Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures

Jari Koistinen, Aparna Seetharaman
Software Technology Laboratory
HPL-98-51 (R.1)
July, 1998

E-mail: [jari,aparna]@hpl.hp.com

quality-of-service,
negotiation,
agents,
distributed
object systems

Quality-of-Service (QoS) encompasses a wide range of non-functional characteristics, including reliability, security, and performance and is becoming increasingly important in business-critical distributed systems. Such systems are increasingly deployed in open networks, such as the Internet, where resource allocation and load varies highly. To provide useful functionality and meet QoS requirements in such an environment, systems need to be able to dynamically (re)configure and adapt to changing QoS conditions.

This paper describes a model for a QoS negotiation mechanism. The model allows clients and servers in distributed object systems to negotiate for QoS agreements involving multiple QoS categories, such as performance, reliability, security, etc. The model includes both a protocol that enables agents to negotiate and a technique for calculating the worth of alternatives. The protocol handles negotiations involving multiple offers and counter offers. It evaluates an offer based on the absolute requirements for QoS levels and their relative worth to the negotiating systems. The protocol and the worth calculations are key features of a general purpose QoS negotiation that involves simultaneous negotiation over multiple QoS categories. We describe how negotiating applications communicate and evaluate offers. In addition, we describe the functions the negotiation mechanism requires from its environment. The protocol has been formally specified and simulated. We are currently investigating implementation approaches and prototypes for the full as well as simplified versions of the described model.

Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures

Jari Koistinen
Hewlett-Packard Laboratories, jari@hpl.hp.com

Aparna Seetharaman
Hewlett-Packard Laboratories, aparna@hpl.hp.com

Quality-of-Service (QoS) encompasses a wide range of non-functional characteristics, including reliability, security, and performance and is becoming increasingly important in business-critical distributed systems. Such systems are increasingly deployed in open networks, such as the Internet, where resource allocation and load varies highly. To provide useful functionality and meet QoS requirements in such an environment, systems need to be able to dynamically (re)configure and adapt to changing QoS conditions.

This paper describes a model for a QoS negotiation mechanism. The model allows clients and servers in distributed object systems to negotiate for QoS agreements involving multiple QoS categories, such as performance, reliability, security, etc. The model includes both a protocol that enables agents to negotiate and a technique for calculating the worth of alternatives. The protocol handles negotiations involving multiple offers and counter offers. It evaluates an offer based on the absolute requirements for QoS levels and their relative worth to the negotiating systems. The protocol and the worth calculations are key features of a general purpose QoS negotiation that involves simultaneous negotiation over multiple QoS categories. We describe how negotiating applications communicate and evaluate offers. In addition, we describe the functions the negotiation mechanism requires from its environment. The protocol has been formally specified and simulated. We are currently investigating implementation approaches and prototypes for the full as well as simplified versions of the described model.

1. Introduction

1.1 Background

We use the term *Quality-of-Service* (QoS) to denote various non-functional characteristics, such as reliability, performance, security, etc. Quality-of-Service has become a differentiating factor for distributed applications and systems. Enterprises are increasingly dependent on the effectiveness of distributed systems for their businesses and daily operations. In addition, systems are more frequently deployed in open network environments, such as the Internet, that cannot be controlled and that have constantly varying levels of QoS. Workers are increasingly mobile, perform their work from many different network connection points and expect the applications to handle the varying levels of QoS accordingly. These are some of the reasons why applications and systems must be able to adapt to changing QoS levels. We also expect systems to be deployed under widely varying QoS conditions. Such systems must also be able to provide reduced services when ideal QoS conditions are not met.

The increased attention on QoS and the variability of open, mobile, dynamic systems suggest that systems must be able to adapt dynamically to changing QoS conditions and end-user requirements. To adapt, an application should explore the services in its environment and determine the levels of QoS it can expect and thereby provide. It must also agree upon the levels of QoS with the servers that it uses and the clients that use it. Such agreements may be negotiated dynamically and take into account many different QoS aspects such as reliability, performance, security etc. We use the term *category* generally for QoS aspects such as performance, reliability, or security. Categories typically consist of multiple dimensions. A *dimension* represents a quantitative or qualitative attribute of a category. For example, we could characterize reliability as a category in terms of dimensions, such as time-to-failure, availability, failure masking, etc.

Sometimes adaptation and dynamic configuration is a matter of prioritizing specific end-users and their requests. In other situations we may wish to give one

QoS category (such as performance) higher weight than another (such as reliability) depending on the user requirements and changes in the environment.

In our view, dynamic negotiation is one key element to dynamic configuration and adaptation. By negotiation we mean two or more distributed agents that try to reach an agreement on the quality-of-service that they will attempt to provide to each other. The negotiation mechanism we propose is performed dynamically as the system executes. The agents communicate their requirements and their provisions. If they agree, they are said to reach a deal and can continue with the service. Dynamic negotiation allows systems to adapt to changing conditions and to operate with different QoS levels depending on the resources and the QoS available. QoS negotiation is essential if we wish to build evolvable and robust systems that meet the QoS requirements for open networks.

Traditionally, QoS negotiation has been limited to specific domains such as multi-media. Such systems commonly consider only a predefined set of attributes and often only a limited and predetermined set of alternative transports. We believe that a more general approach to QoS negotiation must be taken to allow adaptation with respect to the wide range of QoS categories encountered in open distributed systems. Thus, we are concerned with *multi-category* negotiation.

Negotiation is commonly centered around finding acceptable constraints for specific dimensions such as delay or jitter. Although we consider such constraints essential, we also recognize the need for a more general approach in multi-category negotiation. More specifically, we apply *worth*—sometimes called utility—calculation in addition to satisfying conventional constraints. Worth allows agents to weigh the characteristics of different categories and thereby more accurately guide the negotiation process. As noted by Rosenschein and Zlotkin [16], worth-based negotiation enables agreements in situations where goals are only partially satisfied. Hence, we say that we address worth-based multi-category negotiation.

In addition, our approach differs from most other approaches in that we focus on QoS for operation invocations rather than for data streams.

1.2   Functional and Architectural Context

We consider a QoS agreement as having a life-cycle of essentially three steps: *negotiate*, *execute*, and *terminate*. During the negotiation—the focus of this paper—the negotiating parties either establish a deal, or agree that there will be no deal. If there is a deal, subsequent communication can be performed in the context of the deal. A deal can be time-limited or event-limited or a combination of both. A time-limited deal could last, for example, for 60 seconds, while an event-limited deal can span for 20 invocations of an operation. When communication is performed using the deal, we say that the deal is *executed*. Finally, when the deal ends, it will be terminated. The termination step involves an agreement that the deal is terminated and a decision on whether the deal was successfully executed or not. Loosely, we say that a deal was successfully executed if the involved parties each think that the other parties kept their end of the deal. If this is not the case, a means for finding the party responsible for the broken deal and finding suitable compensation for the other parties is required. In this paper, we focus on the negotiation stage although we occasionally refer to the other stages of QoS agreements.

The negotiation mechanism is the software component that accepts requirements and offers as input, negotiates with its peer and delivers a deal or conflict deal.

Figure 1 shows how the QoS mechanisms fits into a communication infrastructure such as an object request broker. The application uses the infrastructure to create and obtain remote object references, as well as to invoke operations on remote objects. We extend infrastructure interfaces to include QoS capabilities, such as support for negotiation.

We roughly identify three layers in the infrastructure: *Request Dispatcher*, *QoS Components*, and *Transports*. The *Request Dispatcher* is a mechanism that allows applications to handle references and to invoke operations on objects in remote processes. The *QoS Component* layer contains various QoS components such as negotiation mechanisms, monitoring, etc. Finally, the *Transport* layer implements transport mechanisms with various QoS characteristics, such as varying reliability, security, performance etc. Concretely, we can view the *Request Dispatcher* and *Transport* layers as jointly corresponding to a CORBA Object Request Broker.
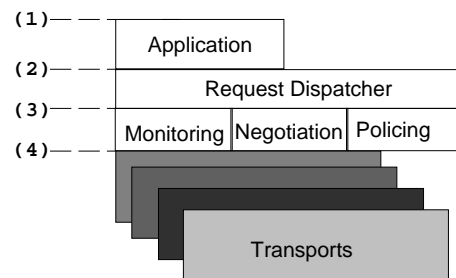

FIG. 1.   QoS Enabled Infrastructure

The important interfaces in the architecture are as numbered in Figure 1.

