of policies for bindings will depend on user requirements and capabilities of client devices, servers and network.

A policy, which offers to satisfy low response time and a certain image quality may have the following expectation: A certain a minimum size of the display on the client side, a network connection satisfying an estimated *“NormalBW”* bandwith and latency better than 20, and a server with a "fast" CPU and a moderate load.

```
Client + ( (Display[800, 400] + Colour)
 ⊕ (NetEstimated + NormalBW + Delay[20] )
 ⊕ (Server + FastCPU + ModerateLoad) )
```

A client environment (e.g. a portable device) may for instance express that it is capable of two display modes: One normal colour mode and one monochrome mode with higher resolution, by including a component sum of two display instances:

```
(Display[200, 100] + Colour)
⊕ (Display[400, 800] + Mono)
```

4. MIDDLEWARE ARCHITECTURE

In this section we describe some highlights of our experimental middleware platform and how such a platform can support the run-time characterisation in the profile model described in section 3 above. The implementation is based on parts of FlexiBind [3].

4.1 Basic binding framework

Binders are pluggable components responsible for establishing bindings. Service deployment would mean associating objects with a suitable *generator* (server side binder), which generate an interface reference which can be *resolved* by a corresponding client side binder. Binders creates *bindings* which are not necessarily active. Bindings are associated with a metapolicy and are represented by explicit objects both on client and server side, or in all address spaces involved in the binding.

Activators are pluggable components responsible for activating bindings according to some policy. Activating a binding involves loading and instantiating an activator component. This may (depending on the policy) allocate resources needed by the activation, as well as setting up protocols, transparency objects, or other aspect implementation components. A policy will actually contain a reference to some activator implementation (a Java class in our experiment).

4.1.1 Channels

When a server generates an *interface reference* for an interface, some protocol information must be passed along with it such that clients know how to negotiate and bind to it. This observation suggest that the activation (protocol stack) should be divided into two parts: (1) A protocol dependent part which is identified in interface references, and a (2) protocol independent part which is negotiable. On the server side, the protocol dependent part should normally be the minimum needed to listen for incoming calls and to perform negotiation. On the client side, this means that we know what the protocol-dependent part of the activation should be at binding time, but it doesn't necessarily have to be activated before the rest of the stack is activated.

It is useful to encapsulate common protocol dependent configurations in components called *channels*. An instance of a middleware platform should offer access to one or more default channel instances and/or an interface to instantiate channels.

4.2 Negotiation aware bindings

Negotiation is handled by *negotiator* metaobjects which can be attached to bindings. They may intercept the methods for activation/deactivation to modify their behaviour. This essentially mean to add a mechanism for deciding on what activator to select. On the server side, binder would set up a negotiation metaobject which also export a special interface to be remotely invoked by clients to perform the negotiation. In our experiments this interface offer the following operations:

- *get_Activation*. Start the binding process on the server. It takes the user-requirement and the client side environment descriptor as arguments. It creates a prioritised list of candidate policies, and return the client part of the first policy which successfully is 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.

A *policy-trading service* is located on the server, and it is used by the negotiation metaobject to compute a list of candidate policies. The trader is loaded with policies, each containing a reference to a client activator and a server activator. A corresponding client side binder would set up a negotiation metaobject (negotiator) which at first invocation or when explicitly requested, computes the two profile expressions to be sent with the *get_Activation* message. Figure 2 illustrates the binding setup where the *target* represent the actual application object on the server or a proxy object on the client.

The approach described here is one of several possible ways to design a negotiation protocol. It assumes a client/server model and that the probability of an activator failure is low. It can be extended to involve more components, for instance a three tier architecture with a backend server.

