Quality of Service Aware Distributed Object Systems

Svend Frølund
Hewlett-Packard Laboratories, frolund@hpl.hp.com

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

Keywords: QoS-specification, QoS-enabled Trading, Distributed Object Systems, Object Component Specification, Object Interoperability

Computing systems deliver their functionality at a certain level of performance, reliability, and security. We refer to such non-functional aspects as quality-of-service (QoS) aspects. Delivering a satisfactory level of QoS is very challenging for systems that operate in open, resource varying environments such as the Internet or corporate intranets. A system that operates in an open environment may rely on services that are deployed under the control of a different organization, and it cannot per se make assumptions about the QoS delivered by such services. Furthermore, since resources vary, a system cannot be built to operate with a fixed level of available resources. To deliver satisfactory QoS in the context of external services and varying resources, a system must be QoS aware so that it can communicate its QoS expectations to those external services, monitor actual QoS based on currently available resources, and adapt to changes in available resources.

A QoS-aware system knows which level of QoS it needs from other services and which level of QoS it can provide. To build QoS-aware systems, we need a way to express QoS requirements and properties, and we need a way to communicate such expressions. In a realistic system, such expressions can become rather complex. For example, they typically contain constraints over user-defined domains where constraint satisfaction is determined relative to a user-defined ordering on the domain elements. To cope with this complexity we are developing a specification language and accompanying runtime representation for QoS expressions. This paper introduces our language but focuses on the runtime representation of QoS expressions. We show how to dynamically create new expressions at runtime and how to use comparison of expressions as a foundation for building higher-level QoS components such as QoS-based traders.

1. Introduction

1.1 Future of Distributed Enterprise Systems

Enterprises increasingly rely on distributed computer systems for business-critical functions. They often use such systems for internal information sharing, handling of business tasks such as orders and invoices, and accounting. In addition, businesses increasingly rely on distributed systems for their interactions with other business such as partners, customers, and sub contractors.

Since distributed systems are business critical, they must not only provide the right functionality, they must also provide the right quality-of-service (QoS) characteristics. By QoS, we refer to non-functional properties such as performance, reliability, quality of data, timing, and security. For some applications, best-effort QoS is acceptable; while others require predictable or guaranteed levels of QoS to function properly. In real-time systems, for example, timing is essential for correctness. In banking systems, security is necessary and must not be compromised. Business-critical enterprise systems and telecommunications systems must be highly available.

Ideally we like all systems to be up 100% of the time, be fully secure and deliver exceptional performance. Unfortunately, building such systems is not realistic. In practice, we need to make trade-offs between QoS and cost of development and between different QoS categories. For example, achieving very high reliability is not only technically difficult, it is also very costly. Furthermore, providing very high reliability will also impose a performance overhead. QoS requirements, such as reliability, cannot be considered in isolation, they must be considered in the context of development cost and other QoS requirements, such as performance and security.

It is also common that a technology for satisfying one QoS aspect will not be compatible with a technology for satisfying another QoS aspect. As an example, it might be difficult to combine a group communication mechanism for high availability with certain security mechanisms.

To find the right solution for an enterprise computing system, we need to understand the cost of not satisfying certain QoS requirements and the cost of implementing and using mechanisms that provide a specific QoS level. To complicate things further, the cost of unsatisfactory QoS characteristics may vary according to the time of day, day of the week, and week of the year. It is also the case that the relative importance of specific QoS characteristics may vary over time. During day time availability might be more important than performance due to online sales transaction processing. During the night, accounting functions might need maximum performance to finish within a certain time.

In summary, we believe that enterprises will become increasingly dependent on distributed systems both for internal business automation and for the interaction with other enterprises and end customers. This will not only require the right functionality to be provided but also that the systems provide adequate quality-of-service. There are many issues in building systems with adequate QoS, and we believe that QoS must be considered systematically throughout the life-cycle of distributed enterprise systems. The goal is to support QoS-enabled systems, and we present a QoS fabric that is an essential building block for such systems.

1.2 QoS-Enabled Systems

We refer to systems that understand the notion of QoS and can provide defined levels of QoS as *QoS-enabled* systems. We believe that to build future enterprise systems we will need to make them QoS enabled. QoS enabling involves many different components such as:

- **Mechanisms :** The ability to provide distinct levels of QoS through combinations of mechanisms such as different reliability and security solutions.
- **Adaptability :** The ability to select services based on the QoS that they provide. Examples of this is QoS-based trading and QoS negotiation. Adaptability also involves the ability to provide graceful degradation.
- **Monitoring :** The ability to monitor the QoS that is provided and received and to check compliance with existing contracts.
- **QoS Awareness :** The ability to describe what QoS can be provided by a specific service and what QoS that it requires.
- **Meta Data :** Information about current load, number of deals, and available resources.

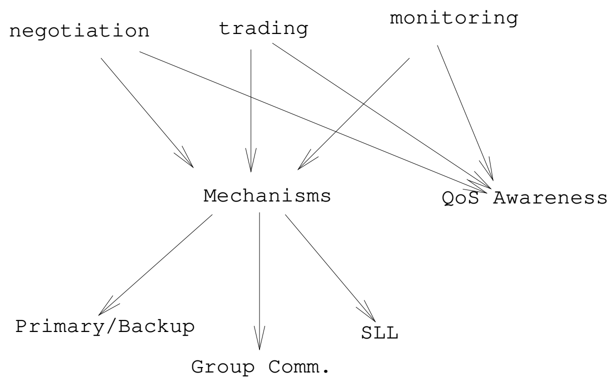

FIG. 1. Dependencies for QoS Enabling

Today, most systems are not QoS enabled. Rather they provide ad hoc or best-effort QoS. Sometimes special security, reliability, or other mechanisms are used, but applications are still unaware of the QoS that is required and provided. Figure 1 illustrates the dependencies of the different component of QoS enabling. First one needs a variety of mechanisms—such as reliability and encryption protocols—that enable a system to satisfy QoS requirements. To be able to describe, negotiate and monitor QoS, the system needs a special protocol and description technique that makes them QoS aware. This paper focuses on such a language and a runtime representation for making distributed systems QoS aware.

1.3 QoS-Aware Systems

For a distributed system to deliver predictable QoS, it must be QoS aware. A system typically has QoS obligations towards its users and to fulfill those obligations, it needs to receive predictable QoS from other systems. To deliver predictable QoS, a system must know its obligations and it must know which requirements to impose on other systems in order to meet those obligations. Systems must establish QoS agreements with each other and be able to monitor compliance of those agreements. Establishing QoS agreements require that systems know, and can can exchange, their respective requirements and properties.

For example, consider a distributed currency trading system. The front-end component of the system presents a user interface to human currency traders. The front-end uses a rate service to get rate updates and a currency trading service to perform currency trades. In order for the front-end to deliver predictable QoS to its human users, it must receive predictable QoS from the rate service and currency trading service. For example, the front-end may expect the rate service to be up 99 % of the time, deliver rate updates every minute, and provide information for a specific set of currencies. If the front-end is designed to work with a particular rate service, these expectations can be incorporated into the overall system design. However, if the front-end connects to a rate service on the Internet, it needs to be explicit about these expectations and establish a QoS agreement with a rate service based on these expectations.