- **(1)** represents the QoS specification provided to the applications.
- **(2)** is the interface between the application and the infrastructure. It allows applications to invoke and receive operations from other applications. It also

allows applications to specify the QoS they provide and require as well as to initiate negotiations and associate deals with individual invocations.

- **(3)** is the infrastructure's internal interface between the dispatcher and the QoS components.
- **(4)** represents the interface that transports must provide to QoS components to enable filtering of QoS specifications. We will collectively call **(3)** and **(4)** the QoS Abstraction Layer (QAL).

1.3 QoS Negotiation

We use the term *agent* generically for both server and client applications with negotiation capability. We assume that the interfaces for agents are defined in interface definition languages such as CORBA IDL [18] or Microsoft MIDL [8].

We use the term *constraint profile*, or just *profile*, for a QoS specification of a CORBA or DCOM agent. Section 1.5 describes a QoS Modeling Language, QML [6], the language we use for such specifications. We use the term *server profile* for the QoS characterization of a server and *client profile* for the characterization of a client.

By QoS negotiation we mean the processes used by a client and a server to reach an agreement on QoS characteristics for their services considering their expected load, network characteristics and other influential factors. Rosenschein and Zlotkin [16] define automated negotiation generally as:

> the process of several agents searching for an agreement. Agreement can be about price, about military arrangements, about meeting place, about joint action, or about joint objective. The search process may involve the exchange of information, the relaxation of initial goals, mutual concession, lies, or threats.

The negotiation mechanism proposed in this paper is restricted to two negotiating agents where one has the role of a client and the other has the role of a server. We assume that the syntactic and semantic characteristics of the associated services are known and agreed upon. QoS negotiation is necessary for applications to adapt to environments in which resource availability and load varies. In addition, negotiation enables one system to be deployed in environments with different QoS characteristics and therefore enables wider deployment of applications. Negotiation also allows a single server to meet different client QoS requirements.

Although, we view a negotiation as ending in one specific agreement, the agreement might need to be filtered to reflect the requirements of the client and server more appropriately. For example, when we negotiate for a specific performance, we must allow transports to filter the agreement accordingly. The client view of the agreement may state a delay of 100 ms, but the estimated network delay may restrict the actual server execution to a delay of 30 ms. In other cases, such as quality of data, the negotiation can be transport-independent and solely dependent on the capabilities of the server. There is often a direct dependency between an agreement and a transport mechanism. Although this dependency must be handled in a negotiation-enabled communications infrastructure, we consider it outside the scope of this paper.

Dynamic negotiation and subsequent interactions go through the stages described below. Observe that the numbers do not define an ordering, rather many of the stages can be performed in parallel:

1. The server application starts up and registers possible QoS offers that it can provide and the requirements that it must impose on clients to guarantee.
2. The server also registers with a service that allows clients to discover the reference to an object contained in the server.
3. A client application constructs a description of the QoS it will require (server profile) for proper operation and its own expected QoS (client profile).
4. The client describes the *worth* (worth profile) of different QoS levels within the bounds of its absolute requirements. Note that in the current model, the client does not share either its worth profile or its server profile with the server.
5. The client finds a server providing the appropriate functional characteristics (interface) and requests a QoS negotiation.
6. The negotiation components of the client and the server negotiate which results in a *deal* or a *conflict deal*.
7. If there is a deal, the client issues requests to the server and associates a deal with each request. Monitoring components on the client and server sides checks the agreement for compliance. If none, the applications are notified about the problem and asked to either cancel, renegotiate or ignore.
8. When the deal has expired, the parties agree upon whether the deal was successfully executed.

There are many possible variations of negotiation. Sometimes requirements are determined statically. In other cases, the client application and possibly the server application are unaware of any ongoing negotiations because the negotiations have been initiated by an external management agent. In the case of broken deals, there are also many different possibilities such as renegotiation, adaptation or penalties. This paper focuses on steps 3—6 described above.

1.4 Assumptions and Demarcations

In our negotiation model we make the following assumptions and restrictions.

Firstly, we consider only a single client negotiating with a single server with respect to a single syntactic interface. A negotiation involving multiple clients and servers requires generalizations or extensions of the described model and is the subject for future papers.

Secondly, we do not incorporate the trustworthiness of an agent. Trust affects the willingness of agents in establishing deals with each other and should be part of the negotiation model. Incorporating trust into the negotiation model requires the use of mechanisms for establishing and verifying trust. We consider a deeper treatment of trust to be outside the scope of this paper.

Thirdly, two negotiating agents keep their *worth profiles* private. While the server publishes its *constraint profile* and its restrictions on the client, the *client profile*, the client only shares its own *client profile* with the server and nothing else. The client keeps its constraint profile and its worth profile private so that the server does not doctor its constraint profile with such knowledge.

Fourthly, we do not consider the performance implications of a negotiation. Such implications are important when the negotiation consumes a significant fraction of the total deal execution time. Thus, when we negotiate short deals, such as for individual calls, performance may be an issue. However, for deals that last longer, we assume that the performance is acceptable.

Fifthly, for time-limited deals, we assume that the deal is specified only in time periods and not in absolute times. Even with this simplification, we need time that is sufficiently accurate and synchronized to satisfy all involved parties.

Finally, a server makes offers based on its own estimates of how many simultaneous deals it will need to support. If the number of deals exceeds this number, the server needs to handle the situation by, for example, refusing new deals or canceling other low priority deals and paying the penalty for doing so.

1.5 QoS Specifications

The negotiation mechanism requires that we can express the QoS characteristics for an interface. We use the QML [6, 7] language that has been developed for precisely that purpose. QML allows the definition of *contract types* that generally correspond to categories. A contract type defines a set of dimensions, each with a sort and a semantic declaration. A *contract* is an instance of a contract type and defines a specific QoS characterization in terms of constraints or statistical characterizations for a subset of the dimensions defined in the corresponding contract type. A QML *profile* associates contracts with various interfaces elements, such as operations.

Figure 2 shows examples of QML contract types, contracts, and a profile. It contains two contract types

(*Reliability* and *Performance*) and a contract that is an instantiation of the *Reliability* type. We also define a profile called *myServerProfile* for an interface called *myInterface*. We assume that *myInterface* has two operations: *operationOne* and *operationTwo*. The profile associates a default reliability contract for both operations and individual performance contracts.

```
type Reliability = contract {
  numberOfFailures : decreasing numeric no / year;
  TTR : decreasing numeric sec;
  availability : increasing numeric;
};

type Performance = contract {
  delay : decreasing numeric msec;
  throughput : increasing numeric mb / sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 5 no / year;
  TTR { percentile 100 < 200;
        mean < 50;};
  availability > 0.8;
};

myServerProfile for myInterface = profile {
  require systemReliability;
  from operationOne require Performance
          contract {
                delay {percentile 50 < 20 msec;
                percentile 100 < 40 msec; };
            };

  from operationTwo require Performance
        contract {
            delay < 400 msec
        }
};
```

FIG. 2. QML Contracts and Profile

We use the term *constraint profile* to denote QoS specifications corresponding to QML profiles. A *server profile* is a constraint profile that specifies QoS characteristics of the server, while a *client profile* describes the characteristics of the client. Note that both the client and the server have their own representations for both profiles. The client-side server profile describes the client's requirements of the server while the client profile describes its own characteristics. The server-side server profile describes the server's QoS guarantees and its client profile captures the server's requirements for the client.

QML allows us to check whether one contract satisfies another contract of the same type. Assume that we have two performance contracts with a specification of the dimension *delay*. If the first specifies *delay* < 10*msec* and the other specifies *delay* < 20*msec*, the first clearly satisfies the second.

We say that the stronger contract *conforms* to the weaker contract. Using contract conformance, QML also enables the checking of one profile's conformance with another. Thus, it is possible to check whether a QoS specification for an interface satisfies another specification. This capability is necessary for the negotiation mechanism that we propose.

Although the proposed negotiation mechanism is independent of QML, some means of specifying and checking QoS specifications is required in our negotiation mechanism.