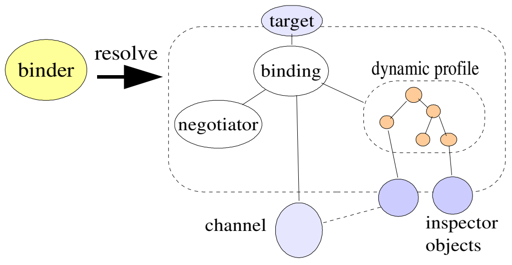

Figure 2. Negotiation aware binding

4.2.1 Interface references

The name of the protocol is part of interface references and is used to select a corresponding resolver component. In our initial scheme, the channel would identify the protocol. However, a server binder could also set up a negotiation scheme. Thererfore a protocol-id would be composed from two parts: One determined by the listening channel and one by the binder. Furthermore, the negotiation scheme described here will require two target identifiers, one for the actual target interface and one for the negotiation interface.

4.2.2 Dynamic profile expressions

We claim that it is a metapolicy issue how the environment-descriptors (c.f. section 2) are produced, since the relevance of properties would depend on the application, the binding type, the platform, the channel used, etc. As shown in figure 2, the binder would set up per binding instance, the necessary structures to produce such expressions.

Some parts of the descriptors may be static. This is the case for platform properties like display resolution or the availability of certain channels. However some properties may change due to varying load etc. Some may depend on the location of the peer, like for instance estimated end-to-end network delay. These 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 parse tree). Parts of this 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 easily set up the composition and the static parts as expressions embedded in the binder code.

4.2.3 Inspector objects

To support dynamic profiles we introduce *inspector* objects. Their role is to generate profile expression fragments describing platform specific facts or measurable properties of the system when requested. Inspectors offer an interface with a method *getProfile()* which returns an expression. A dynamic profile node would refer 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 configured by channels, or they may be configured by binders to report properties of individual bindings.

Some of the inspectors would need to be configured with a *target object* (a reference to a local implementation or a remote interface reference). 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:

- Estimate end-to-end invocation time by invoking probe-operations on the remote system. An inspector could e.g. return a profile `"RTT[n]"` (where *n* is a number denoting the round-trip time in milliseconds). More sophisticated implementations could use of policy specific interceptors or layers in the invocation chain which monitors the time for real operations, however requiring an existing activation.
- Determine by probing, if the remote system is reachable by the UDP protocol (not always the case if endsystems are on different IP-subnets). This can be useful if policies use UDP based invocation protocols or RTP for continous media streams.
- Estimate the load on the CPU, network interface or other resources on the platform. Such an inspector may make use of operating system specific services. For instance the CKRM module [15] for the Linux kernel provides class based reservation and monitoring of CPU, storage or listening sockets. A class could for instance guarantee that its members get a certain share of the resource. This can be used to determine in which class it is possible to place the threads of a session at a given time instance.

In the case of using class based resource management, actual reservations would be encapsulated in policies. The negotiation scheme cannot guarantee that reservation will succeed, unless the middleware is given exclusive access to the classes of interest by the O.S, and unless the negotiation protocol provides proper concurrency control wrt. resources of interest.

4.2.4 Naming and scoping support

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 might set up some, (2) channels might set up some, (3) binders set up some, and (4) some are metapolicy specific (set up by binders) but shared between the bindings sharing a metapolicy. Binders should be allowed to use and compose these objects.

This suggest that the middleware platform should support a naming and scoping mechanism for inspectors. Scoping is organised like in figure 3: The scope of a binding will also include the scope of the platform. We may want override a name defined in the platform scope. For instance, a metapolicy may wish to specialise the behaviour of a display inspector to reflect that only a part of the display could be used. It could then install a special inspector which delegates to the platform level display inspector but modifies its output.

4.2.5 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. It is convenient for a metapolicy programmer to embed textual representations of expressions and let the middleware evaluate it. In such expressions one could use the '$' prefix to refer to parts which are expanded at negotiation time. They would refer to installed inspectors by name.

