GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions*

Eric Wohlstadter†, Stefan Tai‡, Thomas Mikalsen‡, Isabelle Rouvellou‡, and Premkumar Devanbu†

†Center for Software Systems Research
University of California, Davis, CA 95616
wohlstad,devanbu@cs.ucdavis.edu

‡IBM Watson Research Center
New York, USA
stai,tommi,rouvellou@us.ibm.com

Abstract

A holy grail of component-based software engineering is "write-once, reuse everywhere". However, in modern distributed, component-based systems supporting emerging application areas such as service-oriented e-business (where web services are viewed as components) and Peer-to-Peer computing, this is difficult. Non-functional requirements (related to quality-of-service (QoS) issues such as security, reliability, and performance) vary with deployment context, and sometimes even at run-time, complicating the task of re-using components. In this paper, we present a middleware-based approach to managing dynamically changing QoS requirements of components. Policies are used to advertise non-functional capabilities and vary at run-time with operating conditions. We also provide middleware enhancements to match, interpret, and mediate QoS requirements of clients and servers at deployment time and/or runtime.

1 Introduction

Can component A safely inter-operate with component B? This question is not easy to answer, even for a pair of components within a single large project. However, connecting incompatible components may cause havoc: application crashes or (even worse) subtle, semantic errors. Now, consider the far worse problem of two components, previously unacquainted with each other, seeking to dynamically inter-operate in an open network, such as the internet or an ad-hoc wireless network. Even though it sounds daunting, this is exactly what is envisioned in emerging arenas, such as service-oriented computing (SOC), Peer-to-Peer (P2P) networks. This is the setting of our work: we're specially concerned with dynamically reconciling QoS conflicts between components, at run-time.

The first line of defense against component mismatch is a functional interface that allows for static typing or contract/schema verification at connection points to readily expose incompatibilities. Additionally, components can be guarded by logical specifications of pre- and post- conditions governing interactions between components. However, in distributed applications, specially on wide-area-networks, software engineers must also consider quality-of-service (QoS) requirements such as security, performance, and reliability when designing component connections. These requirements must be supported by software on both the client and server in order to operate properly. For example, software that checks passwords on the server side should be complemented by software that provides passwords for the client. Security requirements, however, may vary with deployment context and even at run-time. How can we then ensure that components have compatible QoS features ? Extending component interfaces directly with information about non-functional concerns limits the reusability of the interface and hence any components implementing it; furthermore it also limits customizability, *e.g.,* the ability of local security officers to tailor the policies to suit their settings. Thus, there has been a great deal of interest recently in techniques to provide an effective separation of concerns for end-to-end non-functional requirements and the more stable functional requirements.

With such approaches, components only implement a functional interface; QoS features such as security are left unresolved until deployment time. A declara-

*Prem Devanbu and Eric Wohlstadter were supported by NSF CISE grant No. 0204348. Wohlstadter was also supported by IBM Summer Student Internship.

tive QoS specification, written by a deployment specialist, can be used to transform the original components through the use of containers and generative programming, aspect-oriented programming (AOP) [6, 26], or software wrappers [8]. These approaches are not truly dynamic: they force commitment to QoS features at deployment time. In addition, these approaches are server-centric, and do not consider the issue of matching client-side QoS features to the deployment policy on the server.

This inflexibility and "server-centricity" limits the use of current approaches to QoS feature management in new, emerging application areas such as SOC and P2P computing. Here, we need a highly dynamic, and symmetric (*not* server-centric) way of managing end-to-end QoS requirements. In these settings, components, deployed as autonomous software processes, create and manage relationships with other processes dynamically; processes can cross-dress, playing either client or server role. Processes can exist in different administrative domains, with different deployment contexts (which may also change dynamically) and thus have different QoS requirements. A new approach is needed, to provide *dynamic and symmetric reconciliation* between the (potentially different) QoS features of two communicating processes. However, QoS features can interact in various ways, and this complicates reconciliation. For example privacy requirements of the client and billing or payment requirements of the server may conflict. We use the term feature interaction[27] to reflect how feature combinations affect each feature's ability to function as it would separately.

Feature interactions can be complex, subtle, and very difficult to identify. Finding such interactions is outside the scope of our research. In addition, feature preferences are a matter of deployment policy, and can vary. In our work we assume a fixed ontology of features, with all interactions explicitly identified ahead of time. Our contribution is a mediation mechanism to support the dynamic management of QoS features between two components in a WAN setting that encounter each other for the first time. We provide a declarative language for specifying the QoS feature preferences and conflicts, and a middleware-based resolution mechanism that reasons using these specifications to dynamically find a satisfying set of QoS features that allow a pair of components to inter-operate. The language for specifying QoS features, preferences and conflicts, *GlueQoS* , is an extension of the WS-Policy[3] language.

The remainder of the paper is organized as follows: we start by motivating feature mediation in section 2, then in section 3 we present current approaches and an overview of our approach, a methodology to support building policies is described in section 4, section 5 describes how we have extended WS-Policy to support new problem areas, in section 6 we describe the details of our implementation, we conclude in sections 7 and 8 with related work and conclusions.

2 Security Example

We consider a web services example where two security QoS features are in play. This example illustrates the issues that arise when features interact in a setting where clients and servers have different polices with regards to QoS features.