We introduce the term *worth profile* for a description of how worth is computed based on constraint profiles. A worth profile allows a *worth calculation* to map a constraint profile to a single worth value that reflects the worth of a particular deal to an agent. Worth profiles will be described further in section 4.

2. Negotiation Mechanism

The negotiation component can be viewed as a black box that drives the negotiation process. To perform a negotiation, it needs to interact with other components in the infrastructure, with the application and with other negotiation components. Figure 3 is a UML [3] diagram showing the principal interfaces of our proposed negotiation mechanism.

Without going into detail, we describe the interfaces as follows:

Agent2Neg : this interface allows client and server application code to interact with the negotiation component. For example, servers may register what they can provide while clients may request negotiations through this interface.

Neg2Agent : this interface allows the negotiation component to issue calls to server and client application code when, for example, a deal has been made, broken or expired.

QoSSpec : this interface allows the negotiation component to traverse and compare QoS specifications. One important function of this interface is to allow the negotiation component to determine if one QoS constraint profile is stronger than or as strong as another. This interface includes both constraint descriptions and worth descriptions of QoS characteristics.

Trans2Neg : this interface allows the negotiation component to let transports annotate and adjust potential deals to reflect the QoS characteristics, when transport-specific characteristics have been taken into account. Network delay is an example of a dimension that needs to be adjusted to enable the negotiation of proper deals.

Neg2Trans : this interface allows transports to register themselves and thereby be considered in future negotiations.
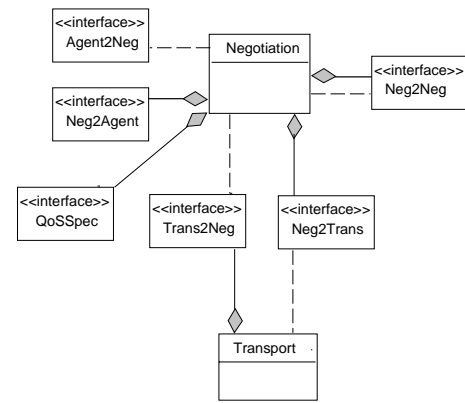

FIG. 3. Negotiation Mechanism Interfaces

Neg2Neg : this is the message interface used by the negotiation component for the actual negotiation. We will discuss this interface in detail in section 3.

Although all of these interfaces are important, we focus only on the internal working of the negotiation component. We also describe how negotiation components interact to reach an agreement/deal (i.e., the protocol for which the *Neg2Neg* interface is an important part). In addition, we describe how applications specify worth and how the negotiation component computes the worth of a potential agreement. Although the other interfaces are essential for the programming model and for understanding the architectural aspects of the negotiation mechanism, we do not describe these interfaces any further in this paper.

3. Negotiation Protocol

3.1 Introduction

The negotiation protocol is the set of messages and procedures that the negotiation mechanism uses in its interactions with its negotiation peer. The protocol that we present allows for both peers to make offers and to end an ongoing negotiation gracefully at any time. The latter is important so that an agent is not locked into a negotiation it believes will not lead to a favorable deal. Each negotiation is parameterized with four parameters: the maximum number of requests that can be made for offers, a boolean argument that determines whether the negotiation mechanism is allowed to make counter offers, the maximum number of counter offers that should be sent and the maximum number of counter offers that will be accepted. We assume that a reliable transport protocol is used for message passing. We also assume that the negotiation function has access to two important sub-functions: *conformance checking* and *worth calculation*.

Before a negotiation can start, the client must have obtained a reference to the server with which it wishes to negotiate. The server must have either registered offers reflecting the QoS levels it believes it can sustain, including some restrictions on client behavior, or be able to produce these offers when requested by the client. The client specifies some absolute requirements that must be satisfied by the server (*the server profile*) and its own characteristics (*the client profile*). If the client makes a request to the server for offers , the client sends its *client profile* over to the server. Furthermore, the client specifies a *worth profile* that describes how the worth of profiles being considered for a deal should be computed.

An *offer* is a tuple consisting of two profiles, a time limit for the offer and the period for which the specified QoS is offered. One of the profiles specifies the QoS that the offering agents promises to adhere to (*the server profile*) and the other profile represents what it requires in return (*the client profile*). A deal is essentially an offer that the two negotiating agents have agreed upon. A deal also has *continuation information* that is part of deal management and tells how the agents should proceed when the deal ends, is violated, etc. We consider deal management outside the scope of this paper since we focus solely on the negotiation process.

Assuming that a client has obtained a reference to a server, we depict a typical negotiation scenario as follows:

1. The client requests the set of offers that the server supports. It supplies the *client profile* to the server so that the server may present suitable offers.

2. If more than one offer satisfies the *server profile* of the client, the client selects the offer with the highest worth and proffers that one to the server.

3. In this case, let us assume that the server cannot accept the chosen offer. It then computes a counter offer that it sends to the client.

4. If the counter offer was acceptable to the client, it sends a *deal* message back to the server.

5. The server acknowledges the deal.

The following two sections describe the messages and procedures of the protocol in more detail. In section 3.4, we discuss both some properties of the protocol and experiences from simulating and verifying the protocol.

3.2   Messages

We can view the messages as defining the interface *Neg2Neg* shown in Figure 3. For our proposed protocol, the messages that can be exchanged between two negotiating agents can be informally described as follows:

request offer set : Sent by client to request all the offers that the server is willing to accept in a deal. The message conveys a *client profile*.

send offer set : Sent by the server in response to a *request offer set* message. The message body contains all the offers that the server supports. The client may select one of these and propose it as a deal to the server.

send offer : Sent by either agent to make an offer (or counter offer) to the other. It can be sent as a response to a *send offer*.

deal : Sent by an agent that received an acceptable offer from the other agent. The message indicates that the agent that received the most recent offer is willing to accept it as a deal.

acknowledge deal : Sent by either agent as a response to the *deal* message.

conflict deal : Sent by either agent as a response to a *send offer* message. The message indicates that the received offer was not acceptable and that the agent does not intend to make a counter offer.

acknowledge conflict deal : Sent by either agent as a response to *conflict deal*.

stop : This message can be sent by either agent to stop negotiating immediately. It can be due to a failure, application shutdown, change of priorities or any other reason.

The message *send offer* conveys an offer consisting of a *server profile* and a *client profile*. The message *send offer set* carries a set of offers provided by the server. All of the messages convey a *transaction identifier*, that becomes the *deal identifier* if a deal is reached. The deal identifier is used to associate individual operation invocations with deals.

The set of messages sent from the application to the negotiation mechanism can be summarized as follows:

abort: Sent by the application to its negotiation mechanism to indicate the termination of all ongoing negotiations for that application.

neg: Sent by the application to initiate negotiation.

regOffers: Sent by server application to register new any offers that provides.

These messages are part of, but do not fully define, the *Agent2Neg* interface. The application can signal its intention to terminate by sending an *abort* message to the negotiation mechanism. This makes the negotiation mechanism stop all ongoing negotiations and refuse any new negotiation requests.

3.3   Procedures

Our protocol simulation models are specified in a specification and modeling language called Promela described in [14]. In this paper, however, we describe the procedures for the client and server negotiation mechanism as Harel state-charts [12]. A box represents a state and an arrow a state transition. Transitions are represented by labels with zero or one incoming or outgoing events and zero or more actions

(inEvent/ ˆoutEvent/action). States can be nested. A transition from an outer state indicates a transition that applies to all the inner states. A circle represents a start state and two nested circles represent a valid end state. Figure 4 and Figure 5 describe the client-side and server-side state machines, respectively.

We can view the state machines as being inside the *Negotiation* class that was shown in Figure 3. Since clients and servers exhibit somewhat different behaviors, we separate them into two different state-machines. Observe that at the described abstraction level, the models ignore how the agents evaluate and select offers and how they decide to propose a deal or a conflict deal. Rather, the models focus on when specific messages can be sent and received.

Normally, the client negotiation mechanism is in an *idle* state. When it receives a negotiation request (*neg* event,) from the client it transitions to *initialOffer*. In *initialOffer* it determines if its server offers are valid or if it needs to request an updated set of server offers. If it already has a valid set of offers, it selects one and makes an offer to the server. The server may accept by sending the *deal* message, turn down the offer by sending *conflictDeal*, or send a counter offer by sending *sendOffer*. If the server sends a counter offer, the client evaluates the offer (indicated by the *incomingOffer* state) and either accepts the offer, rejects the offer, or makes a counter offer by transitioning to *initialOffer*.