Figure 2 illustrates the structure of this simple system and how QoS information need to flow dynamically in the system.

To facilitate the establishment of QoS agreements in distributed object systems, we need a way to specify the QoS requirements of clients (such as the front-end of the currency trading system) and the QoS properties of services (such as the rate service). We also need a way to communicate these specifications and compare them to determine if a particular service meets the requirements of a particular client.

The use of QoS specifications in distributed systems requires adequate tool support. Constructing and managing QoS specifications in an ad hoc manner is complicated because of the expressive power required and because we need to compare specifications to determine if one satisfies another. In terms of expressive power, QoS specifications essentially consist of constraints. However, the structure of these constraints is fairly complex. We need constraints over user-defined domains with a user-defined ordering, and we need constraints over statistical properties, such as mean, variance, and percentiles. Moreover, we need to bind these constraints to fine-grained entities, such as operation arguments, and to coarse-grained entities, such as interfaces. The required expressive power also makes it hard to compare specifications in an ad-hoc manner. For example, the comparison algorithm must operate on user-defined domains and take user-defined orderings into account. Rather than use the data structures of conventional programming languages, we have a developed a special-purpose specification language, called QML. To make it practical, we have integrated QML with existing technologies for distributed systems, such as interface definition languages, and design languages (UML).

This paper describes a QML-based QoS fabric that allows us to systematically use QoS specifications at runtime. Our QoS fabric is called QRR (QoS Runtime Representation). It enables higher-level QoS functions, such as establishment of QoS agreements.

For the purpose of this paper we will use QML to introduce the fundamental concepts behind QRR, because QML supports these concepts in the most direct manner. We then show how these concepts can be mapped to QRR as a combination of C++ classes and CORBA IDL type definitions. Because the basic types of QRR are specified in IDL, we can communicate QRR objects in a CORBA infrastructure. QRR is not inherently tied to C++ or CORBA IDL, we could also implement QRR in JAVA and other languages, and DCOM and other distributed object infrastructures.

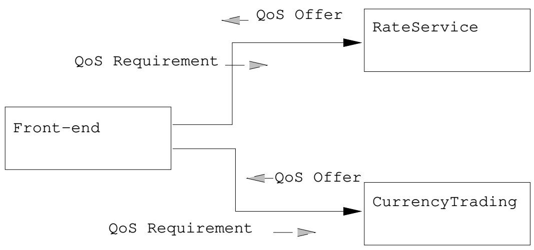

FIG. 2. Structure of the currency trading system

In pursuing QoS mechanisms for distributed object systems we have come to appreciate a specification and representation format for QoS highly. Few QoS mechanisms can be built without one. In addition, open systems will require a common specification technique and format analogously to the IDL and IIOP standards. We believe QML and QRR could be a first step towards such a standard and enable more widely deployed QoS mechanism in future distributed object systems.

The paper is organized as follows. Section 2 introduces QML and its underlying concepts for QoS specification. We then describe QRR in Section 3.1. We outline how we represent the QML concepts in terms of C++ and CORBA IDL, and we show the architecture of the QRR QoS fabric. We illustrate how to use QRR to implement distributed object systems with predictable QoS in Section 4. Finally, we give a brief overview of related work in Section 5, and we conclude in Section 6.

2. QML: A Language for QoS Specification

QML is a general-purpose QoS specification language; it is not tied to any particular domain, such as real-time or multi-media systems, or to any particular QoS category, such as reliability or performance. QML captures the fundamental concepts involved in the specification of QoS properties. Here, we give a brief introduction to these fundamental concepts. For a complete QML language definition, including formal syntax and semantics, consult [5].

QML has three main abstraction mechanisms for QoS specification: *contract type*, *contract*, and *profile*. A contract type represents a specific QoS category, such as performance or reliability. Contract types a user-defined abstractions, there are no built-in contract types in QML. A contract type defines the *dimensions* that can be used to characterize a particular QoS category. A dimension has a domain of values that may be ordered. There are three kinds of domains: *set* domains, *enumerated* domains, and *numeric* domains. A contract is an instance of a contract type and represents a particular QoS specification. Finally, QML profiles associate contracts with *interface entities*, such as operations, operation arguments, and operation results.

We use the currency trading example from the introduction to illustrate the QML specification mechanisms. We show how to specify QoS properties for a rate service object. Figure 3 gives a CORBA IDL [14] interface definition for a rate service object. It provides an operation, called `latest`, for retrieving the latest exchange rates with respect to two currencies. It also provides an operation, called `analysis`, that returns a forecast for a specified currency. The interface definition specifies

```
interface RateServiceI {
  Rates latest(in Currency c1,in Currency c2)
    raises(InvalidC);
  Forecast analysis(in Currency c) raises(Failed);
};
```

FIG. 3. The `RateServiceI` interface described in CORBA IDL

```
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 < 10 no / year;
  TTR {
        percentile 100 < 2000;
        mean < 500;
        variance < 0.3
      };
  availability > 0.8;
};

rateServerProfile for RateServiceI = profile {
  require systemReliability;
  from latest require Performance contract {
    delay {
        percentile 80 < 20 msec;
        percentile 100 < 40 msec;
        mean < 15 msec
        };
  };

  from analysis require Performance contract {
    delay < 4000 msec
  };
};
```

FIG. 4. Contracts and Profile for RateServiceI

the syntactic signature for a service but does not specify any semantics or non-functional aspects. Using QML, we can specify the QoS properties for this interface.

The QML definitions in Figure 4 include two contract types `Reliability` and `Performance`. The `Reliability` contract type defines three numeric dimensions. The first dimension (`numberOfFailures`) represents the number of failures per year. The keyword "decreasing" indicates that a smaller number of failures is better than a larger one. Time-to-repair (`TTR`) represents the time it takes to repair a service that has failed. Again, smaller values are better than larger ones. Finally, the dimension called `availability` represents the probability that a service is available. For the `availability` dimension, larger values are better than smaller values.

In Figure 4 we also define a contract called `systemReliabilty` of type `Reliability`. The contract specifies constraints over the dimensions defined in the `Reliability` contract type. The first constraint specifies an upper bound for the number of failures. The second constraint applies to the `TTR` dimension. This constraint uses statistical properties, such as mean, variance, and percentiles, to characterize QoS along the `TTR` dimension. In QML, we refer to such statistical properties as dimension *aspects*. The aspect "**percentile** `100 < 2000`" states that the 100th percentile must be less than 2000.