The first feature is *authentication*. Open distributed services must protect themselves from unauthorized access; so client requests must be preceded or accompanied by an authentication step involving the presentation of credentials. Credentials can be based on a password, or on public-key signatures. In this case, a QoS feature on the server side would be responsible for checking credentials, and the corresponding QoS feature on the client-side would be required to present the appropriate credentials.

The *client-puzzle protocol* (CPP) QoS feature [5] defends against cpu-bound denial-of-service (DoS) attacks. A DoS attack occurs when a malicious client (or set of malicious clients) overloads a service with requests, hindering timely response to legitimate clients. CPP works by intercepting client requests and refusing service until the client provides a solution to a small mathematical problem. The time it takes to solve the problems are predictable; fresh problem instances are created for each request. The need to solve puzzles throttles back the client, preventing it from overloading the server. Typically the puzzle involves finding a collision in a hash function, i.e., finding an input string that hashes to a given n bit value modulo 2^m, for $n > m$. Such puzzles are very easy to generate and require about 2^m times as much effort to solve, given a collision-resistant hash function. Further details are not relevant to our presentation, and can be found in [5].

CPP and Authentication interact in interesting ways. For example, suppose the server's only QoS requirement is to prevent DoS attacks. If we trust authenticated clients not to mount DoS attacks, then the authentication feature and client-puzzle are *equivalent* and can be substituted one for the other; it would be redundant to use both. However, sometimes authentication may not imply a decreased risk of DoS attacks, so these features would be viewed as *orthogonal*. In other situations, we may require both authentication

and DoS defense; here, the two features are viewed as *complementary*, since an added benefit is gained by using them together.

Client-side preferences must also be considered when selecting the QoS features that govern a client-server interaction. A client may consider CPP and Authentication to be equivalent, and express a policy that it can use either. A client with a performance requirement, however, would naturally prefer to employ authentication to avoid computing puzzle solutions. A client who values its privacy would prefer to expend CPU cycles in order to not have to reveal their identity; this client may prefer to use CPP rather than provide identity-revealing credentials.

Existing policy languages such as WS-Policy can express these above possibilities. GlueQoS builds on such previous work. GlueQoS takes into account both the client's and server's QoS feature preferences, and provides a middleware that can determine a compatible QoS feature composition, when possible, and otherwise declare that the partner's policies are incompatible. This is done at runtime in an open dynamic environment, thus liberating the deployment expert from considering all possible client-server QoS pairings, and certainly also liberating the application developer from tangling application logic with QoS considerations.

Furthermore, GlueQoS supports policy resolution in situations when QoS feature preferences are determined by run-time environmental conditions. For example, a server might only require the CPP feature when it has a high cpu load. Furthermore, a server could continuously increase the difficulty of puzzles as load increases (as advocated in [5]). These environmental changes of the server can interact with a client's QoS features. For example, some clients may be willing to use the CPP feature only for small puzzle sizes. The GlueQoS policy language gives the system deployer flexibility to make these sorts of tradeoffs between feature composition.

3 QoS Features for Middleware

We begin with an analysis of the handling of QoS features in current middleware, and then present an overview of our approach.

3.1 Current Approaches to QoS

Layered composition is a popular way to provide extensibility, when layers can be implemented as separate components. There are many mechanisms that support layered composition, including mixin layers [21], interceptors [17], micro-protocols [4] and Aspects [12]. We use it in GlueQoS to support the incremental addition of many types of QoS features for middleware architectures. It is useful to distinguish this from the Layered Architectural Style (See [20], Section 2.5). In that style, each layer provides one set of interfaces, and relies on a (usually) different set of interfaces from the layer below it. The goal is to provide an increasing level abstraction, and hide details from higher layers. In our case, *all layers* essentially provide a similar interface; higher layers may *add* more services, but typically nothing is hidden from the topmost layer. In this section we describe how QoS features have been added to middleware using three variations of layered composition, decorators, interceptors, and advice.

A decorator layer, also called wrapper, exposes the same interface to layers above as the interface onto which it is composed. This allows client code (the above layers) to remain unaffected by layer composition. Each layer may also extend the interface for use by decorator-aware clients. Similar effects can be achieved by mixin layers. A decorator approach is used in Lasange [24] to provide client customizable remote method invocations. Examples are given for client specific security and business rules. Quality of Objects [18] (QuO) uses decorators that are dynamically chosen based on runtime conditions. The choice of decorators is driven by policies that take into account runtime conditions called System Conditions (SysConds). This allows QuO to provide QoS services relating to intrusion detection, network bandwidth management, and fault-tolerance.

Interceptors also provide layered composition, but they are completely generic, relying on reflection. Information about other layers (including the original application components) is gained dynamically via reflection, and used to monitor and modify application behavior. This provides flexibility at the expense of static type checking. Many CORBA based QoS features are implemented using interceptors including security, fault-tolerance, transactions, and real-time features.

Aspect-Oriented approaches [12] can add incremental QoS features, as can methods based on multi-dimensional separation of concerns [23]. These methods provide the benefits of both decorator and interceptor based approaches. In some cases, composition can be statically checked. These approaches differ from decorator- and interceptor-based approaches in that the effect of composition can be crosscutting. DADO [26] exploits aspects to add security, performance monitoring, and caching examples to CORBA based applications. Duclos et. al. [6] shows how aspects can be used to provide security, transactional se-

mantics, and object persistence to applications using a CORBA Component Model. It is also worth mentioning that the use of aspect-like mechanisms for security and transactions, particularly, is not without controversy [9, 14].

