Scalable QoS-Based Event Routing in Publish-Subscribe Systems

Nuno Carvalho
Filipe Arajo
Lus Rodrigues

DI–FCUL TR–05–4

February 2005

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Scalable QoS-Based Event Routing in Publish-Subscribe Systems[*]

Nuno Carvalho
University of Lisbon
nunomrc@di.fc.ul.pt

Filipe Araújo
University of Lisbon
filipius@di.fc.ul.pt

Luís Rodrigues
University of Lisbon
ler@di.fc.ul.pt

Abstract

For many distributed applications, the Publish-Subscribe communication model emerges as a viable alternative to the Request-Reply model. It provides a strong decoupling among participants, simplify the reutilization of components and the non-stop reconfiguration of applications. Unfortunately, this strong decoupling also makes it hard to support Quality of Service (QoS) parameters, like bandwidth and latency, in an efficient manner.

This paper presents a novel approach to support QoS parameters in publish-subscribe systems. It proposes a model that supports the decoupling of QoS characterization from the event characterization while offering, at the same time, an uniform treatment of both aspects. Furthermore, it describes the architecture of a distributed and scalable publish-subscribe broker with support for QoS. The broker, called IndiQoS, *leverages on existing mechanisms to reserve resources in the underlying network and on an overlay network of peer-to-peer rendezvous nodes, to automatically select QoS-capable paths. By avoiding flooding of either QoS reservations or link-state information,* IndiQoS *is able to scale with respect to network size and number of reservations. Experimental results show the validity of our approach.*

1 Introduction

The indirect communication, in particular the publish-subscribe communication model, is gaining increasing acceptance as a useful alternative to direct communication models, such as the ones based on remote invocations. The main advantage of this paradigm is its support for a weak coupling among participants, which do not need to be aware of the location or number of its peers. This simplifies the reconfiguration of the applications and eases the re-use of the same components in different applications.

A limitation of most existing architectures that support the publish-subscribe communication is their limited support for the negotiation or enforcement of Quality of Service (QoS) parameters (such as required bandwidth or latency). This observation applies both to models, such as the CORBA Event Service [14], CORBA Notification Service [15], Java Message Service [22] and to systems, such as CEA (Cambridge Event Architecture) [3], Distributed Asynchronous Collections [10] or SIENA (Scalable Internet Event Notification Architectures) [8]. This is a significant drawback, since QoS features are an important component of applications, and its use and support has been widely studied in the context of direct communication [5, 6, 23, 4].

There is a fundamental reason for the current state of the art: traditional approaches to QoS provision are based on the establishment of channels or connections that reserve the resources required to provide the desired QoS parameters. This mode of operation fits in a natural way in the direct communication model, where connections are always explicitly setup, but it has an inherent mismatch with the decoupled nature of event based systems. In the indirect communication model, the applications should not be forced to explicitly setup channels. Instead, they should remain oblivious to the number and location of the participants involved in the communication and should be concerned exclusively with the properties of the information they are able to publish or subscribe.

Therefore, a new system model has to be designed to allow the seamless integration of QoS features in indirect communication systems. This model should:

- Allow the application to indirectly negotiate QoS parameters, by allowing it to express QoS properties as a characterization of the information being produced or subscribed.

- Delegate on the message broker the task of establishing the required low-level connections, on behalf of publishers and subscribers. These reservation need to be based on dynamic information, like number, location and characteristics of producers and consumers and also on the QoS characteristics of the information

[*]This work was partially supported by FCT and the *IndiQoS* project (POSI/CHS/41473/2001)

exchanged in the system.

In this paper we propose and evaluate an architecture, called *IndiQoS*, which allows QoS parameters to be treated in a uniform way with regard to other event attributes in publish-subscribe systems. In particular, similarly to well known subject-based, content-based, or type-based subscriptions, it is possible to make QoS-aware subscriptions.

Naturally, a key component of the architecture is a scalable QoS-aware distributed message broker that is able to automatically perform the QoS reservations on behalf of publishers and subscribers. Our broker builds on top of network-level QoS architectures, such as the Integrated services [5] and the Differentiated Services [4], and leverages on recent results of peer-to-peer systems, to provide a QoS support to publish-subscribe systems that is highly scalable. Experimental results show that the *IndiQoS* architecture provides a favorable trade-off between the resulting network utilization, the end-to-end latency between publishers and subscribers, and the required signaling cost.

The rest of the paper is organized as follows. Section 2 introduces the QoS-aware publish-subscribe model used in *IndiQoS* and Section 3 the requirements of a QoS-aware distributed message broker. An overview of previous work in presented in Section 4. The *IndiQoS* architecture is described in Section 5 and evaluated in Section 6. Finally, Section 7 concludes the paper.

2 QoS-Aware Publishing and Subscribing

One of the main advantages of the publish-subscribe model is that it decouples publishers and subscribers in several dimensions. In [9] three dimensions of decoupling are introduced: *space* decoupling (interacting parties do not need to known each other); *time* decoupling (parties do not need to be actively participating in the interaction at the same time); and *flow* decoupling (asynchrony of the model). In this paper we address a fourth dimension of decoupling, *QoS decoupling*, that captures the separation of QoS parameters from the type or content of events.