Next, we introduce a profile called `rateServerProfile` that associates contracts with entities in the `rateServiceI` interface. The first requirement clause in the profile states that the service should satisfy the previously defined `systemReliability` contract. Since the clause does not refer to any particular operation, it is considered a default requirement that applies to every operation within the `rateServiceI` interface. Being part of a default requirement, the `systemReliability` contract is called a *default contract* for the profile. Contracts for individual operations are allowed only to strengthen (refine) the default contract. In the `rateServerProfile` there is no default performance contract; instead we associate individual performance contracts with the two operations of the `RateServiceI` interface. For `latest` we specify in detail the distribution of delays in percentiles, as well as a upper bound on the mean delay. For `analysis` we specify only an upper bound and can therefore use a slightly simpler syntactic construction for the expression. Since throughput is omitted for both operations, there are no requirements or guarantees with respect to this dimension.

We have now specified example reliability and performance requirements for the `rateServiceI` interface. Although the `rateServerProfile` is specified in terms of an interface (`rateServiceI`), it characterizes the QoS of a particular implementation of this interface. We can specify multiple profiles for the same interface, and use distinct profiles for different implementations. The key to this flexibility is that QoS specifications are not embedded within an interface, but defined as separate entities.

Intuitively we would say that the constraint "`delay` < 10" is stronger than the constraint "`delay` $<$ 20." This relationship between the two constraints is due to the fact that `delay` is a decreasing dimension (smaller values are better) and the fact that the value 10 is smaller than the value 20. In QML, we formalize this notion of "stronger than" for constraints and define a general conformance relation over constraints. Stronger constraints conform to weaker constraints. We then use this conformance relation on constraints to define conformance relations on contracts and profiles. Conformance is an important aspect of QoS specifications because it enables us to compare specifications based on constraint satisfaction rather than exact match. As we show in Section 4, conformance is essential for implementing a QoS-based trader: the QoS-based trader should select any service whose QoS properties conform to the client's requirements, the trader should not just select the services whose properties are identical to the client's requirements.

QoS specifications can be used in many different situations. They can be used during the design of a system to understand and document the QoS requirements that must be imposed on individual components to enable the system as a whole to meet its QoS goals. In [6] we show how to use QML at design time. The focus of this paper is the use of QoS specifications as first-class entities at runtime.

3. Requirements for a QoS Fabric

Our QoS fabric, QRR, is based on the following requirements:

1. QRR should support the same fundamental concepts as QML. We want to use the same QoS specification concepts during design and implementation. Using the same concepts implies that QML's precise, formal definition [5] carries over to QRR. A precise definition improves the interoperability of different QRR/QML components.
2. Since some QoS requirements may not be known until runtime, it should be possible to dynamically create new QRR specifications. Rather than use dynamic compilation for such specifications, we want to call generic creation functions in the QRR library. Dynamic compilation implies that the QML compiler must be available at all nodes in a distributed system. Moreover, dynamic compilation is likely to impose a significant performance penalty.
3. It should be possible to explicitly check consistency of dynamically created specifications against the static semantic rules of QML/QRR. The QML compiler checks the rules for compiled specifications. We need a library function that checks the rules for dynamically created specifications.
4. Once created, there should be ways to manipulate QRR specifications. For example, a QoS offering by a server may have to be adjusted relative to the current execution environment to accurately reflect what QoS the client will actually receive.
5. QRR should impose a minimal overhead and be scalable.
6. QRR should provide a minimal set of generic building blocks for runtime QoS specification. In particular, QRR specifications should be independent of the mechanisms and applications that use them. For example, in negotiation, as well as trading, we are interested in agreements between parties involving commitments from both sides. Thus we are dealing with structures consisting of pairs of QoS specifications (one for each party). Rather than provide an agreement abstraction in QRR, we only provide the basic building blocks that represent QoS specifications. It is then up to the mechanisms and applications to use these basic building blocks to create composite structures.

We are implementing QRR to satisfy these requirements. Currently, we have implemented a prototype QML compiler and a prototype QRR library. We have successfully compiled QML specifications into QRR, instantiated those specifications in a CORBA environment, communicated the specifications between distributed components, and compared them using a conformance checking function that is part of the QRR library.

3.1 QRR: A QML-Based QoS Fabric

The QRR implementation contains a generic C++ library that allows applications to create QRR specifications and to check conformance of these specifications. This library is linked into applications that use QRR. The library defines a number of data types that are used to represent QRR specifications in C++. These data types are generated from CORBA IDL type definitions to facilitate the communication of QRR specifications between distributed CORBA objects.

In addition to the generic library, the implementation also contains a QML to QRR compiler. This compiler emits a mix of IDL and C++ code to represent a particular QML specification. The emitted IDL code consists of types that represent that QML specification. The C++ code contains functions to create QRR instances of the QML specification. The emitted IDL code is translated into C++ using a conventional IDL compiler.

3.2 Representation

We describe the representation of QML constructs in terms of CORBA IDL and C++. Profiles are represented as instances of the `profile` struct shown in Figure 5. They contain the profile name, the interface name, a sequence of default contracts (`dcontracts`), and a sequence (`profs`) of structs, each associating an interface entity with a set of contracts. The `profs` sequence represents the individual contracts of the profile. In QRR, all profiles are instances of the `profile` struct. For a particular profile specified in QML, the QML compiler emits a C++ function that constructs an instance of the `profile` struct. The C++ function also constructs and assigns appropriate data structures to the fields of this struct.

```
struct profile {
   string pname;
   string iname;
   contractSeq dcontracts;
   entityProfileSeq profs;
};
```

FIG. 5. IDL for profile

QoS constraints are represented as instances of the struct `constraint` in Figure 7. A `constraint` struct has a sequence of `aspect` structs, as well as a tag indicating whether it is a simple constraint—such as "*delay* < 10'—or a set of aspects representing statistical characterizations. We define a separate struct type for each aspect kind, however, the figure only shows the struct used to represent mean aspects. Because IDL does not allow polymorphism for structs, we cannot directly reflect the relationship between the general notion of aspects, captured by the `aspect` struct, and particular aspect types such as `mean`. Instead of defining particular aspect types as subtypes of `aspect`, we define an *any* field in instances of `aspect` that contains a particular aspect instance. We also define a type tag in instances of `aspect` that indicates which particular aspect type has been wrapped in the *any* field.

We provide two alternative representations for contracts and contract types. In the *generic* representation, all contracts are instances of the same type, and this type is then part of the QRR library. In the *static* representation, only contracts of the same QML contract type are instances of the same QRR type. In addition, the QRR types used for the static representation are emitted by the QML compiler.

The static representation requires that the emitted QRR types are linked into the application that instantiates them. On the other hand, the static representation facilitates a more efficient implementation of conformance checking and other QRR functions.

With the generic representation, applications can dynamically create and communicate contracts whose types are not known at compile time. However, manipulation and analysis of contracts is less efficient in the generic mapping because the structure of contracts must be discovered dynamically.

Although we describe them as separate representations, our goal is to allow their simultaneous use to achieve maximum flexibility and performance. Since our current implementation only supports the static representation, we only give a brief overview of the generic representation and concentrate primarily on the static representation.