In this paper, we are primarily concerned with *acceptable compositions of QoS layers*, particularly in a highly dynamic, distributed setting. However, we do not address the exact ordering of features which is mostly a local phenomenon. We want to consider both client and server QoS requirements, expressed in a declarative form (called policies in our work), and use these in a well-founded manner to arrive at an acceptable set of layers that implement a set of policies acceptable to both client and server. Other work as described above has not addressed this issue. We believe this to be increasingly important as new application areas such as service-oriented computing and P2P applications require interoperation between autonomous components. The focus is on flexibility between autonomous client/server or peer interactions and not at the architectural level such as the work on adaptive, self-healing, or self organizing systems[10].

Handling QoS Negotiations As described above, the layering of QoS features is currently used in various middleware settings. Figure 1 shows 3 QoS features on a server-side component, and the corresponding QoS features on the client side. For example, there may be security layer, a fault-tolerance layer, and a performance-related (*e. g.,* caching) layer. In a WAN setting, the QoS policies of two components may be independently administered. For example, the EJB framework [19] allows server administrators to separately configure policies for transactions and security, using separate specification elements. Clients and servers thus will need co-ordinate at run-time to ensure proper QoS inter-operation. In order to agree on operating parameters, called *attributes* in our framework, each layer may employ a *meta-protocol*.

In a standard (non-distributed) setting, the operation of a *program* can be modified by a reflective *meta-program*. Thus, in a reflective programming environment, a *meta-class* can be written, that enforces access control policies on the methods of a *class*. Likewise, in a distributed setting, a *meta-protocol* is the reflective counterpart of a protocol: *i.e.,* a reflective exchange of signals that modifies the run-time behavior of a protocol or communication layer. It is a kind of handshake or setup protocol.

Existing QoS meta-protocols for middleware are quite limited, and generally not applicable in WAN settings. The policy for each QoS feature is specified separately, and each policy feature negotiates separately with its counterpart to set run-time conditions. Specifically, they cannot reason dynamically about relations between QoS layers that arise when feature interactions are unavoidable.

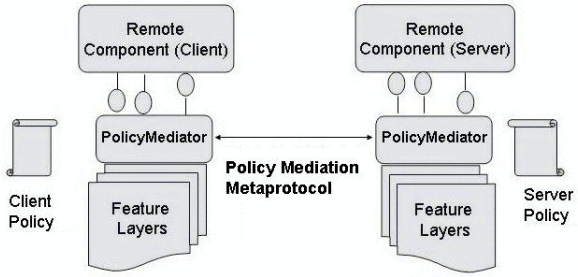

Figure 1. Policy Mediation Metaprotocol

This feature-by-feature configuration and negotiation approach is ripe for trouble, in feature-rich systems operating over dynamic WAN settings. Various types of feature interactions will invariably arise in these systems, as illustrated earlier in § 2. The most obvious approach would be to make each feature implementation sensitive to the presence of other features. However this approach would produce tangled implementations which would interleave the logic of different features, thus violating the separation-of-concerns principle, and making maintenance difficult. Our policy mediation protocol can be of help.

3.2 GlueQoS Overview

GlueQoS separates out the task of handling feature interactions into a GlueQoS policy mediator ($\mathcal{GPM}$) which is added to each component. The $\mathcal{GPM}$ on each end oversees the configuration of QoS features at that end; it communicates with its counterpart $\mathcal{GPM}$ at the other end to select the right set of QoS features (figure 1). This is accomplished using the GlueQoS policy mediation meta-protocol ($\mathcal{GPP}$). Existing systems such as EJB let a deployer (only on the server side) configure each feature separately, using a configuration file; in GlueQoS, a deployment specialist (on both client and server) uses a high-level abstract, declarative language, writing polices that specify both feature interactions and runtime tests. The $\mathcal{GPM}$'s commence an end-to-end interaction, using the $\mathcal{GPP}$ protocol, first evaluating policy based on runtime conditions (this process is called *policy reduction*), then exchanging policy information. They then compute an intersection of the policies (called *policy matching*) to find a composition agreeable to both ends.

The policies are specified in the GlueQoS policy language ($\mathcal{GPL}$). The design goals of this language are to provide an abstract, declarative, expressive means of describing QoS features, their interactions, and their sensitivity to operating conditions. The language provides a set of built in operators to specify feature interactions, as well as the ability to extend the system with functions to measure operating conditions (such as load, available energy, bandwidth, *etc*, similar to the SysConds of QuO as described above). We defer a detailed description of the language to Section 5, until we have presented an analysis of the feature interactions handled by GlueQoS. The semantics of the language are implemented by the $\mathcal{GPL}$; and therefore the language description is in fact a description of the $\mathcal{GPL}$'s internal function.

It now becomes a responsibility of deployment experts to describe acceptable feature combinations using policies. In the next section we take the first steps at providing a methodology to guide this task.