The model advocated in this paper[1] has the following characteristics. The QoS of the event dissemination is established in run-time, based on the desired properties expressed by subscribers, on the shape of the sources advertised by the publishers, and on available resources. An important aspect is that subscribers should be able to express QoS constraints using the same type of constructs they use to express other sort of constraints (such as content-based constraints). Publishers, on the other hand, do not tie a specific QoS with the information produced. However, they must advertise the *shape* of the information being produced, in the form of an *event QoS profile*. The event QoS profile is used in run-time by the message broker to estimate the resources demanded by a given flow and to match the QoS constraints specified by subscribers with the characteristics of the information produced by publishers. The message broker plays an important role in a QoS-aware publish-subscribe system, because it must ensure that QoS requirements are met. Besides, the message broker must cope with QoS related parameters present in advertisements, notifications and subscriptions.

[1]The model has been originally proposed by the authors in a position paper in [2].

To make our case, we will use the following example. Consider a building where rooms are equipped with a number of temperature sensors. These sensors advertise the room temperature in an event of type *Temp*. The attributes of these events are: *room*, which identifies the room where the temperature is being measured; *temperature*, which identifies the room temperature; and *precision*, which identifies the precision of the sensor.

Our case is independent of any particular language construct to be used when specifying notifications or subscriptions. In the following examples we will follow a notion that closely resembles the type-based publish-subscribe model of [10]. Using this model, a typical subscription would be:

```
Subscriber s = subscribe Temp
               where (temperature > 60)
```

The expression corresponds to a subscription of events from any room where the temperature is higher than 60. On the publisher side, the interface is:

```
Publisher p = new Publisher
              of Temp
              withProfile (room="lab1",
                           temperature=any,
                           precision=0.01);
e = new Temp (room="lab1",
               temperature=16,
               precision=0.01)
p.publish (e)
```

The *Publisher* is an auxiliary component that is used to disseminate events. Among other purposes, it allows the publisher to inform the message broker of the type of events it is going to produce. This information takes the form of advertisements. In the example above, we consider only a *content profile*, the profile that characterizes the content of the information being published. In this example, the publisher states that the events it produces may have different values in the *temperature* field but have a fixed value in the *room* and *precision* fields. This information may be used by the broker to optimize the dissemination of events [8]. We will now discuss how to advertise QoS related profile information (in addition to the *content profile*).

Consider now that different sensors have different QoS parameters. For instance, one sensor may produce sporadic events, only when it detects a temperature change, and other sensors may produce new events at a periodic pace, but with different periods. The question is, of course, how to address the QoS characterization of the events, both at the producer and at the consumer. Since we are interested in giving the application designer a uniform interface, we would like to use mechanisms to express the QoS parameters that are similar to the ones used before to express the content of the information being produced.

One possible approach would be to code the QoS information in the event *type*. For instance, one could define two different types: *SporadicSensor* and *PeriodicSensor* and include other QoS information, such as the period, as an attribute of the *PeriodicSensor* type. However, we believe that this approach has several disadvantages. When combined with other QoS attributes, such as reliability or availability, this quickly leads to an explosion of different types for the same information being produced. One of the main reasons to reject this sort of coupling is that some QoS attributes can only be derived at run-time. Consider for instance the case where a subscriber is interested in receiving a temperature event but wants to specify a minimum latency in the event dissemination. Clearly, the latency is not an inherent property of the information being disseminated. Furthermore, latency is a function of several run-time parameters, such as the relative location of the subscriber and the publisher and the load of the links between these participants.

To address these issues we propose an architecture where publication and subscription operations are augmented with QoS attributes that can be used to define filtering conditions in a similar way to that of content-based filtering. In order to do so, publisher must advertise a *profile* of the event publishing pattern. In our example above, sensors should characterize the nature of the event pattern, declaring if it follows a sporadic or periodic profile. For instance, the periodic sensor declares the shape of the information produced as a *QoS profile* that can be provided in addition to the content profile:

```
Publisher p = new Publisher
              of Temp
              withProfile (room="lab1",
                           temperature=any,
                           precision=0.01)
              withQoSProfile Periodic (period = 1)
```

On the subscriber side, the desired QoS attributes are expressed using a filtering condition similar to the one used for the information contents. For instance:

```
Subscription s = subscribe Temp
                 where (temperature > 60)
                 withQoS ((Periodic(period<1))
                         and (latency<10))
```

While advertisements are not mandatory in non-QoS-aware publish-subscribe systems, they are of utmost importance in a QoS-aware system. In fact, some QoS related information, such as the period, is not a characteristic of each individual event but of the *shape* of the traffic produced by the publisher. Given the type of decoupling aimed in the model proposed here, the *profile* of the source must be advertised independently of each individual publish operation.