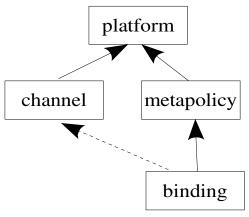

Figure 3. Naming contexts

4.2.6 Example

Recall the example in section 3.3. A client binder sets up an inspector named *'rpc-channel'* which returns the properties of an available RPC channel. An inspector named *'display'* is set up by the platform and returns the properties of the display. The client binder code would contain the following.

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

During negotiation, this expression is evaluated, i.e. the dynamic profile parts are replaced by expressions returned by the inspectors, e.g. *$rpc-channel* estimates bandwidth and delay and return e.g. `"NetEstimated + HighBW + Delay[10]"`. The resulting expression is sent to the server in the *get_Activation* operation and the server adds its own expression (evaluated in a similar way) by using the component sum operator. The resulting expression is then used when searching for a policy.

4.3 Implementation

The ideas presented here has been partially implemented. This includes binders, activators, a negotiator framework, an example negotiator pair, a simple policy-trader, dynamic profile-expression evaluation, an inspector framework and some example inspector and naming spaces which can be linked to provide proper scoping. Experiments using this implementation is currently being carried out.

We observe that the idea of naming and scoping have a wider application than only inspectors. In [3] we propose a extensible interface hierarchy for the PPI (policy programmer interface), which is used by policies to get access to services of the platform and which facilities the pluggability check of policies by using the dynamic type checking mechanism of the programming language. This scheme does not scale well wrt. number of possible platform configurations with different sets of services. It is not suitable for handling a varying number of instances of the same type, e.g. channels. This also indicates that one could benefit using declarative scripting, not only for composing dynamic profile expressions, but also for defining binders in general since different binders often represent sligthly different ways to configure and use a set of standard components.

5. RELATED WORK

Binding models in reflective middleware [1] is maturing. The ANSA FlexiNet framework [12] allows dynamic pluggability and selection of binders, [3] adds the concept of pluggable and replaceable policies for binding activation. The OpenORB binding model [14] focuses on extensibility wrt. binding types. Here, the client/server model is one of many specialisations. Since the concept of binding types includes a negotiation protocol, scope of binding etc, it overlap with our concept of metapolicy. The binding type will clearly constrain metapolicy, but it also seems like metapolicy would need to contain different aspects, some of them orthogonal to binding-type.

Much research has been done in QoS but is often tied to specific application domains, technologies, components or layered architectures (c.f. [16]). This includes QoS negotiation which is typically based on parameters and explicit constraints on parameter ranges, which may be computationally complex.

QuO [17] focuses on adaptation, contractual QoS and aspect languages. Contracts may be defined in a specific language, based on regions, constraining values on measured properties. Contracts are explicitly represented at run-time and closely tied to the server implementation. Furthermore this model does not address negotiation among autonomous components. QML [2] is mainly a language for QoS contract specification. A run-time representation is possible, however somewhat ad hoc. CQML [9] extends and generalises over this model and add some support for composition in the individual QoS characteristics. QML and CQML connect contract-templates to the service interfaces by use of so called profiles. We aim to make contracts more orthogonal to service types. Also, our approach offer a hybrid of declared and rule-based conformance instead of a strictly parameter based approach. Furthermore it addresses composition which is weakly supported in other approaches.

QuA [18] propose platform managed QoS as a general solution to preserve the safe deployment property for compositions of independently developed components. An important part of QuA is a framework for service planning [19], i.e. composing software components and resources to realise a service according to a set of QoS constraints. This is not far from the purpose of policy trading. QuA proposes to use a quality-loss model and utility functions, which has a more limited scope than our profile model but at the other hand, is suitable for maximising satisfaction in addition to just finding satisfactory contracts.

6. CONCLUDING REMARKS

We propose a model for declarative expressions to be used in negotiation of bindings in open systems. From application or domain specific rule-bases, we can infer conformance between pairs of expressions in this model. A compiler can derive a full set of rules and generate code which facilitates efficient conformance checking. Our model supports composition, i.e. conjoining of expressions describing separate components.