4 Methodology

The GlueQoS methodology introduces three new roles (see Figure 2). A *feature engineer* analyzes application-independent QoS requirements to document the QoS requirements addressed by each QoS feature, and the interaction between QoS features. *Feature builders* construct features; features can be parameterized by so-called *attributes* that allow QoS features to be tuned by middleware at runtime. The *deployment expert* chooses compositions of features (called *feature combinations*), possibly based on runtime *tests* of the environment, suitable for specific application dependent requirements. Now we describe the tasks performed by these three roles in more detail.

4.1 Feature Engineering

The feature engineer is tasked with identifying, and standardizing end-to-end QoS features. Feature Engineers begin the process (an example is shown in figure 2) by analyzing application-independent non-functional requirements, relevant features, and feature interactions.

Feature engineers also identify feature components that implement QoS feature requirements. The mapping from QoS non-functional requirements to feature components may not be simple and one-to-one. Figure 2 illustrates four requirements: *Access Control, Availability, Performance, and Privacy.* Two are mapped to components: *Access control* is realized via *Authentication* and *Availability* as the CPP (Recall

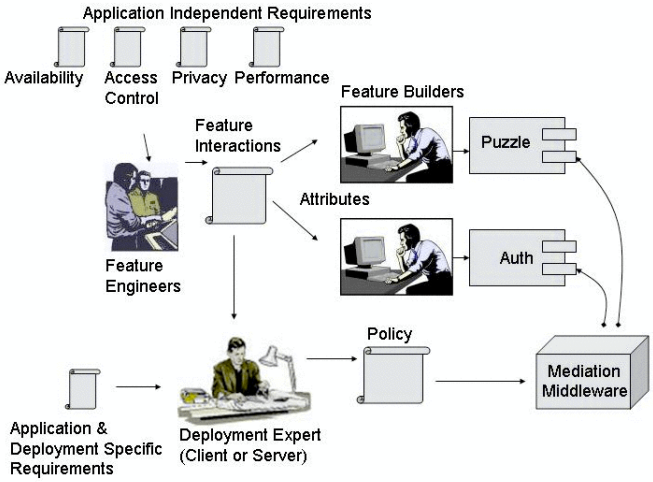

Figure 2. High Level Process: Feature Engineers analyze application independent QoS requirements to determine features, attributes, and interactions. Deployment experts choose combinations and attribute values based on application dependent requirements and runtime tests. Feature Builders implements features parameterized by attributes. Middleware mediates policies and adjust features through attributes.

that the CPP protocol can defend components from denial-of-service attacks, thus promoting availability). As we saw earlier, in Section 2, the other two requirements are accounted for through feature composition and runtime feature parameterization. In GlueQoS a feature composition is a logical combination of feature names that is used to express how features may be combined for a particular situation. In each situation, specific non-functional requirements are indicated by deployment-time and runtime conditions. These non-functional requirements determine the applicable feature combinations and feature attributes. Feature engineers must *a-priori* consider various possible circumstances, and identify for each the applicable combinations of QoS features, and QoS feature parameterizations. We recommend that the feature engineer produces a *"Feature Interactions Document"*, describing this information. Our model for feature interactions is described next.

For our purposes, a *feature interaction* is the effect that two QoS features have on each other. Figure 3 shows our ontology for various kinds of feature interactions. In this table, we denote the qualitative effect on a particular requirement as the function $\mathcal{E}$(A) where A is some feature. This notation is only meant to provide an intuition as to the meaning of the various categories of interactions in our ontology, and does not imply the use of formal analysis. The interaction of features A and B is denoted as $\mathcal{E}$(A,B), the combined effect of fea-

tures A and B. We call the effect positive (> 0) when it acts to satisfy some non-functional requirement, negative (< 0) when it prevents other components from satisfying functional or non-functional requirements and zero (0) when there is no observable change. The table also shows sets of possible feature deployments that are induced by these interactions.

Here we briefly describe each interaction:

Orthogonal: Two features `A, B` are orthogonal if their combined contribution to requirements fulfillment is exactly equal to the sum of their individual contributions. These features could reasonably be combined together or individually.

Complements: Two features are complementary if their combined contribution is greater than the sum of their individual contributions. It is advantageous to combine these features together but they may be deployed individually.

Dependent: A feature A is dependent on feature B if their combined effect is positive but the individual effect of A is non-positive. Feature A should only be deployed with feature B.

Conflicts: Two features conflict if their combination has a negative effect on the behavior of the entire application. The deployment of one feature should exclude (XOR) the deployment of the other. The decision that an effect is negative is arbitrary but may includes effects such as introducing deadlock or putting sensitive data in inconsistent states.

Prevents: A feature A prevents feature B if their combined effect is equal to the individual effect of A. The deployment of A excludes B from effecting the system regardless of policy. This is different from conflicting because the effect is confined to the features themselves.

Equivalent: Two features are equivalent if their individual effects are qualitatively the same. There is no need to deploy these features together but remote partners might only support one of them, using an XOR or OR combination yields greater flexibility in the face of heterogeneity.