The IDL definitions in Figure 8 describe some elements of the generic contract representation. In the generic representation, all contracts are instances of the struct called `contract`. A contract's dimensions are then represented as a sequence of structs of type `dimension`. A contract has a type identifier (`tid`) that refers to its contract type. Contract types are built from various generic structs that capture all the information about domains and their ordering. These type representations are quite elaborate as they contain information about all values, how these values are ordered, and whether the dimension is increasing or decreasing. Due to space constraints we do not describe these type structures in detail in this paper.

With the static representation, the QRR compiler will map a QML contract type into a number of C++ classes and an IDL struct. The C++ classes repre-

```
enum aspectKind {
   ak_freq, ak_perc,
   ak_mean, ak_var,
   ak_simple
};

struct mean {
   operators op;
   value num;
};

struct aspect {
   aspectKind ak;
   any asp;
};

typedef sequence <aspect> aspects;

enum constrKind { ck_simple, ck_stat };

struct constraint {
   constrKind ck;
   aspects asps;
};
```

FIG. 7. IDL for aspect and constraint

```
struct dimension {
   string name;
   constraint constr;
};

struct contract {
   tid ct;
   sequence<dimension> dims;
};
```

FIG. 8. IDL for generic contracts

sent the contract type itself. They contain information about the domain elements and the domain ordering for the dimensions defined in the contract type. The IDL struct is used to represent contracts that are instances of the contract type. An instance of the emitted IDL struct represents a particular contract.

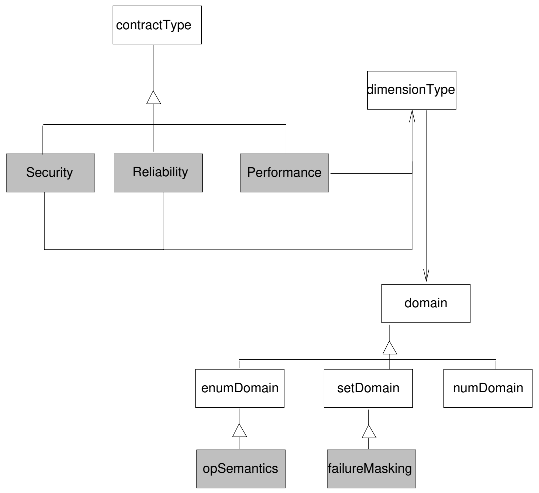

FIG. 6. Class diagram for contract type representation.

The emitted C++ classes inherit from, and adds to, a set of contract type base classes implemented in the QRR library. Figure 6 shows—using UML [3] notation—a simplified view of the C++ classes for contract types in the static representation. Emitted classes are grayed and classes defined in the QRR library are white.

Sub-classes of `contractType` represent the emitted classes for specific contract types. These classes contain data members that represent the dimensions in the contract type. They also contain a conformance checking function. Since the conformance checking function is emitted on a per-contract type basis, can directly refer to the type's dimensions as data members.

If the contract types contain set or enumeration domains that are ordered, we also emit C++ classes that represent these domains. The domain classes are subclasses of the library class called `domain`. The main role of emitted domain classes is to provide information about the domain ordering.

In addition to the C++ classes, the compiler also emits an IDL struct definition for each contract type. The name of this struct is the contract type name with *_i* appended to it. The struct has one field for each dimension. Each field has the same name as the corresponding dimension and is of type `constraint`.

In Figure 9 we show a QML contract type called `Reliability` and the corresponding emitted IDL struct.

An instance of the `Reliability_i` struct will hold instances of constraints that in turn hold the aspects specified for each individual constraint. An instance also contains the type identifier of its contract type. Currently, we represent type identifiers as text strings. Notice that QML has a flat name space for contract types.

The programmer can manually instantiate instances of contract structs, such as `Reliability_i`. Manual instantiation is tedious because the programmer must explicitly create structs that represent the constraints of the contract struct. To automate the instantiation process, the QML compiler emits instantiation functions for each contract and profile declared in QML. Without going into much detail, but still give a concrete sense of how these instantiation functions are constructed, we provide a simple contract in Figure 11 and the corresponding emitted construction function in Figure 7.

Running these simple definitions through the QML to QRR compiler will produce the static representation which is a struct with name `T_i`. The compiler also emits the C++ class `T`, which describes the contract type and implements conformance checking. In addition, it produces a function—shown in Figure 12—with the same name as the specified contract (in this case `C`). When `C` is called it will return an instance of `T_i` representing the constraints specified in `C`. The `C` function uses the same lower level functions as are provided to applications that manually composes contracts and profiles. Similarly, we produce functions for profiles that build up the corresponding QRR structures.

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

struct Reliability_i {
   tid ct;
   numberOfFailures constraint;
   TTR constraint;
   availability constraint;
};
```

FIG. 9. IDL for statically generated contracts

```
type T = contract {
   l : increasing numeric msec;
   s : enum {initial, amnesia,
                    noguarantee, rolledback};
};

C = T contract {
   l {
     percentile 40 < 50 ;
     mean < 20
    };
   s == amnesia;
};
```

FIG. 11. A simple contract type and contract

3.3 Library Functions

When an application needs to check conformance, it invokes the library function `conformsTo` whose signature is shown in Figure 13. This function takes two profiles, and checks conformance between their contracts. Inside profiles, contracts are stored as a pair consisting of a contract type name and an element of type *any*. For a performance contract, the *any* element will contain an instance of type `Performance_i` and the contract type name will be the string "Performance". To check conformance between performance contracts, the `conformsTo` will use the string "Performance" to lookup the C++ object which represents performance contract types at runtime. This object is of type `Performance` and will have a virtual function called `conformsTo` (the signature of this function is given in Figure 13 as `Performance::conformsTo`). The `Performance::conformsTo` function is emitted. It expects two *any* arguments that both contain instances of the struct `Performance_i`. Since it is emitted, the `Performance::conformsTo` function knows which objects to extract from the *any* arguments.

```
T_i * C(){
   T_i * _C;
   _C = new T_i;
   _C->ct =  CORBA::string_dup("T");
   //Create aspects for l
   aspect * _l;
   _C->l.asps.length(2);
   _C->l.ck = ck_stat;
   _l = qml_perc_asp(le,40,(float)50);
   _C->l.asps[0] = *_l;
   delete _l;

   _l = qml_mean_asp(le,(float)20);
   _C->l.asps[1] = *_l;
   delete _l;

   //Create simple for s;
   aspect * _s;
   _C->s.asps.length(1);
   _C->s.ck = ck_simple;
   _s = qml_simp_constr(eq,(float)2/*amnesia*/);
   _C->s.asps[0] = *_s;
   delete _s;
   return _C;
};
```

FIG. 12. Emitted function that creates a `T` contract

Figure 13 also shows the signatures of some of the functions provided by the library to dynamically create and check QRR specifications. As an example, the `checkSem` function will check the static semantics of a profile instance.

To dynamically create profiles, we need to create contracts and aspects and build the appropriate runtime structure. The construction of such runtime structures can involve many operations and be error prone. We therefore intend to provide more convenient programming abstractions in the future. The current library implementation only provides convenience functions for individual aspects. The functions `qml_perc_asp` and `qml_freq_asp` are examples of such functions for percentile and frequency aspects respectively.

3.4 Programming Model

To give the reader a better feel for the programming model offered by QRR, we describe a simple QoS compatibility-checking mechanism that allows a client to send its QoS requirements in the form of a QRR profile to a server. The server checks whether it can satisfy the client's requirements.