The negotiation will always be interrupted if the client receives the *stop* message.
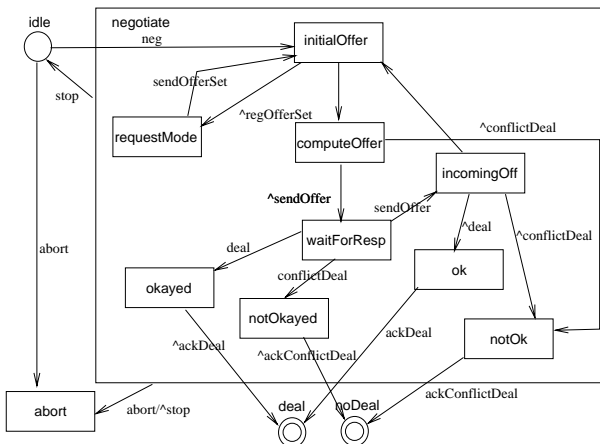

FIG. 4. Client Side Negotiation Procedures

The server-side state machine goes into the *negotiate* state when it receives an offer or a request for an offer set. If it receives a request for an offer set, it collects the offers it currently can support and sends them to the client. If the server receives an offer, then the client chose to propose an agreement immediately as the client already had what it believes is a valid set. When the server receives the offer, it evaluates the offer

and decides whether it accepts, rejects, or counters the offer. The conformance checking of constraint profiles and the worth calculation take place in the *incomingOffer* state for the client and in the *evaluateOffer* state for the server. Essentially, the agent checks whether the offer profile conforms to the profile that expresses its absolute requirements. If it has several acceptable alternatives—such as when the server returns multiple acceptable offers with the *send offer set* message—it computes worths and compares the results.
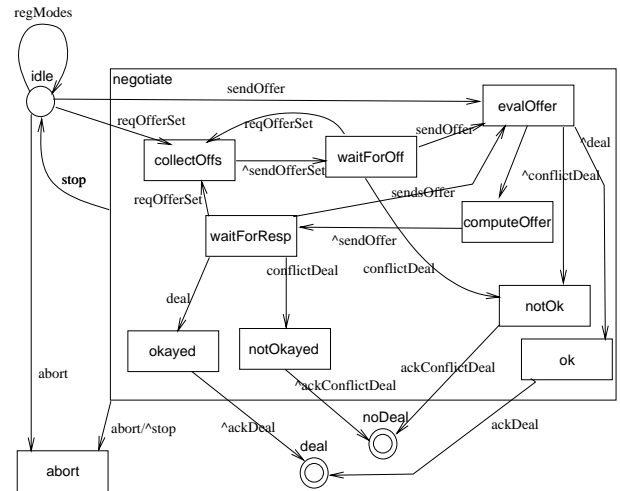

FIG. 5. Server Side Negotiation Procedures

The server application can update the valid set of offers that it supports.

The interaction diagram in Figure 6 shows one possible interaction between applications and negotiation mechanisms. This particular negotiation results in a conflict deal, thus the offer sent by the client is not accepted by the server who wants to discontinue further negotiation.
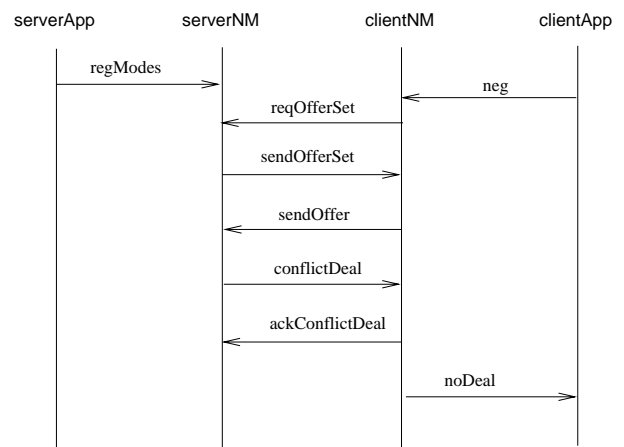

FIG. 6. A Possible Interaction

The protocol has been restricted in several ways in order to keep it and its analysis simple. As an example,

we do not allow an agent to respond with a new offer after the other agent sends a conflict deal message, although such behavior could be desirable in some situations.
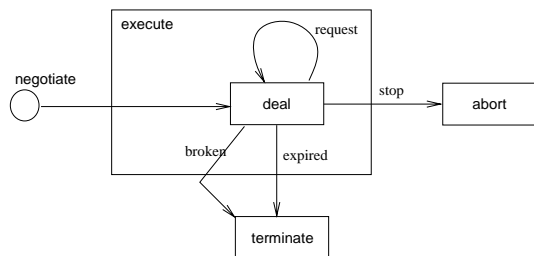

FIG. 7.   Generic Procedure for Execution Phase

When the negotiation mechanisms goes into the *deal* state they also transition from the *negotiation* phase to the *execution* phase. We require that the execution phase can accept *stop* messages, since otherwise a race condition could occur. Assume the server has sent the *deal* message and transitioned to the *ok* state. The client accepts the *deal* in the *waitForResp* state and sends back a *ackDeal* message. At the same time the server receives an *abort* message from the application that causes it to send a *stop* message and transition to the *abort* state. Since the client has transitioned to the *deal* state it can not—according to the description in Figure 4—accept a *stop* message. However, we view the transition to the *deal* state also a transition to the *execution* phase of the agreement life-cycle. This phase requires both clients and servers to maintain deal information and to accept a variety of messages. Although this phase is not the focus of this paper we will outline a state machine for this phase in Figure 7.

The negotiation mechanism must also handle the case when one party sends a *stop* message and transitions to *abort* while the other party sends a negotiation message before it receives the *stop* message. This negotiation message will arrive to the original stop sender when it is incapable of accepting any of the negotiation messages. This is, however, not a problem and can be handled by for example allowing the negotiation component to discard negotiation messages for all messages with deal identifiers that do not match any of the active ones.

3.4   Discussion

We used Promela and SPIN [14] for simulation and verification of the protocol. Promela is a simple, CSP-like [13] language in which the model is described as a set of sequential processes. The descriptions can be annotated with semantic annotations and invariants. Choices—such as deciding whether an offer is good or not—are modeled as indeterministic choices. A model can be simulated and validated. The validation pro-

cedure uses semantic annotations and invariants to detect problems such as live-locks, deadlocks, inconsistent states, etc.

Besides pointing out some errors in earlier versions, the models also identified a few more serious problems, which we address below.

Firstly, since the protocol allows counter offers, a negotiation can degenerate into an infinite exchange of offers. Ideally, we would like to detect non-progress in a negotiation by comparing the worth of a new offer with the worth of previous offers. Since agents do not share their worth functions, this solution is not viable. Without the worth functions of the other agent, an agent has no means of ensuring that it indeed makes a concession, i.e., increases the worth for the other agent. On the other hand, sharing the worth profiles would make an agent more vulnerable to cheating. Our solution is to parameterize each negotiation with the maximum number of counter offers that may be sent and received. The client side is also parameterized with the maximum number of *request offer set* messages that may be issued. These restrictions allow agents to set a maximum bound on the number of negotiation rounds that may take place. Also, a parameter on the server-side capturing the maximum number of requests for offers restricts the client from swamping the server with requests.

Secondly, it must be possible for either agent to enforce the end of a negotiation session. A general purpose *stop* mechanism may be used to do this. This message does not require any acknowledgment from the other agent and hence does not block the agent.

After the model was enhanced with these parameters, the simulations and validations indicated that the model defines a well-behaved negotiation protocol.

4.   Worth Calculation

4.1   Introduction

According to our protocol, the client receives several QoS offers for a service. It then either selects one of these offers to propose a deal or it proposes a conflict deal.

As part of the selection process, the client agent first applies conformance checking to obtain the set of server offers that meet its absolute requirements. This set may contain more than one offers. In order for the client to pick the offer that best satisfies its requirements, we need a means for assessing offers.