There are a number of issues regarding this model that need to be emphasized. First, some of the QoS attributes specified in the subscription, such as the latency attribute, have no match in the information being advertised, and must be interpreted by the message broker itself. Other examples include a QoS specification including a reliability attribute that depends of the available transport protocols. Additionally, a subscription may be refused due to lack of system resources. For instance, it may be impossible to satisfy the latency constraint specified in the subscription.

3 QoS-Aware Distributed Message Brokers

Some QoS parameters are already supported in some publish-subscribe models or systems, such as CORBA Notification Service [15], Java Message Service [22] or Distributed Asynchronous Collections [10]. This is the case of message reliability, message priority, message earliest delivery time, message expire time, duplicate message detection or message ordering, for instance. Depending on the architecture, these QoS parameters may be supported or not.

As far as we know, QoS parameters that have been widely studied in the direct communication paradigm, such as latency and bandwidth, are not adequately addressed in publish-subscribe systems. Hence, we envision a message broker that also copes with these QoS parameters. This is a difficult task that is considerably different from ensuring existing QoS parameters such as message reliability or message ordering, for instance. To ensure this sort of QoS parameters it is necessary to do reservation of resources along the path(s) connecting publishers and subscribers. In a publish-subscribe system, to preserve the decoupling among the participants, reservations should be done by the message broker on behalf of the applications. This clearly prompts for the development of QoS aware distributed message brokers.

A QoS-aware message broker is a distributed component that manages the following entities: *i)* Advertisements of publishers, including the *QoS profiles* of the information being advertised; *ii)* Subscriptions, including desired *QoS conditions*; *iii)* System resources. The system resources represent the networking, memory and processing resources available to support the exchange of events. They encapsulate low-level QoS protocols, such as RSVP or other similar mechanisms widely used in direct communication systems [5, 6, 23, 4].

A *naive* implementation of a QoS-aware message broker could rely on a centralized event server: all participants would directly contact the server that would forward the messages from publishers to subscribers. Unfortunately, such solution is inherently non-scalable, as the capacity of the system would be limited by the bandwidth and processing power of the central server. In this paper we are particularly interested in building a scalable QoS-aware message broker, i.e., a broker able to provide service to a large number of participants.

4 Related work

There are two classes of systems that are relevant to the *IndiQoS* architecture: publish-subscribe message brokers (typically without QoS support) and systems to support QoS routing (that can be used to augment publish-subscribe brokers with QoS support). We will now briefly review the most relevant related work in these two classes.

4.1 Publish-Subscribe Message Brokers

There are three main different classes of publish-subscribe systems: *brokerless* systems, where subscribers connect directly to publishers; *centralized broker* systems, and *decentralized broker* systems. The *Cambridge Event Architecture* (CEA) [3] is an example of a *broker-less* system. The system uses the *publish-register-notify* paradigm, where subscribers register directly in the publisher nodes and messages flow directly from the latter to the former. This model does not provide the level of decoupling required by many applications. The *CORBA Event Service* [14], the *CORBA Notification Service* [15] and the *Java Message Service* [12] are examples of publish-subscribe systems with a centralized broker. This approach is not scalable in the number of applications and subscription supported. Examples of architectures using decentralized brokers are the *Scalable Internet Event Notification Architecture* (SIENA) [8], the Scribe [21] and the *Hermes* [17].

SIENA is composed by a network of routers that need to first disseminate all advertisements among them[2] and then use reverse paths for matching subscriptions. Whenever possible, SIENA merges advertisement or subscription messages, to reduce signaling traffic, but the basic need to flood information contained in the advertisements is not eliminated. To preclude the flooding of information, Scribe and Hermes use a Distributed Hash Table (DHT), together with the notion of rendezvous nodes. Subscriptions and advertisements are done on the rendezvous node of the specified type. Using this approach, the system does not need to maintain the information about subscriptions and advertisements in all routers: each event type is associated with one router in a deterministic way and routing is performed by the DHT.

The *IndiQoS* architecture leverages on the Hermes architecture by augmenting it with appropriate QoS routing mechanisms.

4.2 QoS Routing

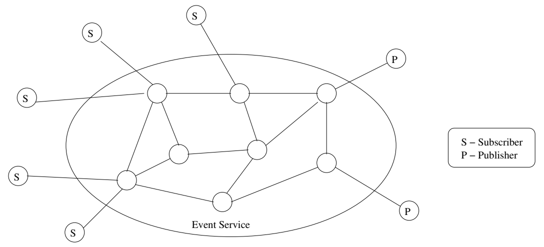
Routing messages using QoS parameters as input variables naturally requires availability of QoS information to the routers. Possible solutions to this problem may range from flooding routers with QoS information, thus enabling routers to locally decide which paths are best, to the other extreme where no QoS information is distributed and any routing decision is taken after flooding the entire data network with a reservation request.

Quality of Service Extensions (QoSPF) is a well-known example of a protocol that tries to maintain updated QoS information at the routers [1]. To support QoS, QoSPF adds two new link state advertisement messages to OSPF: one to describe available resources, the other to describe resources that are reserved (in a given link). Any change in the available resources or in the reservations triggers a new message. In practice this makes QoSPF not scalable, because the additional cost of these updates is not negligible. Despite this weakness, a similar approach is used to support traffic-engineering [16] inside a single autonomous system.