```
int conformsTo(profile &stronger,profile &weaker);

int Performance::conformsTo(CORBA::Any *stronger,
                            CORBA::Any *weaker);

int checkSem(profile &p);

template <class elemType>
aspect * qml_perc_asp(operators op,
                      int pe ,
                      elemType val);

template <class elemType>
aspect * qml_freq_asp(operators op,
                      int fr,
                      rangeType l,
      elemType low,
                      elemType high,
                      rangeType r);
```

FIG. 13. Some library function signatures

```
interface QoSAware {
   exception invalidProfile{};
   boolean compatible(in profile p)
       raises (invalidProfile);
};
```

FIG. 10. QoSAware interface

To support the QoS-checking mechanism, the server implements the interface `QoSAware`, which we describe in Figure 10. The operation `compatible` allows the client to send the profile it requires to the server. The server responds with true if the client's requirements and the server's capabilities are compatible; and with false otherwise. If the profile is semantically invalid, the operation raises an exception.

To make the QoS checking more concrete, let us assume that a server A provides an interface I_1 and uses a server B that implements an interface I_2. We can describe, in QML, the requirements of server A on server B as a profile for the interface I_2. We can also describe the QoS provided by A as a profile for interface I_1. Having defined those profiles and the contracts that they use we can emit QRR code that can be compiled and linked with both servers. Notice that server A plays the role of client relative to server B.

```
CORBA::Environment env;
profile * p = i2_prof();
if (I2ref->compatible(*p,env) {
   //OK to use this server
....
} else {
   //use another server
....
};
```

FIG. 14. Client call

```
CORBA::Boolean B_serverImpl::compatible(
                    const profile & p,
                    CORBA::Environment & _ev)
{
  profile &p1 = (profile &)p;

   if (! checkSem(p1) {
      throw QoSAware::invalidProfile();
   };

   if(conformsTo(myprof(),p1)){
      cout << "Conformance... " << endl;
      return 1;
   }
   else {
      cout << "Non-conformance..." << endl;
      return 0;
   }
};
```

FIG. 15. Server implementation

We can create the specified profiles in server A by invoking the emitted functions that have the same names as the profiles specified in QML. If we have a profile named `i2_prof` specifying A's requirements on I_2, A objects would use an emitted function called `i2_prof` to create an QRR instance of this profile. The C++ code in Figure 14 illustrates how a profile can be created and sent with an ordinary CORBA request.

The implementation of `compatible` simply takes the profile specified for the server and checks its conformance to the profile supplied by the client. We assume that the server obtains its own profile by invoking a function called `myprof`. The implementation of `compatible` checks the static semantics of the profile before doing performance checking. In the future we intend to include information in a profile that allows a program to determine whether a profile has already been checked for semantic validity or not. With this extra information, we can avoid redundant semantic checks. Figure 15 describes a simple implementation of a server that supports the `QoSAware` interface.

3.5 Discussion

In the generic mapping, contracts can be created, embedded in profiles and communicated even if they are not statically known. We could require that it is decided up front whether the generic or static mapping will be used during a session. If we use the generic mapping we would initially send contract type descriptions for all contract types to be used during the session. This would ensure that each communicating peer has all contract type descriptions available. Using the static mapping, on the other hand, we require that all contract types are known statically and compiled into the participating objects. If a received contract is an instance of an unknown type, the receiver can do little more but raise an exception. Having a strict separation of the generic and static mapping, and only use one of them during a particular session, would simplify the implementation, but it would also be quite inflexible. In the case of the generic mapping, it also imposes unnecessary performance overhead when contract types are already known.

We envision a protocol for exchange of QoS specifications where the communicating parties assume that contract types are known by all parties but where each participant is also prepared to transmit type information on demand to other parties. This more flexible scheme can be used for the generic mapping. It will also be useful in the future when we allow a combination of generic and static mapping. The problem of locating type information at runtime is similar to the problem of locating code for migrating objects in Emerald [9].

Global consistency of types is another issued that we are considering. It is necessary to have a mechanism for identifying and comparing types to determine if they are the same. The current implementation uses an oversimplifying approach based on text strings. Our plans for future work includes leverage from previous solutions such as the handling of types in CORBA to resolve this issue in QRR.

4. Example: A Qos-Based Trader

This section illustrates how QRR can be used to construct higher-level QoS components. We show how to build a QoS-based trader, and explain its utility in the context of the currency trading example from the introduction section (do not confuse a QoS-based trader with the currency trading service in the currency trading system).

The purpose of this section is not to discuss or advocate the merits of QoS-based trading, but to show concretely the expressive power and flexibility of QRR. The constructs of QRR make it relatively straightforward to implement QoS-aware components that would otherwise be complicated to build. One of the main advantages of using QRR is the precisely defined notion of specification conformance and a library function to check conformance at runtime.

4.1 QoS-Based Trading

A conventional trader [15] in distributed systems facilitates the binding between clients and services. Services register with the trader, and clients query the trader to find a service that satisfies certain criteria. We show an example interface to a very simple conventional trader in Figure 16.

Services register themselves by calling the `offer` method on the trader. A service passes a description of its properties, the properties that can be used for service selection, and a reference to itself. The trader returns an offer identifier to the service. The service can use this identifier to later withdraw its offer by invoking the `withdraw` method. Clients call the methods `find` and `findAll` to obtain obtain service references. The `find` method returns a single service reference. The found service satisfies the criteria passed in as parameter to the `find` call. The `findAll` method returns a list of service references. It returns all the services that match the criteria passed in as parameter.

In general, the trader matches criteria passed in by clients against properties passed in by services. A very simple trader could represent properties and criteria as text strings, in which case the trader becomes a simple name service. Conventional trader services typically use name-value pairs for server selection. The server passes in one such attribute list, and clients pass in an attribute list that is matched against server attribute lists. Typically, matching is based on name-value equality, with the provision that the client's attribute list must be equal to a sub-list of the server's attribute list.

The main idea behind QoS-based trading is to use QoS specifications as service properties and client criteria, and to use specification conformance to perform the criteria-to-properties matching. We show that with QRR it is relatively straightforward to implement a QoS-based trader. An alternative implementation strategy would be to use a conventional trader and represent QoS specifications as name-value pairs. However, we would then be limited by the expressive power of name value pairs; it is not clear how to elegantly represent the concepts of QML and QRR in terms of name-value pairs. More seriously, with a conventional trader we would use equality for server selection. It is essential that we can select a service even if the client's requirements are not equal to the service's properties: we want to select a service as long as the service's properties satisfies, or *conforms to*, the client's requirements. Notice that even if we decided to represent QoS specifications as name-value pairs, we would still need to replace the matching procedure in the trader to perform matching based on conformance rather than equality. It would be complicated to implement this modified matching function because we cannot define a general conformance relation on name-value pairs. We would have to restrict the conformance checking to only apply to name-value pairs that in fact represent QoS specifications.

We want to use the QoS-based trader to establish QoS agreements between clients and services in distributed object systems. A QoS agreement is a contract between a client and a service. In our discussion so far, we have talked about client requirements and service properties. This is a somewhat simplified picture. The service properties may depend on the way in which the client uses the service. For example, the throughput that the service can provide may depend on how frequently the client calls the service. So, for performance, a QoS contract may involve requirements and properties for both clients and services.