We propose that this be done by using a worth function that computes the worth of an offer based on the client's current preferences. This requires that we provide a means for specifying the relative worth of different values for specific dimensions and categories (con-

tracts in QML). The client may assign a higher worth to those values of a dimension that are more desirable and a lower worth to those values that are less desirable. The client may also assign worth to contract types to indicate their relative importance to the client. Finally, the client may assign different worth to the different operations in an interface to indicate that it prefers to obtain a better QoS for some of the operations.

One important criteria in the design of the worth calculation was that agents should be able to use worth-based negotiation without having to explicitly specify worth profiles for all entities and contracts in a profile. To enable this, there must be a reasonable default worth specification that can be used when worth has only been partially defined by the application. The second criteria was that when a worth calculation is specified, it should be necessary to only specify those portions of the worth profile that are different from the default.

4.2 Worth Functions

We view the worth calculation as a function ($profileWC$) that takes a server profile (from an offer) and a worth profile as input arguments.

$$profileWC : (serverProfile \times worthProfile) \rightarrow worth$$

The result of the calculation is the *expected* worth of the server profile. This worth function uses other functions that calculate the worth for individual operations. The operation worth function in turn uses functions that calculate the worth of contracts.

The worth calculation for operations and contracts is, to a large extent, concerned with assigning weights. When calculating the worth of a contract, however, the worth function by default, assigns different values for individual dimensions. The expected worth is then calculated as the integral of that function over what is specified in the server profile.

Our model enables agents to provide their own contract worth functions. Such functions could be defined in terms of worth functions for dimensions, but in order to reduce the number of worth functions that must be specified, we only specify at the granularity of contract worth functions. The signatures below represent the worth calculation functions that take customized worth functions as input and produces an expected worth:

$$operationWC : (contracts \times worthFunc) \rightarrow worth$$

$$contractWC : (dimensions \times worthFunc) \rightarrow worth$$

Figure 8 shows an example of a worth function for a dimension called *availability.* We think of the dimension as being part of a *reliability* contract that also has other dimensions such as *MTTF, MTTR,* etc.

The availability worth function essentially maps different availability values to different worths. Assume that a server profile guarantees that the availability

will be between a and b with an uniform distribution. Our expected worth can then be calculated as the mean worth in that interval.
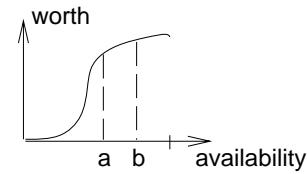

FIG. 8. Agent Specific Dimension Worth Function

Figure 9 shows the default worth function for *availability* used by the negotiation mechanism. The default worth function assumes equal worth for all acceptable values of the dimension.

The worth of an operation is computed by invoking the appropriate contract worth functions and weighing the worths of contracts according to agent preferences (see hierarchy of worth functions in Figure 10). Finally, we weight the worth of different operations according to agent preferences. Our model enables agents to supply their own contract worth functions, contract weights for each operation, and operation weights. Although we focus on weights in this description, we do support a more general notation of worth functions at the operation and profile levels as well.

The specification of the weights and worth functions for contracts associated with specific operations is called a *worth profile.*

The worth function for dimensions can be arbitrarily complicated, depending on the worth function and the statistical distributions of the input profile. In the following sections, we omit any discussions on statistical computations. Instead, we focus on how the results of such computations are aggregated to a worth for a profile.

4.3 The Worth Calculation Process

The goal of the worth calculation process is to produce one numeric value that represents the *expected worth* of the server profile using the supplied weights and worth functions. The expected worth reflects the worth of a specific *constraint profile* to a client with respect to a specific *worth profile.* It is called expected worth since it is based on the expected statistical distributions of dimensions.

Before we start computing, we identify the contracts for each operation in the server profile. We then omit those contracts that do not have a corresponding contract in the constraint profile of the client. A contract for an operation in one profile corresponds to a contract in another profile if they are of the same contract type. If a contract appears in the server profile, but does not have a corresponding contract in the client's constraint

profile, then the server has made guarantees about a QoS category that the client does not require. Therefore, we also assume that the client is not assigning worth to contracts that do not appear in its constraint profile. Thus, we can omit these contracts from our worth calculation process.

We are left with the contracts for each operation in the server profile that have a corresponding contract in the client's constraint profile. Now, we apply the corresponding worth function to each of the contracts to obtain the worth of each contract. We also normalize the worth of the contract by dividing it by the sum of the worths of each dimension defined in it to obtain the expected worth of each contract.

Next, we calculate the aggregated worth of an operation by multiplying the weights of each contract (as specified by the operation worth function) with their calculated expected worth and by summing all the results. We then obtain the expected worth of an operation by dividing its worth by the sum of the weights of all the contracts defined for it.

Finally, we multiply the expected worth of each operation by the weight associated with the operation. The aggregated worth of a profile is then the sum of all the weighted worths of these operations. The expected worth of a server profile is obtained by dividing its worth by the sum of the weights of the operations defined in it.

A more formal description of the algorithm for calculating the expected worth is given in Figure 11. In Figure 11, SP stands for server profile, CP for constraint profile, EW for expected worth of a server profile, wf_c for the worth function that calculates the worth of a contract, wf_o for the worth function that returns the weight of a contract in an operation and wf_p for the worth function that returns the weight of an operation.
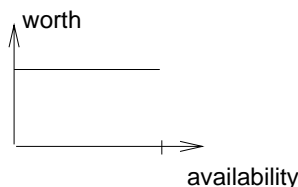
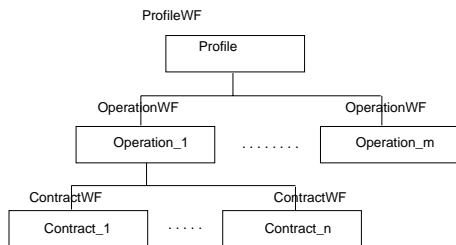tion. The function $Sums(X)$ returns the sum of the weights/values of all Xs. That is, if $X = C$, then $Sums(X)$ returns the sum of the values of every dimension in every contract. If $X = O$, then the function returns the sum of the weights of all the contracts. Finally, if $X = P$, then $Sums(P)$ returns the sum of the weights of all the operations.

The expected worth calculation algorithm may be further refined to include a threshold such that only those server profiles that have an expected worth above a certain threshold are considered. This enhancement is easily made by adding an extra statement at the end of the current algorithm that checks if $EW > Th$, where Th represents the threshold. If it is, then the function may return EW. If not, the function may return a null to represent an unacceptable server profile. Note that null may be used even without the introduction of the threshold.

In the absence of an agent-defined worth profile, a default worth profile is used. The contract worth functions in the default worth profile return one as the worth of each dimension. The default weight for each contract and each operation is also set to one. Thus we would obtain a default expected worth of one for each server profile. This, of course, means that if we only have default profiles, there is no need to calculate the worth; rather, we can assert that all server profiles have the same worth and an arbitrary choice can be made.

If an agent-defined worth profile has been supplied, it overrides the default profile. The agent-defined profile need not be complete; rather, it may override selected portions of the worth profile. The calculation algorithm uses the agent-specified weights and functions for the selected portions of the worth profile and the default values otherwise.

4.4 Worth Specification

FIG. 9. Default Dimension Worth Function

FIG. 10. Worth Function Hierarchy

```
Begin
  EW = 0; Worth_p = 0;
  For all operations O in SP
  {
    Expect_o = 0; Worth_o = 0;
    For all contracts C for O in SP and CP
    {
      Expect_c = wf_c(C) / Sums(C);
      Worth_o += wf_o(C) * Expect_c;
    }
    Expect_o = Worth_o / Sums(O);
    Worth_p += wf_p(O) * Expect_o;
  }
  EW = Worth_p / Sums(P);
End;
```

FIG. 11. Expected Worth Calculation Algorithm

To specify worth, an agent needs to be able to specify two different aspects: the worth functions and weights for contracts, operations, and profiles; and the individual functions along with how and where the functions are used.

Worth functions can be defined in a variety of ways including implementing regular programming language functions. There might even be specialized languages developed for this purpose. Such functions would take a contract as input and produce a worth. For this paper, however, we assume that such functions can be defined by the agents that are available to their negotiation components.

The other aspects that need to be defined are how the worth is aggregated, what specific worth functions are to be used for individual operations, etc. We call these aspects the *worth profile*. Although we have developed a prototype language for this aspect, we will omit describing this in detail in this paper. We will, however, provide some simple examples to illustrate the idea.