Feature interaction	**Possible Combinations**	**$\mathcal{E}$(A,B) Interaction Effect**
Orthogonal	{A, B} , {A} , {B}	$\mathcal{E}(A) + \mathcal{E}(B)$
Complements	{A, B} , {A} , {B}	$\mathcal{E}(A, B) > \mathcal{E}(A)+\mathcal{E}(B)$
Dependent	{A , B}	$\mathcal{E}(A, B)>0$, $\mathcal{E}(A) \leq 0$
Conflicts	{A}, {B}	$\mathcal{E}(A,B)<0$
Prevents	{A}, {B}	$\mathcal{E}(A)$
Equivalent	{A, B} , {A} , {B}	$\mathcal{E}(A) == \mathcal{E}(B)$ $\mathcal{E}(A,B) == \mathcal{E}(B)$

Figure 3. Feature interactions

4.2 Deploying Features

The deployment expert considers local requirements, and the feature interactions document provided by the feature engineer to design his QoS Policy. Perhaps a client is interested in privacy, but also in performance. *CPP* and *Authentication* are features relevant to both privacy, and performance, which interact (as described in §2); this information would be available from the Feature Interactions Document. Suppose a client values privacy in most cases, but is willing to give away his privacy if it requires too much of a performance sacrifice. Using the information in the Feature Interactions Document, the deployment expert can formulate this as a policy in terms of feature combinations and feature parameters. Combinations are determined by the feature engineer through use of the logical *and*, *or*, and *xor* operators provided by the policy language.

4.3 Implementing Features

The feature builder is responsible for implementing the feature components. This may require opening up some of the details of the implementation[13] for configuration or introspection so that middleware may mediate attribute values acceptable to both parties. For example, the puzzle feature is required to expose the bit length of puzzles so that policies can configure the puzzle feature based on the observed server load and so clients can express performance requirements in terms of puzzle length. This decision of what attributes a feature builder must export is made by the feature engineer in the analysis stage and should be provided in the feature interactions documentation. The methodology described in this section is motivated by the need to provide detailed feature interaction information to the deployment expert. Now a site specific policy can be constructed based on this knowledge using the GlueQoS policy language.

5 GlueQoS Policies

Our policy based approach to feature composition is inspired by emerging standards in the web services industry known as WS-Policy [3]. The WS-Policy specification provides a standard XML syntax for web services to advertise their non-functional capabilities and requirements. We extend the language here for purposes of mediating policies between remote hosts and to consider dynamic effects of deployment environments. In this section we describe the elements of our GlueQoS policy language which uses a superset of the WS-Policy syntax for interoperability.

For our purposes WS-Policy is a syntax for expressing logical combinations of XML data. The data, called assertions, can be arbitrary but is intended to describe relevant non-functional characteristics of a web service. WS-Policy provides three n-ary operators for composition of assertions. They are the All, ExactlyOne, and OneOrMore operators. We restrict these to binary composition and our middleware algorithms interpret them as `and`, `xor`, and `or` operators. These operators are associative so there is no loss in expressive power.

The GlueQoS policy mediation protocol uses a superset of WS-Policy, called WS-Policy+, to exchange client and server policy information. Each feature is represented as one XML element identifying the feature's unique name. The attributes of features are expressed as the element's attributes (XML terminology for parameterization). Agreement of feature attribute values is an important part of feature agreement especially when feature's attributes are affected by the dynamic runtime environment. WS-Policy only accommodates constant attribute values, our WS-Policy+ extends WS-Policy with attribute constraints (such as inequalities).

Feature attributes are used by remote hosts to agree on configuration of features. In the security example, hosts needed to agree on the puzzle size of the CPP feature, in order to resolve the client's performance requirement and the server's DoS defense requirement. We currently support integer, string, and floating point attribute data types. Integer and floating point values can be constrained to a single continuous range using the greater than, less than, and range operators. String values can be constrained using regular expressions. When our policy matching algorithm finds matching feature elements in client and server policies, it compares all the attributes to ensure none of the values or constraints of attributes are in conflict.

The grammar of our complete GlueQoS language can be seen in the EBNF of figure 4. GlueQoS provides a syntactic sugar on top of WS-Policy+ and accommodates consideration of runtime data. Policies make use of QoS Conditions (similar in spirit to QuO SysConds), to include runtime data in the mediation of policies. This is represented in our language by using the object oriented dot notation for static field access anywhere in a policy where a value is expected. The middleware will invoke a no-argument method with the same name on the singleton object managed for the particular class of QuO Conditions identified in the expression. The values returned can be used for the calculation of attribute values or to further refine polices through tests.

Tests are boolean expressions using arithmetic comparisons or string regular expressions (using the infix operator `matches`). The `and` and `xor` operators are overloaded to allow combining feature assertions and tests. In these cases, when a test fails (evaluates to false) the feature assertion is removed from the policy to be mediated by the middleware. This is achieved through a policy reduction step that occurs before policies from client and server are matched. A test is exploited in our security example as presented below.

```
Expr::          LetEnv |
                Assertion (AssertionOp Expr)?
LetEnv::        "let" Assignment* "in" Expr
Assignment::    Var "=" ValueExpr
Assertion::     FeatureName ("[" AttrAssertion* "]")?
                | Test | "(" Expr ")"
AssertionOp::   "and" | "or" | "xor"
AttrAssertion:: Var AttrOp ValueExpr
AttrOp::        "=" | ">" | "<" | ">=" | "<=" | "matches"
Test::          ValueExpr TestOp ValueExpr
                | Boolean
TestOp::        "==" | ">" | "<" | ">=" | "<="
ValueExpr::     ValueExpr ValueOp ValueExpr
                | Atom
                | "(" Test ")"
ValueOp::       "+" | "-" | "/" | "%" | "*"
Atom::          FieldAccess | Var | integer | float
                | regex | "(" ValueExpr ")"
FieldAccess::   uppercase_id "." lowercase_id
FeatureName::   uppercase_id
Var::           lowercase_id
```

Figure 4. GlueQoS Language EBNF: Policies consist of feature assertions and runtime tests combined with logical operators. Feature assertions can be qualified through open point values or functions.