```
interface Trader {
  OfferId offer(in ServiceProperties sp,
    in Object obj) raises (invalidOffer);
  Match find(in Criteria cr) raises(noMatch);
  MatchSeq findAll(in Criteria cr) raises(noMatch);
  void withdraw(in OfferId o) raises(noMatch);
};
```

FIG. 16. The interface of a simple conventional trader

Thus in general, the `ServiceProperties` argument to the `offer` method in a QoS-based trader will involve service properties and requirements. The service can provide the properties if the client satisfies the requirements. Figure 17 gives a possible structure for `ServiceProperties` using QRR. We describe the service requirements and properties using profiles. Figure 17 also shows the structure of client criteria. These also have a two profiles: one representing client requirements and one representing client properties.

With the `conformsTo` function on profiles introduced in Section 3.1, we can now implement the matching procedure in the QoS-based trader. We illustrate the conformance-based matching procedure in Figure 18. The procedure iterates through the registered services and for each service checks whether that service satisfies the criteria passed in as arguments. The function `conformsTo` takes two profiles and determines whether the first profile conforms to the second profile. To find a matching service, the `find` method checks whether the server properties conform to the client requirements, and it checks whether the client properties conform to the server requirements.

```
struct ServiceProperties {
  profile properties;
  profile requirements;
};

struct Criteria {
  profile properties;
  profile requirements;
}
```

FIG. 17. The structure of `ServiceProperties` and `Criteria`

```
// C++ Implementation sketch of the find method
// in a QoS-based trader
Match QoS-Trader::find(const Criteria &cr)
 throw(noMatch)

 ServiceIterator it = ...;
 for(it.init(); ! it.done(); it.advance())
   serviceProperties sp =
     it.currentServiceProperties();
   if(conformsTo(sp.properties,cr.requirements) &&
      conformsTo(sp.requirements,cr.properties)){
        return it.currentServiceMatch();
      }

 throw noMatch();
;
```

FIG. 18. Matching based on conformance

4.2 Using a QoS-Based Trader

Here, we use a QoS-based trader to implement the currency trading system from the introduction section. Currency trading is a complex process that requires significant information and analysis [13]. Although we appreciate the complexity of systems supporting such trades, we have to simplify the problem for the purpose of this paper.

As a reminder, our simple currency trading system consists of three logical components: a front-end that serves as a user interface, a rate service that provides information about current exchange rates, and a currency trading service to execute currency trades. We will focus only on the rate service that provides exchange rate information. The structure of this simple currency trading systems was shown in Figure 2.

Assume that we want to build a currency trading system that uses a rate service available on the Internet. Different service providers may implement different rate services with the same basic functionality, but with different QoS properties. For example, one rate service may provide frequent exchange rate updates and thus higher precision. Another rate service may provide less frequent update, which implies that the information will not be as accurate. Different services may provide information for different currencies. Moreover, one rate service may be highly available and expensive to use, whereas another rate service may be more unreliable but cheaper to use. The point is that one size does not fit all: different clients have different requirements or expectations about the QoS delivered by a rate service, and different clients are willing to pay for high levels of QoS, whereas other clients are not.

The QoS-based trader provides a mechanism for performing server selection in this kind of environment. The various rate services register themselves with the QoS-based trader and provide QoS specifications that reflect their particular notion of QoS. Clients then consult the QoS-based trader when they want to connect to a rate service. In doing so, clients communicate their QoS expectations to the QoS-based trader to select a suitable service.

Figure 19 shows the structure of the currency trading system with a QoS-based trader. We could of course select between multiple currency trading services as well, but we want to simplify the example and focus on rate services. The issues involved in selecting a currency trading service are similar.

In the following we show how to use the concepts of QML and QRR to create QoS specifications for a couple of rate services and for a front-end. Due to space constraints, we provide a somewhat simplified version of the various QoS specifications.

First, we need a number of contract types to represent the various QoS categories under consideration in the QoS agreement between the front-end and a rate service. Figure 20 outlines these contract types. The first type, called `DataQuality` captures the notion of data quality: accuracy and content. The `currencies` dimension reflects the currencies that a particular service can provide information on. The `updateFrequenct` reflects how often this rate information is updated at the server and thus gives an indication of the precision. The `Reliability` contract type captures the reliability of the rate service. In our previous work, we identified a number of dimensions to characterize reliability for distributed object systems [10]. Here, we only use a very simple characterization. We use two dimensions: availability is the probability that the server is up when trying to contact it and reference validity states whether the object reference to the server remains valid after the server has crashed and come back up. Finally, the `ClientPriceBound` contract type captures how much the front-end is willing to pay for the rate service. Specifications of this type can characterize an upper bound, or the exact price, for the service cost as seen by the client. An upper bound does not determine the actual price the service charges, it merely serves as a first-order filtering criterion when matching up clients and services. The actual determination of how to charge for a service may involve more sophisticated negotiation protocols that we do not consider here. Payment can be specified both as a per-invocation cost and a per-session cost, where the per-session cost is determined from the length of the session.

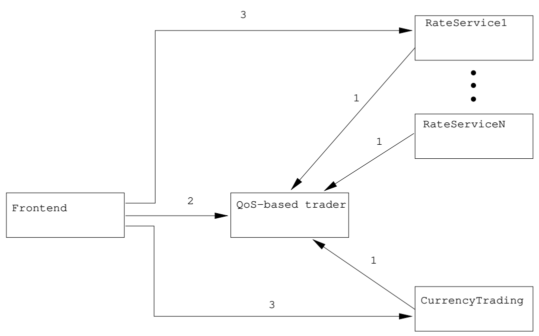

FIG. 19. Structure of the currency trading system with a QoS-based trader

```
type DataQuality = contract {
  currencies: set { USD, JPY, SEK, FIM , DKK, DEM
                ITL, KGS, EEK, KYD, BWP, LUF, FRF,
                GBP, QAR,  RUR, TOP, MAD, SAR };
  updateFrequency: increasing numeric updates/min;
};

type Reliability = contract {
  availability: increasing numeric;
  referenceValidity = increasing
    enum { invalid, valid }
    with order { invalid < valid };
  ...
}

type ClientPriceBound = contract {
  costPerInvocation: decresing numeric cent/call;
  costPerHour: decreasing numeric cent/hour;
}
```

FIG. 20. Contract types for the currency trading system

Given these basic contract types, we can now start to define the QoS properties of services. To simplify the example, we only specify service properties and client requirements. In other words, we do not consider client properties and server requirements. Moreover, we specify the contracts as default contracts—contracts that apply to the rate service object rather than individual methods in this object.

In Figure 21, we outline the QoS properties for two different rate services. Notice that both services implement the `RateService` interface defined in Figure 3. The first service is characterized by the `service1props` profile. This service is a highly available, expensive service that supports many currencies with a low update frequency. The second service is characterized by the `service2props` profile. The second service does not provide any reliability guarantees. On the other hand it is fairly cheap to use. It supports only a few currencies, but the frequency of updates is high. Both services can only charge based on a per-invocation scheme, they cannot charge for entire sessions.