Since our mechanism allows negotiation even when there are no explicit worth profiles specified, there needs to be a default *worth profile*. Assume that we have a simple interface called *myInterface* with two operations *op1* and *op2*. In addition, we are using only contracts of type *C1* and *C2*. The default worth profile—shown in Figure 12—would then specify that for the contract types we use the default contract worth function, and that the operations have equal weight of one(no weight defined). To aggregate the contract worths for individual operations, we also use a default function that gives contracts of different type the same weight.

If we explicitly define a worth profile, it selectively overrides functions specified by the default worth function. In Figure 13 below, we define a profile that replaces the worth function for contracts of type *C1*. In addition, it gives the contract of type *C2* a weight of two in the computation of the worth for *op1*. In all other cases the implicitly define default worth profile still holds.

```
DefaultWP for myInterface = worth {
  contract type C1 use DefaultContractWorthFunction;
  contract type C2 use DefaultContractWorthFunction;
  entity op1 use DefaultOperationWorthFunction;
  entity op2 use DefaultOperationWorthFunction;
};
```

FIG. 12.   Default Worth Profile

```
DefaultWP for myInterface = worth {
  contract type C1 use myCWorthFunction;
  entity op1 weight 2 for C2;
};
```

FIG. 13.   A Worth Profile

This model allows agents to selectively change parts of a worth profile without needing to redefine the entire worth profile. The example given is very brief; a full treatment of our prototype language is outside the scope of this paper.

4.5   Discussion

A client agent may already know how to utilize a service provided by the server. It may know which operations in an interface are going to invoked, how often it will invoke them, the limitations and advantages of the environment from which it invokes a service, etc. By allowing the client to assign worth to the values of dimensions, contract types and even the operations, we facilitate the making of intelligent deals between the agents maximizing the throughput of both the client and the server.

This worth calculation scheme allows calculation and thereby negotiation even when the client has not supplied any worth functions. If desired, however, agents may replace default worth functions arbitrarily, allowing them to change the worth calculation with minimal effort.

One question that can be raised, however, concerns the process an agent (usually a designer or end-user) should use to define worth functions and weights. Although we do not address this issue explicitly, we have been investigating it and believe that conventional decision theory techniques—such as multi-attribute decision theory [19]—can be used for this purpose.

For the purpose of this paper, we have mainly described how the worth for operations and profiles are specified by assigning weights. We do, however, support the more general notion of providing worth functions that aggregate the worths of operations and profiles. This allows us to capture dependencies between contracts. As an example, we might specify that we are prepared to accept lower performance, if the reliability is better, but not otherwise. Such dependencies can not be described by weights alone. This extension is applied by associating worth functions with operations and profiles rather than just with weights.

Although we have described the worth calculation as a client evaluating server profiles, it need not be restricted to this. The schema could be used more generally for any situation in which an agent evaluates the offer made by another. Similarly, the worth functions are not dependent on QML. Rather, any specification language may be used to capture QoS. Worth functions can be applied to any QoS specification.

The worth calculation mechanism will undoubtedly impose a performance overhead. We expect to be able to report more on this as the implementation work has been is finished. We have, however, limited worth calculations to clients to avoid imposing too much compu-

tational overhead on the server before the performance characteristics are more thoroughly understood.

5. Example

5.1 Introduction

To illustrate the negotiation process, we use a small fictitious example. The scenario for the example is a foreign exchange trading system. The user is a manager of funds who needs to access the system from many different locations. He uses a laptop computer that connects to a network providing him access to many foreign exchange trading services.

The application on the laptop provides the user interface. The application uses a *QuoteServer* and a *TradingServer*. The *QuoteServer* provides foreign exchange rates and is the primary information source for the human user. The *TradingService* provides an entry point to a banking system that allows the user to buy and sell currency.

Generally, the user would want reliable, secure, and fast access to both information and trading. But when faced with resource scarcity, the user could prioritize availability and performance of the *QuoteServer* over other characteristics such as security because the user always needs recent and accurate information, but may perform trading by other means, e.g., by calling the bank. Depending on resource availability, the application may make tradeoffs with respect to the QoS characteristics of the *QuoteServer*.

To achieve this, the client application could specify its absolute requirements (a *server profile*), its expected behavior (a *client profile*) and the relative worth (a *worth profile*) for various QoS characteristics. Using these, our proposed negotiation mechanism would be able to negotiate the best possible QoS agreement with respect to the user's preferences. The negotiation would be performed without any user intervention. Naturally, if the absolute constraints could not be meet on either side, the negotiation will not reach an agreement. In such a case, the user could agree to weaker requirements or take actions that enable a stronger agreement, such as moving to a faster network connection.

In the example below, we show what such specifications may look like and how a negotiation mechanism might use them. For simplicity, we only consider negotiation for the *QuoteServer* service.

5.2 Specifications

The IDL interface specification of *QuoteServer* is provided in Figure 14.

```
interface QuoteServer  {
    rate getRate(in Currency from, in Currency to);
    sequence<rate> getRateSeq(in Currency from,
                                 in Currency to,
        in TimeI ti);
};
```

FIG. 14.   QuoteServer interface

In order to specify the desired and provided QoS, we need to define QML contract types. Figure 15 describes three simplified contract types. *Reliability* has two dimensions, one for *Time-To-Repair* and one for *availability*. *Performance* has only a single dimension, *delay*. Finally, *Security* has one dimension for characterizing *anonymity* and another for *encryption* level. The *anonymity* dimension has two values denoting whether anonymity can be provided or not. For the purpose of this example, the *encryption* dimension provides only different levels of encryption. A higher level indicates that it takes a longer time to crack an encrypted message.

```
type Reliability = contract {
  TTR : decreasing numeric sec;
  availability : increasing numeric;
};

type Performance = contract {
  delay : decreasing numeric msec;
};

type Security = contract {
  anonymity : decreasing enum { true, false};
  encryption : increasing
              enum {high, med, low, none}
              order {none < low, low < med,
                              med < high};
};
```

FIG. 15.   Contract Types

In Figure 16, we introduce contracts that reflect different levels of QoS. The contracts *highRel* and *lowRel* reflect higher and lower levels of reliability. Similarly, *highSec* and *lowSec* reflect different levels of security. We combine these contracts in profiles to specify different combinations of security, performance, and reliability levels.

Figures 17, 18 and 21 define different server profiles with high, medium and low performance guarantees respectively. The profile in Figure 17 specifies low security and reliability but high performance guarantees. Observe that in contrast to the security and reliability contracts, we do not name the performance contract;

instead, we define it as a set of values within the pro-

files.

Figure 18 defines a profile with a high reliability guar-

antee. However, the server is not be able to provide the

```
highRel = Reliability contract {
  TTR {
        percentile 100 < 200;
        mean < 50;
        variance < 0.3
      };
  availability > 0.8;
};


lowRel = Reliability contract {
  TTR {
        percentile 100 < 2000;
        mean < 1000;
        variance < 0.3
      };
  availability > 0.4;
};

highSec = Security contract {
  anonymity = false;
  encryption > low;
};

lowSec = Security contract {
  encryption < high;
};
```

FIG. 16.   Contracts

```
qsFast for QuoteServer = profile {
  require lowRel;
  require lowSec;
  from getRate
        require Performance contract {
                      delay < 40 msec;
                    };
};
```

FIG. 17.   Profile with high performance guarantees

```
qsMedium for QuoteServer = profile {
  require highRel;
  require lowSec;
  from getRate
        require Performance contract {
                      delay < 70 msec;
                    };
};
```

FIG. 18.   Profile with medium performance guarantees

```
clientHigh for QuoteServer = profile {
  require myHighRel;
  require mySec;
  from getRate
        require Performance contract {
                      delay < 350 msec;
                    };
};
```

FIG. 19.   Client Requirements

```
clientLow for QuoteServer = worth {
  entity getRate weight 5 ;
  contract type Performance use cPerfWort;
};
```

FIG. 20.   Client Worth Profile

same performance guarantee as that specified by the *qs-Fast* profile and the security guarantee is weak as well.