We also propose a scheme for how middleware can support automatic characterisation of resources or other relevant properties as well as composition of these. 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 perform mapping 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. The profile model can simplify negotiation, and matching of policies can be more efficient than with more traditional parameter based negotiation, but requires careful design of profile models as well as conventions for composing expressions. A negotiation scheme strictly based on conformance does not support finding an optimal solution. That is a disadvantage in some cases.

Issues for future work in this area include validating this approach by applying it to application scenarios and alternative binding types. We observe that the metapolicy includes many aspects and that binders to a large extent share code. One could explore the use of declarative scripting languages for defining platform setup, binders, negotiators and activators. Since various applications or application domains may define their own profile models it is interesting to see how we can provide interoperability among autonomous domains by combining their models. Here, we may benefit from work performed in the area of semantic web with ontologies.

7. REFERENCES

[1] Kon, F., Costa, F., Blair, G and Campbell, R. H. *The Case for Reflective Middleware*. CACM June 2002/Vol. 46, No. 6.

[2] Frølund, S., and Koistinen, J. *Quality of Service Aware Distributed Object Systems,* Hewlett Packard Software Technology lab. report: HPL-98-142. 1998.

[3] Hanssen, Ø. and Eliassen, *F., A Framework for Policy Bindings*, In Proceedings of DOA'99, Edinburgh, IEEE press,

[4] Hanssen, Ø. and Eliassen, F., *Policy Trading*, In Proceedings of DOA'00, Antwerp, IEEE press, 2000.

[5] Bearman, M. Y., *ODP Trader*, In Proceedings of ICODP'93, Berlin, 1993

[6] *ODP Trading Function*, Report, ITU-T X.950 – ISO/IEC 13235.

[7] Hanssen, Ø. and Eliassen, F., *Towards a QoS aware Binding Model*, In Proceedings of SYBEN'98, Zurich, Spie press, 1998.

[8] Abadi, M., and Lamport, L. Open Systems in TLA, In Proceedings of ACM Symposium on Principles of Distributed Computing, August 1994.

[9] Aagedal, J. Ø., *Quality of Service Support in development of Distributed Systems,* Ph.D. Thesis, University of Oslo, 2001.

[10] Abadi, M., and Lamport, L. *Conjoining Specifications*, Digital Systems Research Center, Report 118.

[11] Hanssen, Ø. *A Declarative Profile Model for QoS Negotiation*. Technical report 2005-54, University of Tromsø, Computer Science Department, 2005.

[12] Hayton, R. and Herbert, A., *FlexiNet: A Flexible, Component-Oriented Middleware System*, Lecture notes in Computer Science, 1752, p. 497 ff, Springer Verlag, 2000.

[13] Blair, G.S., et al. The Design and implementation of Open ORB 2, IEEE Distributed Systems Online, 2, (2001), no. 6.

[14] Parlavantzas, N., Coulson, G. and Blair, G.S., *An extensible Binding Framework for Component-Based Middleware*, In Proceedings of EDOC 2003.

[15] Nagar, S., et al., Improving Linux resource control using CKRM, In Proceedings of the Linux Symposium, Vol two, July, 2004.

[16] Ecklund, D., et al., *QoS Management Middleware: A Separable, Reusable Solution*, In Proceedings of IDMS 2001, LNCS 2158, pp. 124-137, Springer Verlag 2001.

[17] Loyall, D.E.,et al. *Specifying and Measuring Quality of Service in Distributed Object Systems*, In Proc. ISORC'98, IEEE press 1998.

[18] Staehli, R. and Eliassen, F., *A QoS Aware Component Architecture*, Simula Research Laboratory Research report, 2002 – 12.

[19] Solberg, A., Amundsen, S., Aagedal, J.Ø. and Eliassen, F., *A Framework, for QoS-Aware Service Composition*, In Proceedings of 2nd ACM Intl. Conference on Service Oriented computing, ICSOC 2004.