5.1 Security Revisited

```
(1) Server:
(2) let cpu = SystemMonitor.cpuUsage,
(3)    puzzleMax = 16
(4) in
(5)   ((CPP[size = cpu*puzzleMax/2] and Authentication)
(6)   xor (Authentication and (cpu < .5))
(7)   xor CPP[size = cpu*puzzleMax])
(8)
(9) Client1:
(10)  (Authentication)
(11)
(12) Client2:
(13)  (CPP[size <= 4] xor
(14)  (CPP[size <= 4] and Authentication) )
```

Here we see a possible realization of the security example as expressed in GlueQoS. The first policy is shown for the server. A variable `cpu` is declared (line 2) and bound to the current cpu usage (between 0 and 1) using a QoS Condition Object. A constant `puzzleMax` is bound (line 3) to the integer 16 to represent the maximum size of puzzles. Then come lines 4,5, and 6, each representing an alternative QoS combination for the server. An `and` combination of the CPP and Authentication features (line 5) states that with Authentication,

the size of puzzles varies linearly from 0 to 16 depending on cpu load. Another combination (line 6) uses a *test* (`cpu < .5`) to determine whether this combination will be allowable. When cpu load is less than 0.5, the server allows Authentication to be used without the CPP; otherwise just CPP, with the largest puzzle size, can be used. This shows how runtime conditions can dynamically adapt the feature combinations expressed by hosts.

The first client policy is shown on line 10. This client will only use the Authentication feature (perhaps because of software unavailability, or because it is too performance-limited for CPP). Therefore, this client can only create a session with the server when the server's load is less than 0.5.

The second client policy (lines 13 and 14) uses attribute constraints to choose between two feature combinations. Recall that the xor semantics in our language implicitly implies a preference for the first (left) alternative. Consider a situation where this client wishes to maintain its anonymity by not using the Authentication feature. However, it also has a performance requirement that takes precedence. Perhaps the client is on a mobile device with low computing power. Line 13 expresses the client's preference to maintain anonymity by agreeing to the CPP only. However, in order to keep performance at a certain threshold the client will also use Authentication if it will keep the puzzle size low. By comparing to the sample server's policy (lines 5 and 7 in particular) if this client contacts the server when the servers cpu load is 25 percent or lower the client can maintain its anonymity by using CPP only. However, if it contacts the server and the servers cpu load is between 25 percent and 50 percent it will agree to reveal its identity to maintain higher performance. When the servers load passes 50 percent the client will be unable to mediate a satisfactory feature composition with the server.

We have tested this example using our policy mediation middleware to assure that the correct feature combinations were resolved in a variety of situations, however, these policies have not been used to drive actual CPP and Authentication implementations, which are currently only available for DADO [26], whereas GlueQoS currently works in a web-services setting.

5.2 Policy Matching

Policies of client and server must be considered together to find mutually acceptable features and attribute values. The GlueQoS Policy Mediator does this by *policy matching*. First, we describe the policy matching algorithm, assuming information for both client and server is available at one host. In section 6.1 we detail the policy mediation protocol that facilitates the actual exchange of policy data. Policy matching involves three steps: reduction of policies through runtime tests, calculating acceptable assertion sets, and finding compatible assertion sets.

5.2.1 Policy Reduction

In the reduction step, feature assertions are treated as unknown boolean variables, and the entire policy is simplified as much as possible. This allows hosts to express different policies based on runtime conditions. Tests are combined with feature assertions or other tests using the `and` and `xor` operators. The effect of combining a feature assertion, A and a test which evaluates to a boolean value in the reduction step is defined below (the operators are symmetric for this purpose):
A `and` True $= A$
A `and` False $= \mathit{False}$
A `xor` True $= \mathit{True}$
A `xor` False $= A$

As a simple example, consider line 6 in Section 5.2, with the cpu load returned by the `SystemMonitor` greater than 0.5. Thus this would be reduced as follows:
`Authentication and False` $\Longrightarrow$ `False`

In this way, more complex expressions are reduced. On their own, True and False can be used as feature assertions: True matches any feature and False matches no features. Thus, `cpu < 0.99` on its own can be used by a server to simply refuse service if the cpu load exceeds 0.99.

5.2.2 Calculating Acceptable Assertion Sets

We describe the calculation of acceptable assertion sets in terms of boolean algebra [1]. Each feature assertion is interpreted as a unique boolean variable. Matching begins by enumerating the acceptable feature sets of both client and server policies: we calculate all unique sets of feature assertions (interpreted as boolean variables) which, when set to true, would reduce the entire policy to true. For example, the policy of Client2 in Section 5.1 is interpreted as two acceptable feature sets (with attributes removed) `[[CPP],[CPP,Authentication]]`. The operators are order preserving, to capture the client's preferences for the matching algorithm. The time complexity of And and Or operations is n*m where n and m are the sizes of the list operands because they involve finding the cross product of two lists. The other operations are constant time.