The server mode specified in Figure 21 provides significantly lower performance than *qsFast* and *qsMedium*, but in return we get both higher reliability and security guarantees. The reason for the lower performance is, of course, that the reliability protocol and the encryption generate considerable computational overhead. Figure 19 specifies the client's QoS requirements. The client requires that the server satisfy the contracts *myHighRel* and *mySec* for both *QuoteServer* operations. In addition, a constraint is imposed on the delay of the *getRate* operation. Without going into the details of the contracts, we assume that both *lowSec* and *highSec* conform to *mySec*. In addition, we assume that *highRel* conforms to *myHighRel*, and *lowRel* does not conform to *myHighRel*.

Figure 20 describes the worth profile for the client. The worth profile gives *getRate* a higher weight than the other operations of *QuoteServer*. This means that the differences in worth for the different profiles is magnified for *getRate*. In addition, we provide a worth function for *Performance* contracts. Rather than defining the worth function in detail, for the purpose of this paper we assume that the function gives a higher worth to better performance.

The previously presented profiles focus on the QoS that the server can provide to the client and the QoS

```
qsLow for QuoteServer = profile {
  require highRel;
  require highSec;
  from getRate
        require Performance contract {
                      delay < 300 msec;
                    };
};
```

FIG. 21.   Profile with low performance guarantees

that the client requires from the server. A QoS agreement goes two ways, which means that the server may not promise any QoS without knowing how the client is going to be use it. Thus, each profile contained in an offer by the server also has an associated requirement on the behavior of the client. In our negotiation model, the QoS provided by the client is also specified as a QML profile. These specifications typically involve call frequency and distribution, complexity of the arguments, etc. For the simplicity of this example we omit describing these profiles in detail.

5.3   A negotiation

Assume that the human trader connects his laptop to the network and starts the *TraderStation* software application. The application connects to a repository from which it obtains references to the *QuoteServer* and *tradingService* services. The application then attempts to negotiate a QoS agreement with the reference to satisfy the user's requirements for timeliness of data, reliability etc.

We describe a possible negotiation scenario as shown below. This scenario is simple and straight-forward scenario, in contrast to one complicated by including multiple offers.

1. The client sends its *client profile* to the server with a *request offer set* message. The client profile specifies that the client will issue 300 calls per hour to *getRate* and five calls per hour to *getRateSeq*.

2. The server selects modes that it can provide, assuming the specified client behavior. In this negotiation scenario, this set consists of *qsLow*, *qsMedium* and *qsFast*. The server sends these modes to the client with the *sendOfferSet* message.

3. The client checks whether the profiles it received conforms to its server profile (*clienthigh* (shown in Figure 19). It concludes that *qsLow* and *qsMedium* do conform to *clientHigh*. The profile *qsFast*, however, does not and will be removed from the client's set of potential agreements. *qsFast* does not conform since it provides *lowRel* which does not conform to the reliability contract *myHighRel* (as we assumed in section 5.2) required by the client.

4. The client is left with two acceptable server profiles and needs to select one. The client calculates the worth of the two remaining profiles based on the worth profile described in Figure 20. Since the performance for *getRate* is better in *qsMedium* it gets a higher worth. The difference is magnified because the worth profile gives *getRate* a higher weight.

5. Based on the worth calculation the client selects *qsMedium* and sends it back to the server with a *send offer* message.

6. The server receives the offer and verifies whether it can still accept it and reach an agreement. It then sends back a *deal* message to the client indicating that it is prepared to strike a deal.

7. The client receives the *deal* message and acknowledges it by sending an *acknowledge deal* message to the client. The identifier that passes with each of the negotiation messages is now the unique identifier for the deal. This identifier is used with subsequent operation calls.

5.4   Discussion

This example illustrates the basic principle of our negotiation mechanism although we have omitted several specifications and detailed descriptions of computations in order to keep the example simple.

In a more complex situation, the client and server may dynamically create new offers that are presented as counter offers. Such behavior requires that an application be QoS-aware and obtain information about the conditions of its environment, current load, etc.

The example negotiation above seems to send profiles over the wire multiple times. A smart protocol implementation would optimize by sending profile identifiers once the profiles have already been transmitted.

6.   Related Work

Most existing QoS negotiation protocols, such as GSM [10], QoS-A [11], NAFUR [9] and NRP [5, 1] are focused on multi-media applications and deal with only a restricted number of QoS parameters. Furthermore, most of these protocols regard negotiation as the process of reserving resources. We argue that QoS negotiation can be viewed at a higher level and be more generic. We view an application as being capable of carrying out many services, each at different QoS levels. Negotiation for a service is then the process of choosing one of the many QoS levels associated with that service based on worth, that both the client and server applications agree upon. Resource reservation is then performed to support the agreement. None of the negotiation protocols that we have reviewed allows counter offers when initial offers are unsuccessful. This prohibits them from continuing the negotiation if an initial mismatch occurs even if a potential for useful agreements exists. NAFUR and NRP differ from other related work in that they attempt to address some of the negotiation concerns in a more generic manner.

In NAFUR, Hafid et al. [9] present a negotiation model for distributed multi-media applications. In their model, they present representations for offers and deals as well as a negotiation protocol that allows future reservations. When a request can not be satisfied, NAFUR allows the server to propose a later time when the re-

quest can be fulfilled. NAFUR includes a description of how offers can be related, e.g., one offer being stronger than another. Although we also consider such relations necessary, we rely on the general QML [6] QoS specification language for such comparisons. The model described by Hafid et al. [9] includes a model for both how a future load is projected and how offers are computed. In contrast, we consider such algorithms as agent-specific and outside the scope of a generic negotiation model. We do, however, consider the generic computation of worth within the scope of the negotiation mechanism. NAFUR supports the distinction between the time at which a request is made and the time it is serviced. We agree that this is an important distinction and our model consequently supports a similar distinction, although we treat such parameters as any other QoS dimensions. NAFUR, like the model proposed in this paper, includes the duration of a service in the negotiation. NAFUR [9] does not support probabilistic or statistical values for any of its attributes; neither does it provide groupings for the subsets of logically related attributes as we do using QML. The NAFUR approach focuses on providing acceptable service, but has no notion of worth; rather it focuses on the constraints associated with an offer.

Dermler et al. [5, 1] present a negotiation and resource reservation protocol (NRP) for end-to-end QoS reservations in distributed multi-media applications. NRP is general in the sense that it can support different application architectures, i.e., different topologies of video stream sinks, sources, and mixers. The NRP protocol allows components to specify their capabilities and merge them into specifications of capabilities supported by multiple nodes. Assume, for example, that an application uses a mixer conncted with two cameras. The resolution provided by the mixer is dependent on both cameras and thus is limited to the resolution supported by both cameras. To allow end-user reservation, Dermler et al. propose an additional application-specific QoS reservation layer on top of NRP. This layer allows an application to specify requirements independent of individual components. Rather, the layer translates end-to-end requirements to NRP reservations. It is not clear how generic the NRP protocol is; it has clearly been designed with multi-media applications in mind and is therefore highly specialized for such applications. In contrast to the approach proposed in this paper, NRP does support the determining of the level of QoS provided by intermediate nodes. This capability of NRP is dependent on the components knowing the QoS dimensions that are combined. It also assumes a high-level understanding of the functions of the components whose capabilities are combined. As an example, they define a mixer as a component with multiple in-streams and only one out-stream; thus we can figure out that the in-streams must be synchronized to produce an out-stream. In contrast, we are addressing the general need for distributed applications and can not make assumptions about the specific functions of components. Consequently, our approach does not address the problem of deriving—based on the components used—the level(s) of QoS of a client component. Rather the component (agent in our terminology) needs to understand how the different levels of the received QoS levels affect the QoS levels it provides.

In summary, we view NRP as mainly a reservation protocol with some elements of negotiation. Dermler et al. [5] discuss QoS negotiation in terms of using the intersection between the client's QoS and the QoS of all the components involved, but does not offer alternatives or worth assessment and is application-specific.

Although most QoS mechanisms are limited to multi-media streams and do only a limited QoS negotiation, there are other types of QoS negotiation mechanisms. As an example, Chen et al. [4] propose a mechanism that allows applications to adjust to changing bandwidth and failure rates in wireless networks.

In [15], Rajahalme et al. introduce negotiation for a predefined set of both attributes and transports in wireless networks. The negotiation selects media codings and session protocols for all the terminals involved in a particular communication session. They focus mainly on data streams and assume a predefined parameter set for the negotiation.