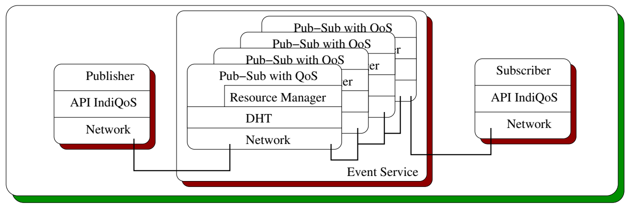
A radically different approach is followed by the algorithm in [18] that keeps QoS information local to the links. However, unlike the previous approach, routers do not have the necessary information to locally select paths. Therefore, whenever an application requests a reservation of resources, it must flood the request throughout the network. This flooding will serve two purposes: i) do a tentative reservation in the links it goes through and ii) collect QoS information kept in the links. This flooding process is kept under control by a pruning mechanism, because paths known to be non-optimal may be discontinued. This may happen at all nodes that receive two or more messages relative to the same reservation. Therefore, there is a wave moving forward with the reservation messages and another one moving backward pruning non-optimal paths. A third message is needed to issue the definitive reservation, whenever an optimal path reaches destination(s). The reader should notice that each reservation might require at least two messages by link in the flooding process.

In a way that is similar to our own approach, there are

[2]Even if advertisement messages were not used, the same would be necessary for the subscription messages.



(a) Overview



(b) Interaction between the components of the system

Figure 1. *IndiQoS* **architecture.**

protocols that try to find a compromise between these two extreme solutions. This is the case of protocols that build trees taking into account QoS parameters. For instance, the *QoS Manager for Internet Connections* (QoSMIC) builds the tree restricting connections based on available bandwidth [24]. When a node wants to join, it connects to the nearest node that has the necessary bandwidth. To do this, each tree node must know the network topology, including the available bandwidth in the connections. This applies also to other protocols, like *QoS Dependent Multicast Routing Algorithm* (QDMR) [11]. However, in the context of a publish-subscribe system, *IndiQoS* is inherently better than any of the previously existing solutions, because it embodies the lightweight structure of a DHT, which requires nodes to have information of only $O(\log n)$ neighbors, keeps QoS link state information local and, unlike [18], uses a restricted dissemination of reservation messages.

5 The *IndiQoS* Architecture

The *IndiQoS* architecture, illustrated in the Figure 1, is a type-based publish-subscribe system that uses a decentralized message broker to connect publishers and subscribers. The message broker is composed by a set of routers, structured in a peer-to-peer overlay network. The routing of events is made by a DHT. Applications connect to the message broker using one of the routers. This architecture is inspired by Hermes [17]. However, *IndiQoS* includes mechanisms to manage the reservation of network resources to provide QoS guarantees with small signaling overhead.

5.1 Decentralized Message Broker

The *IndiQoS* message broker is composed by a set of nodes connected using an overlay network. These nodes behave as event routers that cooperate to form event dissemination trees able to satisfy the QoS requirements requested by the applications. The routing functions required to build the tree are provided by a DHT.

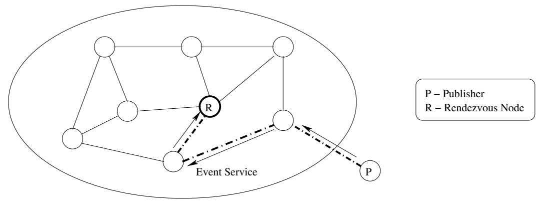
As depicted in Figure 1, each node of the overlay network executes a protocol stack composed of: (i) a publish-subscribe layer, that manages the advertisements and subscriptions and, in response, automatically establishes the reservations required to satisfy the applications; (ii) a DHT (overlay) layer, that supports message routing and (iii) the underlying network layer, encapsulated by abstract network components.

A DHT is a fundamental building block for peer-to-peer applications. Basically, it allows a group of distributed hosts to collectively manage a mapping from keys to nodes. The DHT used in the current implementation of *IndiQoS* is *Bamboo* [19]. Bamboo is based on Pastry [20] that uses the same geometry[3] but relies on alternative neighbor management algorithms that aim at improving the path quality (namely the latency). We exploit these properties of *Bamboo* but we have slightly adapted its behavior to avoid the reconfiguration of the overlay in stable conditions (in order to preserve the stability of established network reservations).

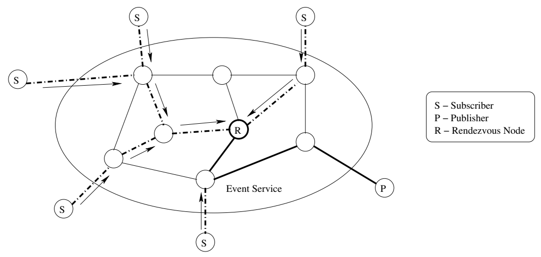
As in *Hermes*, the *IndiQoS* uses the notion of *rendezvous nodes*. The rendezvous nodes are responsible for keeping control information about specific event types. Applications interact with the message broker using one node as the gateway. The gateway uses the rendezvous node to find a path between publishers and subscribers. As illustrated in Figure 2, advertisements and subscriptions related to a given event type are both routed to the same rendezvous node.