[1] A similar interpretation for the single host policy case given in the original WS-Policy specification.

5.2.3 Finding Compatible Assertion Sets

Now that the acceptable assertion sets of both parties have been calculated we need to find sets between both that are compatible. This is achieved by an exhaustive comparison between sets in both lists. Sets may not necessarily be exactly equivalent to be compatible because attributes may involve constraints that could be satisfied by multiple values or unconflicting constraints. For example,matching `[[Puzzle(size < 4)]] and [[Puzzle(size > 2)]]` would resolve into `[[Puzzle(size=range(2,4))]]` (Note that all operations are over the real numbers). Assuming set membership and attribute comparison can be implemented in constant time; the matching itself takes O(n*a+n*m) where a is the total number of attributes from feature assertions in the first list, m is the number of assertions in the first list, and n is the number of sets in the second list. The intuition is that each set in the first list must be compared to each set in the second list which may involve comparing all the attributes of the sets.

Once the matching sets from both parties have been found one of the sets must be chosen to drive the middleware's configuration of features. If more than one such set exists, the first one in the list is chosen in order to comply with any arbitrary preferences of the client. In our current implementation, if no sets match, an exception is thrown. It can be caught and handled by policy aware client code otherwise the exception must be handled as a generic remote exception.

6 Implementation

6.1 GlueQoS Policy Mediation Meta-Protocol ($\mathcal{GPP}$)

The GlueQoS policy mediators at each end of an interaction determine an applicable composition of QoS features for each application session. These features and their operating parameters remain fixed for the lifetime of the application session. The $\mathcal{GPP}$ protocol is primarily active during the initial negotiation phase Figure (5) shows the state diagram of the protocol. Transitions are triggered by either client messages (dashed lines), server messages (solid lines), or server internal events (dotted lines). Each protocol state represents the global state of both participants with respect to the protocol. Client and server start out *disconnected*. When a client locates a server (through a directory service or another third party) it sends a *policy request* to the server to initiate a session. The server creates a session for the client and evaluates any

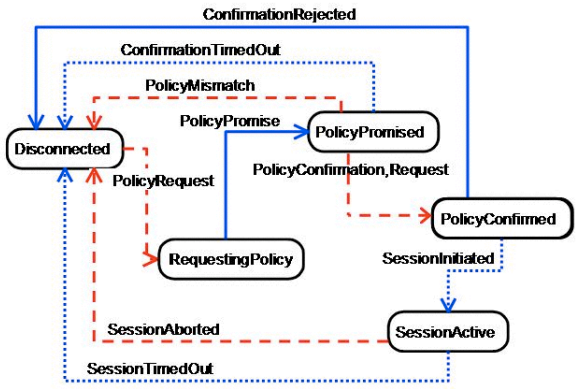

Figure 5. GlueQoS Policy Mediation Protocol State Diagram: Circles represent the two party state of the protocol. Dashed transitions are triggered by client messages. Solid transitions are triggered by server messages. Dotted transitions are triggered by server internal events.

applicable environmental tests (*e.g.* to measure load, collision rate of packets, etc) in its policies, to finalize its current reduced policy. The current policy is considered guaranteed by the server for the lifetime of the session. The server saves this policy, associates it with the current session, and sends the policy to the client through a *PolicyPromise* message. Now, the client must match its own policies with the server and choose a feature combination acceptable to both. A client matches policies by carrying out the three policy matching steps detailed in section 5.2. The feature assertion set chosen by the client is then sent to the server through a *PolicyConfimation* message. This message may piggyback on a real request to the server in order to reduce message traffic. Next, the server verifies that the client's choice of features is compatible with the policy promised. If so, the server removes the promised policy and associates the feature set with the client session where the protocol remains in the *SessionActive* state until the session is terminated.

6.2 Implementation

Our prototype implementation is built around the Apache Axis web service middleware. The functional interfaces of web services can be described using an interface definition language known as WSDL[2]. Policies are attached to WSDL elements (port types, operations, bindings) as metadata. We follow a standardized approach specified by WS-Policy. Our policy mediators on clients and servers process WSDL with attachments in order to determine policies for specific web services

interactions. Our implementation currently mediates policy only at the operation granularity. Invocation of operations with different policies requires a new policy mediation step per session.

All the examples reported in the paper have been tested in prototype form. This was done to ensure that GlueQoS worked as designed. However, it is important to note that the main contribution is not the feature implementations, but the GlueQoS policy language, policy mediator, and the policy mediation protocol that are used to select the correct set of QoS features and QoS feature parameters. This is in the same spirit as [22], where the goal was to select a transaction protocol for a client-server interaction.

7 Related Work

7.1 Electronic Contracts and SLAs

Our work on QoS feature composition relates to work on electronic contracts[11] and (formal representations of) service-level agreements (SLAs), in particular those that address nonfunctional requirements such as WSLA (Web SLA)[15]. Such contracts define agreed-upon, non-functional (and possibly other) characteristics of (Web) services and a model for measuring, evaluating, and managing the compliance of these characteristics. Their representation involves assertions comparable to our policies, and algorithms for assertion match-making and negotiation have been developed in the context of self-managing systems (systems management) and dynamic e-business.