```
service1props for rateService = profile {
  require DataQuality contract {
    currency >= { USD, JPY, SEK, FIM , DKK, DEM
                ITL, KGS, EEK, KYD, BWP, LUF, FRF,
                GBP, QAR,  RUR, TOP, MAD, SAR };
    updateFrequency >= 1;
  };
  require Reliability contract {
    availability >= 0.99;
    referenceValidity == valid;
  };
  require Price contract {
    costPerInvocation <= 100;
  };
};

service2props for rateService = profile {
  require DataQuality contract {
    currencies == { SEK, FIM, DKK, MAD, GBP };
    updateFrequency >= 60;
  };
  require Price contract {
    costPerInvocation <= 50;
  };
};
```

FIG. 21. Profiles for rate services

We can now create profiles, using QRR, during server initialization. One service will create a profile according to the `service1props` specification, and the other service will create a profile according to the `service2props` specification. We can create the QRR profiles by calling the profile-creation functions emitted by the QRR compiler. Alternatively, we can bypass the QRR compiler and create the profiles by calling the QRR library functions directly. For simplicity we assume that each service has a single profile. In a more realistic situation, each service may have multiple profiles to handle service differentiation and QoS variations over time.

Both services will register with the QoS-based trader, and when registering, they will provide their respective profiles. In a dynamic environment servers can withdraw old offers that can not be supported any more and make new offers.

We also need to specify the requirements of the front-end. We give an example of such a specification in Figure 22. According to the figure, the front-end has no reliability requirements, it is willing to pay medium cost on a per-invocation basis, it requires rates for only Swedish Crowns, Danish Crowns, and British Pounds, and it requires relatively high update frequency. This specification can either reflect the requirements of the front-end object as such, or it can represent the requirements of a user of the front-end object. In the first case, we can write the profile in QML and generate profile-creation functions based on the QML specification—we know the QoS requirements when we implement the front-end. In the second case, where the requirements reflect user requirements, we do not know the requirements until runtime, and we cannot compile profile creation functions into the front-end. In this case, we have to call the generic profile creation functions in QRR to dynamically create a profile that reflects the user's requirements.

Once the front-end has created a profile that reflects its requirements, it can then call `find` on the QoS-based trader to obtain a reference to a rate service object that satisfies those requirements. In our case, the front-end's requirements will give rise to selection of service number two. The profile `service2props` conform to the profile `frontendReqs`, where as the profile `service1props` does not.