One positive aspect of an architecture based on a DHT is that it allows distribution of the load among the nodes of the overlay and it offers a very efficient way for each participant to contact the desired rendezvous node. The DHT just needs an identifier to find a path to the rendezvous node. We use a deterministic algorithm that maps event types to identifiers and with this identifier, the DHT just routes the message to the respective node. Using the DHT do distribute the rendezvous among the nodes of the overlay network we reduce the possibility of getting a bottleneck in some part of the network.

[3]The term *geometry* is used to refer to the pattern of neighbor links in a DHT, independent of the routing algorithms used.



(a) Advertisement



(b) Subscriptions

Figure 2. Paths in the message broker.

5.2 Event Distribution Trees

IndiQoS creates event distribution trees where the publisher is the root of the tree, the subscribers are the leafs, and the rendezvous acts as a bifurcation point. By building the dissemination tree in this way, the rendezvous node may easily become a bottleneck, since all the events of the corresponding type have to pass through this node. To alleviate this effect, any node of the overlay can intercept a subscription: if there is already a path to the rendezvous node that satisfies the subscription, a bifurcation is placed in the intercepting node. Therefore, the distribution tree is constructed in a distributed way: the publisher starts by registering itself, sending an advertisement to the rendezvous node. In a similar manner, subscribers also route the subscription to the rendez vous. The first subscription establishes the necessary resource reservations in the path from the subscriber to the rendezvous (and from the rendezvous to the publisher). Subsequent subscriptions that cross this path do not need to be forward up to the rendezvous: instead, a bifurcation is established at the crossing point. As the system evolves, bifurcations are more likely to be found closer to the subscribers. This is illustrated in Figure 2. As it can be seen in Figure 3, in the final tree, there is a bifurcation of events not only in the rendezvous node R, but also in the routers $R1$, $R3$ e $R4$.

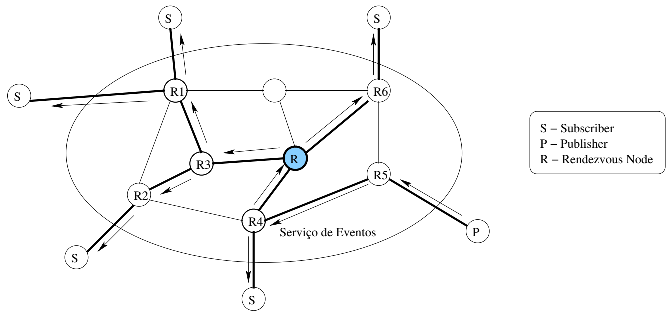


Figure 3. Dissemination tree.

5.3 Quality of Service

The *IndiQoS* architecture is independent of the network-level mechanisms used to reserve resources on individual links. Available mechanisms include those proposed by the Integrated Services or the Differentiated Services architectures. This independence is provided by encapsulating concrete network level mechanisms inside abstract network components called *Infopipes* [13]. Infopipes are software components that can be connected to each other to form an overlay network. These components are connected to each-other by ports. Information items are received by Infopipes through source ports and forward to other Infopipes through destination ports. A set of connected Infopipes is called a *Infopipeline*. There are several types of Infopipes. For instance, *Source* and *Sink* connect Infopipelines to applications; the *Split Tee* simulates multicast, connecting one incoming Infopipe to several outgoing Infopipes; and the *Netpipe* connects two different hosts. The most important Infopipe to be defined in this context is the *Netpipe*. The Netpipe is the Infopipe responsible for transferring events between different hosts. A Netpipe can be defined as a quadruple $\langle S, D, L, W\rangle$, where S is the Source host, D is the Destination host, L is the Length of the Netpipe, which also represents its latency, and finally W, the Width of the Netpipe, represents its maximum bandwidth. This software component also has an interface to the above layer in the *IndiQoS* routers, exporting an API to make reservations, getting information about important parameters of the network. We can have different implementations of the Infopipes and, in particular, we implement the Netpipe using several solutions to make the real resource reservations.

The resources are reserved by a *Resource Manager*, using the *Netpipe* interface. Reserved resources are managed internally by this component. So, is the resource manager that makes the real reservations in the *Netpipes*, distributing them by publications and subscriptions as they arrive. In the *IndiQoS* system, two QoS parameters are guaranteed: latency and bandwidth. Messages are only routed to links that satisfy the requested resources. The reservations made by the Resource Manager are only made locally in each router and only when paths are already defined. A system client, after contacting the rendezvous node, receives a reply that already contains a path that satisfies the QoS requirements. Having this path defined, it is enough to send a control message in the reverse direction to commit the respective reservations in the Resource Manager of each router of the overlay network. As the paths are bound in the rendezvous node, this node has to verify that the requested QoS can be satisfied by the system. When the publisher and the subscriber are both registered, the rendezvous node makes use of the information about the requested bandwidth and latency requested by the subscriber. With this information, the rendezvous node verifies if the QoS can be satisfied. In the case of the presence of several publishers of the same event type, but advertising information with different QoSs, the rendezvous node can also choose the publisher most apt to satisfy a subscriber's request.