In the area of artificial intelligence (AI), work has been done on agents that negotiate to perform interacting tasks and reach intersecting goals. These models provide an excellent basis for understanding and analyzing various negotiation scenarios. Rosenschein and Zlotkin [16] describe three different negotiation models in the AI realm. The first model is task-oriented and addresses the problem of dividing a set of tasks among a set of agents. The goal is to find a solution in which every agent benefits from the agreement. The second model is state-oriented. It is more general and involves a common world state that can be manipulated by any one agent. The goal is to agree upon both a final world state and the means of reaching it. There are situations in both the task and the state-oriented domains that may result in a conflict deal because the benefits are calculated based on the cost to the agent if it were doing its own tasks. In the third model Rosenschein and Zlotkin introduce worth, which allows deals that could not have been reached with task or state-oriented models. This model—the worth-oriented model— allows agents to assign a worth to a deal independent of other alternatives. It is therefore possible to reach deals in those situations for which state or task-oriented models would result in a conflict deal. Rosenschein and Zlotkin describe different negotiation models for these three domains and their associated problems and possibilities. On an abstract level, the state and worth-oriented domains match QoS

negotiation quite well, if we view the global state as the common QoS agreement. We have drawn much inspiration from their descriptions of these models. There are, however, some fundamental differences between their models and what we believe is necessary in QoS negotiations. Rosenschein and Zlotkin assume that all agents can perform the same set of operations, that agents do not commit themselves to some future behavior, and that past history does not affect an agent's decision. We do not believe that such assumptions are realistic for QoS negotiations in open distributed environments. As an example, we believe future reservations are necessary in real applications. In an open system, we need to consider how well agents have fulfilled their commitments in the past, thus making trust important. Although our model does not currently include trust, we do believe it is essential to the future development of QoS negotiations. Moreover, all the protocols presented by Rosenschein and Zlotkin attempt an agreement at the time that the request is made. They do not distinguish between the time the request was made and the time that the request is serviced. Neither do they consider the duration of the service as a factor for QoS negotiation. Rosenschein and Zlotkin do recognize—and we agree—that a separate negotiation oracle will not provide a scalable solution. Thus, the negotiation mechanism should have a protocol that ensures progress in the negotiation without using a single point for offer evaluation, conflict resolution, etc.

In contrast to the majority of existing implementations of negotiation models, the model proposed in this paper is intended to be general. We believe that there is a need for a generic QoS negotiation protocol that is application-independent, supports multiple QoS categories and can accept fully satisfactory agreements. Moreover, we have found that none of the existing protocols describe a language for specifying QoS constraints or worth in a general way. Nor are there any that present algorithms that show exactly how different alternatives are compared, how choices are made and how contracts are agreed upon. We not only provide these algorithms, but we also allow the user to replace any of them to support different applications and user preferences.

7. Implementation Issues

In this section, we discuss some implementation issues with respect to the negotiation mechanism discussed in this paper.

An offer by a server will not always correctly describe the QoS that would be received by a client and vice versa. As an example, consider a server that offers a delay of 30 ms. In a distributed system we need to take into account the time between when the client issues the request and it is received by the server. If this takes 20 ms and the return of the result takes 15 ms, the client should expect a total round-trip delay of 65 ms. The invocation and the return may take different amounts of time due to, for example, differences in the complexity of transferred data. We have a similar situation between what is specified by the client and what is actually required by and provided to the server,

To take these differences into account an implementation must allow transports (see section 1.2) to annotate offers and requirements appropriately.

We use *filter* as a generic term for any mechanism that alters the QoS specified by the application in a transport and agent implementation specific way. We also see the need for a mechanism that allows the negotiation mechanism to notify the filters of the offers in any negotiation and to bind the appropriate filters to the QoS captured in the deal. Each filter may modify an offer to produce multiple offers corresponding to the results of filtering. Each filter may also further modify information put in by other filters. Note that the filters need to deal with inter-dependencies between the contracts in the profiles.

A possible implementation scheme for the filters is to arrange them in a pipeline and pass the offers through them. The pipelining must satisfy interdependencies between filter, which might not be trivial.

Another implementation issue is the representation of QoS profiles and offers. This representation should be compact and allow additional data to be added to each offer by the filters. Note that the two negotiating agents may not support or understand all of the contracts, dimensions or additional information present in the offers that they receive from one another. The negotiation mechanism should be able to handle such a situation gracefully.

Other issues include persistent storage for offers that become deals so that they may be referred to when calls are made associated with those deals or when a dispute occurs between the negotiating agents, handling expired or broken deals, etc., that we collectively term deal management. Deal management is a subject for future research.

8. Concluding Remarks

QoS negotiation is essential if we are to build systems that can dynamically (re-)configure to changing QoS conditions and requirements. Such adaptations will enable systems to be deployed under many different QoS conditions, and still provide services with acceptable QoS. QoS negotiation allows systems to trade different QoS characteristics off against each other when resources are scarce or when there are partial failures. To enable such adaptation, we must consider negotiation over multiple categories, as well as negotiation based on worth, rather than absolute goals.

To implement a negotiation mechanism, we need to understand how such a mechanism fits into a communications infrastructure, how negotiation is performed, and how the worth of QoS specifications can be computed. This paper has presented a model and architecture for such a mechanism. The protocol has been specified formally and simulated; we are currently in the process of investigating implementation techniques and prototypes for the full as well as restricted versions of the proposed model.

Acknowledgments

References

1. Ingo Barth, Gabriel Dermler, and Walter Fiederer. Levels of Quality of Service in CINEMA. University of Stuttgart, Report.

2. Christian R. Becker and Kurt Gheis. MAQS—Management for Adaptive QoS-enabled Services. Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, December, 1997.

3. Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Unified Modeling Language. *Rational Software Corporation*, version 1.0, January 1997.

4. Tsu-Wei Chen, Paul Kryzankowski, Michael R. Lyu, Corman Sreenan, and John A. Trotter. Renegotiable Quality of Service—A New Scheme for Fault-Tolerance in Wireless Networks.

5. Gabriel Dermler, Walter Fiederer, Ingo Barth, and Kurt Rothermel. A Negotiation and Resource Reservation Protocol (NRP) for Configurable Multi-edia Applications. Proceedings of Multimedia'96, IEEE, 1996.

6. Svend Frølund and Jari Koistinen, QML: A Language for Quality of Service Specification. Hewlett-Packard Laboratories, Tech. Report, HPL-98-10, January 1998.

7. Svend Frølund and Jari Koistinen, Quality of Service Specification in Distributed Object Systems Design. In Proc. 4th USENIX Conference on Object-Oriented technologies and Systems. April, 1998.

8. Richard Grimes. Professional DCOM Programming. WROX Press Ltd, 1997.

9. A Hafid, G. V. Bochmann, and R. Dassouli. Quality of Service Negotiation with Present and Future Reservations. Computer Networks and ISDN Systems, 1996.

10. Constant Gbaguidi, Simon Znaty, Jean-Pierre Hubaux. A Generic Service Management Architecture for Multimedia Multipoint Communications. Proceedings of the IFIP TC6 Conference on Intelligent Networks and New Technologies, 1995.

11. Andrew Campbell, Geoff Coulson, David Hutchison. A Quality of Service Architecture. Computer Communication Review, ACM SIGCOMM, April, 1994.

12. David Harel. STATECHARTS: A Visual Formalism for Complex Systems. Science of Computer Programming, Vol 8., 1987. Elsevier Science Publications B. V.

13. C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

14. Gerard J. Holzmann. Design and Validation of Computer Protocols. Prentice-Hall, 1991.

15. Jarno Rajahalme, Telma Mota, Frank Steegmans, Per F. Hansen, and Fernanda Fonseca. QoS negotiation in TINA. Proceedings of TINA'97 Conference. November, 1997.

16. Jeffrey S. Rosenschein and Gilad Zlotkin. Rule of Encounter: Design conventions for automated negotiation among computers. The MIT Press, 1994.

17. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions om Computer Systems*, Vol. 2, No. 4, November 1984.

18. Object Management Group. *The Common Object Request Broker: architecture and specification*, July 1995. revision 2.0.

19. K. Paul Yoon and Ching-Lai Hwang. Multiple Attribute Decision Making: An Introduction. Sage Publications Inc. 1995.