```
frontEndReqs for rateService = profile {
  require DataQuality contract {
    currency == { SEK, DKK, GBP};
    updateFrequency >= 10;
  };
  require Price contract {
    costPerInvocation <= 75;
  };
}
```

FIG. 22. Front-end profile

4.3 Discussion

A QoS-based trader facilitates the server-selection process in open systems where clients are not built to work with one particular server. In an open system, clients must be prepared to select from a range of different services that provide the same functionality at different levels of QoS. The QRR fabric makes it relatively straightforward to implement a QoS-based trader. In contrast, it is non-trivial to implement a similar functionality using a conventional trader with name-value pairs.

The QoS-based trader that we presented here does not solve the whole issue of establishing QoS agreements between clients and services. We ignored the issue of agreement duration. Furthermore, we only touched on the topic of payment for QoS. In the example, we described a very simple way to perform a first-order screening based on how much clients are willing to pay. What clients actually pay may be less than this upper bound, and will probably be the result of further negotiation between the client and the server given that the upper bound is satisfied. Finally, the trader does not address the issue that services may provide different levels of QoS depending on the dynamic environment. For example, a service may be able to provide higher levels of QoS in an environment with plentiful resources and few clients. We believe that these issues can be addressed as extensions to the simple QoS-based trader that we described. The issues will not change the basic functionality of the trader. Many of the issues can possibly be addressed as separate QoS components that complement the trader.

5. Related Work

Generally, interface definition languages, such as OMG IDL [14], specify functional properties, but lack any notion of QoS.

TINA ODL [20] is different in that it allows programmers to associate QoS requirements with streams and operations. A major difference between TINA ODL and our approach is that they syntactically include QoS requirements within interface definitions. Thus, in TINA ODL, one cannot associate different QoS properties with different implementations of the same functional interface.

Similarly, Becker and Geihs [1] extend CORBA IDL with constructs for QoS characterizations. Their approach suffers from the same problem as TINA ODL: they statically bind QoS characterizations to interface definitions. They also allow QoS characteristics to be associated only with interfaces, not individual operations. In addition, they support only limited domains and do not allow enumerations or sets. Finally, they allow inheritance between QoS specifications, but it is unclear what constraints they enforce to ensure conformance. QoS specifications are exchanged as instantiations of IDL types without any particular structure.

There are a number of languages that support QoS specification within a single QoS category. The SDL language [11] has been extended to include specification of temporal aspects. The RTSynchronizer programming construct allows modular specification of real-time properties [17]. In [7], a constraint logic formalism is used to specify real-time constraints. These languages are all tied to one particular QoS category, namely timing. In contrast, QML and QRR are general purpose; QoS categories are user-defined types in QML, and can be used to specify QoS properties within arbitrary categories.

The specification and implementation of QoS constraints have received a great deal of attention within the domain of multimedia systems. In [18], QoS constraints are given as separate specifications in the form of entities called QoS Synchronizers. A QoS Synchronizer is a distinct entity that implements QoS constraints for a group of objects. The use of QoS Synchronizers assumes that QoS constraints can be implemented by delaying, reordering, or deleting the messages sent between objects in the group. In contrast to QML, QoS Synchronizers not only specify the QoS constraints, they also enforce them. The approach in [19] is to develop specifications of multimedia systems based on the separation of content, view, and quality. The specifications are expressed in Z. The specifications are not executable per se, but they can be used to derive implementations. In [2], multimedia QoS constraints are described using a temporal, real-time logic, called QTL. The use of a temporal logic assumes that QoS constraints can be expressed in terms of the relative or absolute timing of events. Campbell [4] proposes pre-defined C-language structs that can be instantiated as QoS specifications for multimedia streams. The expressiveness of the specifications are limited by the C language, thus there is no support for statistical distributions. Campbell does, however, introduce separate attributes for capturing statistical guarantees. It should be noted that Campbell does not claim to address the general specification problem. In fact, he identifies the need for more expressive specification mechanisms that include statistical characterizations. In contrast to QML, the multimedia-specific approaches only address QoS within a single domain (multimedia). Moreover, these approaches tend to assume stream-based communication rather than method invocation.

Zinky et al. [23, 22] present a general framework, called QuO, to implement QoS-enabled distributed object systems. The notion of a *connection* between a client and a server is a fundamental concept in their framework. A connection is essentially a QoS-aware communication channel; the expected and measured QoS behaviors of a connection are characterized through a number of *QoS regions*. A region is a predicate over measurable connection quantities, such as latency and throughput. There approach does not seem to enable dynamic creation, communication, and manipulation of QoS specifications. In particular, it is not clear how to use their approach to dynamically establish connections in an open environment based on QoS needs and provisions.

In [21] Zinky and Bakken discuss the problem of managing meta-information in systems with adaptable QoS. The paper discusses various kinds of data that is needed for adaptive CORBA systems. They do not, however, present any concrete way in which the information can be described and communicated. We believe that QML and QRR can be used to described several of the facets—such as *kind* and *mechanism*—identified in the paper.

Linnhoff-Popien and Thissen [12] describe methods for evaluating the distance from the ideal characteristics of a required service to what the available offers provide. They use computed distances to select the most appropriate service. In contrast, QML and QRR focuses on statistical characterization of QoS and systematic comparison by means of conformance. We could extend our approcah to incorporate a notion of "preference" if many services satisfy a client's requirements. The area of utility theory is a promising foundation for such an extension.

Within the Object Management Group (OMG) there is an ongoing effort to specify what is required to extend CORBA [14] to support QoS-enabled applications. The current status of the OMG QoS effort is described in [16], which presents a set of questions on QoS specification and interfaces. We believe that our approach provides an effective answer to some of these questions. ISO has an ongoing activity aiming at the definition of a reference model for QoS in open distributed systems. In a recent working paper [8] they outline how various dimensions such as delay and reliability could be characterized. They lack, however, any proposal or recommendations for representations or languages with which such constraints can be expressed and communicated.

6. Conclusion

We believe that one of the next major advances of distributed object systems is to make them QoS enabled. The first step towards QoS enabling is to facilitate QoS characterizations of distributed object components. We have previously suggested the QML language for this purpose [6]. When we have characterizations we need to allow these to influence how distributed objects are connected and what underlying communication mechanism and transports they use. Currently, these decisions are typically made at design time and hardwired into the system. However, to build flexible applications that execute in internet-like environments, we need to support dynamic connections based on QoS matching. The dynamic connections can be facilitated by QoS components, such as traders and negotiators. A QoS characterizations is of little use if we can not verify that components of a system actually complies to the QoS agreements that have been set up among them. This can be accomplished by monitoring connections between objects.

Trading, negotiation, monitoring and many other functions of QoS-enabled distributed systems *require* a format for exchanging QoS specifications. We have designed and are implementing a language (QML) and a run-time representation (QRR) that is tailored for making distributed object systems QoS aware. One of the main advantages of using QRR instead of ad-hoc runtime representations is that QRR comes with a precisely defined notion of conformance. Moreover, the QRR library comes with a generic conformance checking function. Although conformance appears somewhat manageable for simple constraints over numeric dimensions, it is challenging to define and check conformance for statistical aspects and set domains with user-defined ordering.

QML and QRR allow middle-ware developers to invent new mechanisms and services for QoS-enabled distributed systems.

OMG has standardized—among other things—CORBA IDL and IIOP to facilitate interoperability of heterogeneous distributed objects. Analogously, we believe that open systems can only meet QoS requirements if they can specify and communicate their QoS characteristics and requirements. QML and QRR could be viewed as a first attempt to come up with a common specification language and inter-change format for QoS enabled distributed object systems.

References

1. Christian R. Becker and Kurt Gheis. Maqs—management for adaptive qos-enabled services. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997.
2. G. Blair, L. Blair, and J. B. Stefani. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and ISDN Systems*, 29, 1997. Special Issue on Specification Architecture.
3. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language*. Rational Software Corporation, January 1997. version 1.0.
4. Andrew T. Campbell. *A Quality of Service Architecture*. PhD thesis, January 1996.
5. Svend Frølund and Jari Koistinen. Qml: A language for quality of service specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories, February 1998.
6. Svend Frølund and Jari Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, April 1998.
7. Gopal Gupta and Enrico Pontelli. A constraint-based approach for specification and verification of real-time systems. In *Proceedings of Real-Time Systems Symposium*. IEEE, December 1997.
8. ISO. Working draft for open distributed processing—reference model—quality of service. Result from the SC21/WG7 Meeting, July 1997.
9. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1), February 1988.
10. Jari Koistinen. Dimensions for reliability contracts in distributed object systems. Technical Report HPL-97-119, Hewlett-Packard Laboratories, October 1997.
11. S. Leue. Specifying real-time requirements for sdl specifications—a temporal logic-based approach. In *Proceedings of the Fifteenth IFIP WG6 (Protocol Specification, Testing, and Verification XV)*, June 1995.
12. C. Linnhoff-Popien and D. Thissen. Integrating qos restrictions into the process of service selection. In *Proceedings of the Fifth IFIP International Workshop on Quality of Service*, May 1997.
13. C. Luca. *Trading in the Global Currency Markets*. Prentice-Hall, 1995.
14. Object Management Group. *The Common Object Request Broker: architecture and specification*, revision 2.0 edition, July 1995.
15. Object Management Group. *CORBA Services — Trader Service*, formal/97-07-26 edition, July 1997.
16. Object Management Group. *Quality of Service: OMG Green paper*, draft revision 0.4a edition, June 1997.
17. S. Ren and G. Agha. Rtsynchronizer: Language support for real-time specifications in distributed systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*. ACM, June 1995.
18. S. Ren, N. Venkatasubramanian, and G. Agha. Formalizing multimedia qos constraints using actors. In *Proceedings of the Second IFIP International Conference on Formal Methods for Open, Object-Based Distributed Systems*, 1997.
19. R. Staehli, J. Walpole, and D. Maier. Quality of service specification for multimedia presentations. *Multimedia Systems*, 3(5/6), November 1995.
20. Telecommunications Information Networking Consortium. *TINA Object Definition Language*, June 1995.
21. J. A. Zinky and D. E. Bakken. Managing systematic metadata for creating qos-adaptive corba applications. In *Proceedings of the Fifth IFIP International Workshop on Quality of Service*, May 1997. Short paper.
22. J. A. Zinky, D. E. Bakken, and R. D. Schantz. Overview of quality of service for distributed objects. In *Proceedings of the Fifth IEEE conference on Dual Use*, May 1995.
23. J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1), 1997.