5.4 Replication of the Rendezvous Points

The use of a single rendezvous point for each event type may limit the performance, scalability and fault tolerance of the system. If a given rendezvous node fails, all the control information has to be rebuilt as soon as the underlying DHT is reconfigured. Also, the limited bandwidth and processing power of the rendezvous limits the scalability of the system in terms of the number of applications advertising/subscribing to a given event type. Therefore, *IndiQoS* replicates each rendezvous node.

To associate n rendezvous replicas to one type, one can trivially modify the algorithm that maps types to keys of the DHT to deterministically return n keys instead of one. Furthermore, it is necessary to adapt the algorithm that constructs the event distribution trees to support n rendezvous nodes for each type. The publisher, after getting n keys, registers itself in all replicas of the rendezvous. The subscribers, on the other hand, also make request to the n rendezvous replicas, receiving n replies. These replies are then evaluated by the subscriber, who chooses the rendezvous replica that better fits the requested QoS. When the rendezvous replica is chosen, the subscriber registers and is added to the correspondent dissemination tree.

Figure 4 shows a distribution tree for one event type with two rendezvous replicas. A significant advantage on having several rendezvous replicas for each event type is that the paths are distributed and there is no longer a single bottleneck point in the overlay. The compairison between Figure 3 and 4 illustrates the difference between a configuration that uses a single rendezvous and a system that uses two replicas of the rendezvous. As the number of rendezvous replicas grows, the system has more alternative

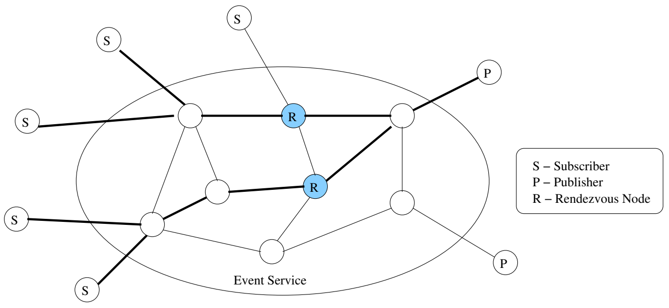


Figure 4. Using a replicated rendezvous.

paths between publishers and subscribers and it becomes more likely finding a path satisfying the QoS requirements of any subscription.

6 Evaluation

This section presents the results obtained in the evaluation of the *IndiQoS* architecture. The results show the benefits of using the DHT and the replication of rendezvous nodes. We also compare our system with different approaches for building the event distribution trees.

To evaluate the *IndiQoS* system, we have used the network simulator provided with the distribution of the Bamboo DHT [19]. The network was generated by GT-ITM [7] using a transit-stub network. In our simulations, there are 252 *IndiQoS* nodes and 30% of these nodes have one subscriber application connected. The system has also one publisher for each event type. The simulations generate subscriptions until the maximum network usage is reached for the tested configuration. Each subscription requires 12.5% of the bandwidth available in each link. The subscription requests are randomly assigned to subscribers and we assume that the maximum network usage was reached when we detect a sequence of 100 consecutive refused subscriptions.

6.1 Benefits the DHT

Using a DHT to implement a system like *IndiQoS* has several advantages. One advantage is that, in opposition to a centralized server approach, the load imposed by subscriptions is distributed, given that subscriptions associated with different event types are routed through different rendezvous points distributed among the network nodes. Another advantage is that the DHT provides a very efficient way for each participant to locate, and contact, the rendezvous point for any given event type. This advantage is inherent to the routing way of the DHT.

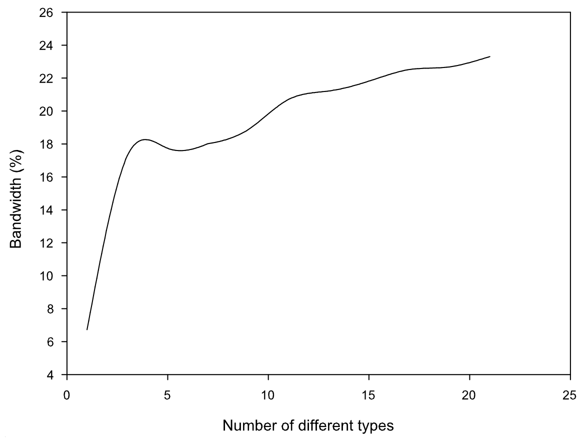


Figure 5. Network utilization vs event types.