However, SLAs such as those described using WSLA focus on performance characteristics only, and on locally (nondistributed) measurable phenomena as seen by the client. The service provider's performance offerings are matched with a client's specified expectations to determine a binding contract. Adherence to this contract can be monitored. GlueQoS feature policies address a wider range of end-to-end quality-of-service requirements, such as transactions, security, etc. In addition, GlueQoS has been designed to support policy-driven configuration of client and server middleware to ensure interoperability. SLAs can be used to model contracts that can still be violated. With GlueQoS, an interaction is only executed if a compatible feature composition has been determined, and where no violation should be possible.

7.2 AOP

Research in aspect-oriented software development (AOSD) explores non-traditional software composition approaches to provide flexible extension, adaptation and integration of components. Multi-dimensional separation of concerns (MDSOC) [23] or AspectJ [12] are examples of those compositional approaches. AOSD techniques were first mostly explored in the context of programming languages, they are currently being extended to support aspect-oriented software engineering across the full software lifecycle [1].

Most of the current work in AOSD, however focuses on building services/component with a given set of features. While our work clearly adopts the principle of separation of concerns, it focuses on matching (sometimes conflicting) QoS feature requirements and/or capabilities of interacting distributed QoS components. The middleware mediates between the various requirements/capabilities of the different QoS components to establish and maintain an overall set of potentially context-dependent QoS features (e.g., concerns dependent on deployment, runtime conditions, etc).

7.3 Requirements Engineering

As described in section 4 we envision our middleware to be part of a larger software process. Work on managing conflicts between requirements or features is especially relevant.

The KAOS [25] methodology uses formal specification to detect conflicting requirements using goals which can be identified using a temporal logic. Feather [7] extends the approach for monitoring of actual runtime behavior. Their work is concerned with functional requirements as opposed to mediation of conflicting QoS requirements between software in a distributed setting. Mylopoulos [16] considers the representation of non-functional requirements.

Automatic detection of interacting features in communication systems is an area of active research. Much of the work focuses on reconciling customer features in a telephony setting such as call-waiting and voice mail. Zave [27] views features as modular components connected in a pipeline architecture similar to our layered feature architecture. She identifies an ontology of feature interactions and provides techniques for automatic detection of interactions.

We have not developed any formal techniques for identifying requirements level phenomenon. This paper presents a middleware to be used for mediating policies between QoS features once possible feature interactions have been elucidated.

8 Conclusion

GlueQoS is middleware mediation software to support dynamic adjustment of QoS features between clients and servers. QoS feature preferences are specified in the GlueQoS policy language. These policies are exchanged at binding time between systems interacting in an ad-hoc setting, using the GlueQoS mediation meta-protocol. The polices are then matched up, and resolved by the GlueQoS mediator. The resulting policy resolutions are deployed and executed. In this paper, we have described GlueQoS and provided some illustrative applications. GlueQoS has been implemented in the context of Apache Axis. We have built some sample QoS features and tested that they are deployed according to some sample policies.

References

[1] Concern Manipulation Environment, http://www.research.ibm.com/cme.

[2] Ariba, IBM, and Microsoft. Web services definition language (WSDL), March 2001.

[3] BEA, IBM, Microsoft, and S. AG. Web services policy framework (WS-Policy), May 2003.

[4] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *International Conference on Distributed Computing Systems*, 2001.

[5] D. Dean and A. Stubblefield. Using client puzzles to protect tls. In *USENIX Security Symposium*, 2001.

[6] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *AOSD*, 2002.

[7] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD*, 1998.

[8] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[9] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[10] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Workshop on Self-Healing Systems*, 2002.

[11] Y. Hoffner, S. Field, P. Grefen, and H. Ludwig. Contract-driven creation and operation of virtual enterprises. *Computer Networks 37(2)*, 2001.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[13] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. C. Murphy. Open implementation design guidelines. In *International Conference on Software Engineering*, pages 481–490, 1997.

[14] J. Kienzle and R. Guerraoui. Aop: Does it make sense? - the case of concurrency and failures. In *ECOOP*, 2002.

[15] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. A service level agreement language for dynamic electronic services. *Journal of Electronic Commerce Research, Vol. 3, Issue 1,2*, 2003.

[16] J. Mylopoulos, L. Chung, and B. A. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering*, 18(6):483–497, 1992.

[17] P. Narasimhan, L. Moser, and P. Mellior-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, July 1999.

[18] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using qdl to specify qos aware distributed (quo) application configuration. In *International Symposium on Object-Oriented Real-time Distributed Computing*, 2000.

[19] E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans*. Wiley, 2001.

[20] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*.

[21] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[22] S. Tai, T. Mikalsen, E. Wohlstadter, N. Desai, and I. Rouvellou. Transaction policies for service-oriented computing. In *Data and Knowledge Engineering Journal: Special Issue on Contract-based Coordination and Collaboration*, 2004.

[23] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceeding of the 21st International Conference on Software Engineering*, 1999.

[24] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jorgensen. Dynamic and selective combination of extensions in component-based applications. In *International Conference on Software Engineering*, 2001.

[25] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 1998.

[26] E. Wohlstadter, S. Jackson, and P. Devanbu. Dado: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *ICSE*, 2003.

[27] P. Zave. An experiment in feature engineering. *Programming Methodology*, 2003.