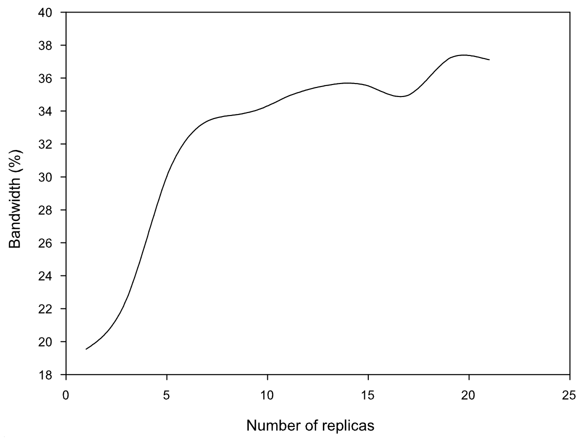


Figure 6. Network utilization vs replication.

Figure 5 shows the increase in network utilization as the number of event-types increases. With a single event type, all advertisements and subscription are managed by the same rendezvous node (this corresponds to a centralized solution). The bandwidth to this node quickly becomes exhausted while other links in the network may remain underutilized. By increasing the number of types, and the number of corresponding rendezvous nodes, a better utilization of system resources is promoted.

6.2 Benefits of Rendezvous Replication

A key aspect of the *IndiQoS* architecture is the replication of rendezvous nodes for each event type. This strategy has two complementary goals. In the first place, it increases the amount of subscriptions supported for each type. Given that there is a limited amount of bandwidth available to each rendezvous node, the replication of the rendezvous nodes increases the available bandwidth for each event type. In second place, when more than one rendezvous node is able to coordinate the reservations for a given subscription, it becomes possible to select the path that offers a better end-to-end latency.

Figure 6 depicts the maximum achievable network utilization as a function of the number of replicas of the rendezvous node for a single event type. As expected, it is observed an increase in the number of subscriptions that are satisfied as the number of replicas of the rendezvous increases.

Figure 7 depicts average latency between the publisher and the subscribers as a function of the number of replicas of the rendezvous node. An interesting aspect of the results is that significant latency gains can be achieved with as few as four replicas, and that further increase in the number of replicas does not provide a significant improvement.

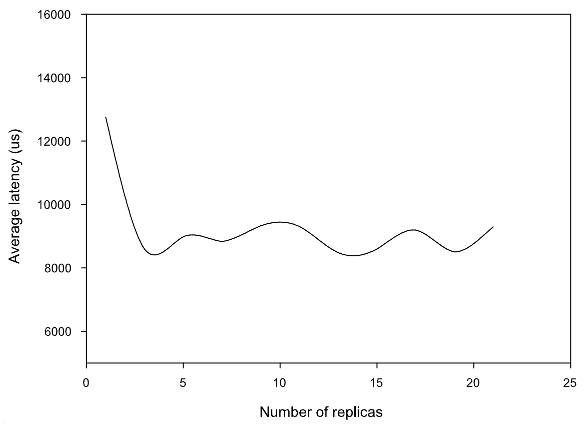


Figure 7. Average latency vs replication.

Naturally, the advantages of augmenting the number of replicas of a rendezvous node come with cost: there is an increase in the signaling required for satisfying a subscription. This happens because a subscription needs to be forward to the different replicas of the rendezvous node. Figure 8 shows the average number of control messages for each request that had success. Increasing the number of replicas also increases the number of control messages to register in the rendezvous nodes and make resource reservations. As wee will show next, when comparing our approach with other alternatives, the signaling cost is competitive for a small number of replicas.

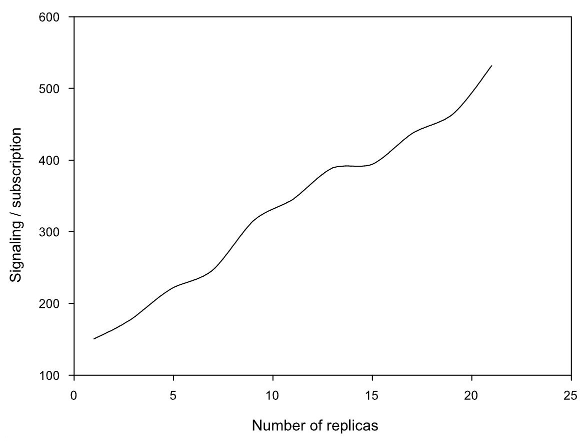


Figure 8. Signaling vs replication.

6.3 Comparison With other Strategies

A fundamental goal of the *IndiQoS* architecture is to implement a scalable message broker. In particular, we are interested in measuring the signaling costs of our solution when compared with 1 and 3 replicas of the rendezvous node (*IndiQoS* 1 PC and 3 PC). Is also compared to other solutions. There are two alternative approaches that we have used for comparison:

i) One approach consists in using a link-state protocol to ensure that every node keeps an up-to-date representation of the network state (FNS). As a result, each node can autonomously select the best path to satisfy a given subscription. This approach is used in commercial traffic-engineering solutions (such as [16]): it requires each node to keep the state of the complete network and to flood a link-state update whenever the bandwidth of a link changes significantly.

ii) Another approach consists in flooding a subscription request to every node of the network in order to find an acceptable path (FR). This approach, used in [18], does not require every node to keep up-to-date information about the state of the network, but has a significant signaling cost associated with each subscription.

Figure 9 shows the signaling cost of *IndiQoS* against these two alternatives. As it can be seen, the signaling cost is substantially smaller (five times less). Naturally, given that *IndiQoS* operates without global knowledge of the network conditions, it cannot find paths as good as the other approaches. However, it can be seen that the increase in latency is smaller when compared with the signaling gains.

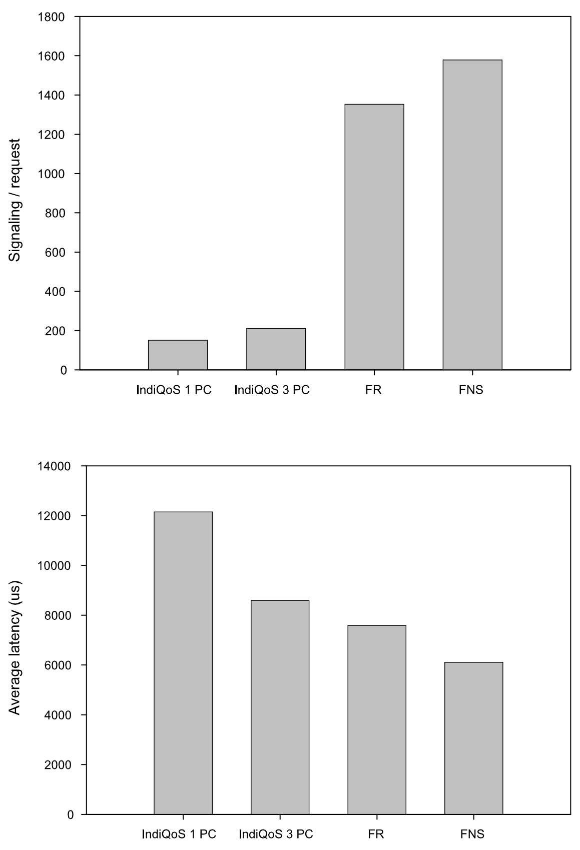


Figure 9. Comparison.

7 Conclusions and Further Work

The paper presented the *IndiQoS* architecture, a scalable QoS-aware publish-subscribe system. With QoS-aware publications and subscriptions that preserve the decoupling that makes the publish-subscribe model so appealing. Using QoS based subscription, consumers of information may specify in a declarative manner both the type, content and QoS attributes such as latency, reliability, etc, of the information they are interested in. To support such model, the *IndiQoS* includes a decentralized message-broker based on a Distributed Hash Table that leverages on underlying network-level QoS reservation mechanisms. To increase the network usage and to reduce the end-to-end latency, and still offer low-cost signaling, we propose to replicate the rendezvous points for each event type. Experiments show that the resulting system offers a small signaling overhead without a significant performance penalty (end-to-end latency and network utilization), when compared to solutions that require the system to maintain or obtain global knowledge.

References

[1] G. Apostolopoulos, D. Williams, S. Kamat, R. Guerin, A. Orda, and T. Przygienda. QoS routing mechanisms and OSPF extensions, August 1999. RFC 2676.

[2] F. Araújo and L. Rodrigues. On QoS-aware publish-subscribe. In *Proceedings of the International Workshop on Distributed Event-Based Systems*, pages 511–515, Vienna, Austria, July 2002. IEEE. (Proceedings the 22nd International Conference on Distributed Computing Systems Workshops).

[3] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, March 2000.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differenciated services, December 1998. RFC 2475.

[5] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, June 1994. RFC 1633.

[6] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) — version 1 functional specification, September 1997. RFC 2205.

[7] K. Calvert, M. Doar, and E. Zegura. Modeling internet topology. *IEEE Communications Magazine*, June 1997.

[8] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[9] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[10] P. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *In 14th European Conference on Object Oriented Programming (ECOOP 2000)*, pages 252–276, June 2000.

[11] L. Guo and I. Matta. QDMR: An efficient QoS dependent multicast routing algorithm. In *Proc. of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, 1999.

[12] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. *Java Message Service*. Sun Microsystems, April 2002.

[13] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *Proceedings of the International Workshop on Multimedia Middleware*, pages 44–47. ACM, October 2001.

[14] OMG. *Event Service Specification*. Object Management Group, March 2001.

[15] OMG. *Notification Service Specification*. Object Management Group, August 2002.

[16] E. Osborne and A. Simha. *Traffic Engineeering with MPLS*. Cisco Press, 2003.

[17] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *22nd IEEE International Conference on Distributed Computing Systems Workshops (DEBS '02)*, 2002.

[18] H. Pung, J. Song, and L. Jacob. Fast and efficient flooding based QoS routing algorithm. In *Proceedings of IEEE ICCCN99*, pages 298–303, September 1999.

[19] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. Technical report, University of California at Berkeley, December 2003.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[21] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[22] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303, USA. *Java Message Service*, November 1999.

[23] J. Wroclawski. The use of RSVP with IETF integrated services, September 1997. RFC 2210.

[24] S. Yan, M. Faloutsos, and A. Banerjea. QoS-aware multicast routing for the internet: The design and evaluation of QoSMIC, February 2002.