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

August 9, 2005

Abstract

This paper proposes 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 the support for a weak coupling among participants, which do not need to be aware of the location or number of their 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).

*Selected sections of this report were published in the Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05), July, 2005, Cambridge, MA, USA. This work was partially supported by the LaSIGE and by the FCT project INDIQoS POSI/CHS/41473/2001 via POSI and FEDER funds.

This observation applies both to models, such as the CORBA Event Service [13], CORBA Notification Service [14], Java Message Service [21] 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, 22, 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 necessary resources. This mode of operation has an inherent mismatch with the decoupled nature of event based systems, where peers do not explicitly set up connections. Therefore, a new system model has to be designed to allow the seamless integration of QoS features in indirect communication systems. This model should: *i*) 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; *ii*) delegate on the message broker the task of establishing the required low-level connections.

Ensuring QoS to applications is a challenging problem, because QoS routing requires up to date QoS link state information [1]. On the other hand, updating link state can clog the network, specially if the system updates this information too often to keep accurate routing decisions. To overcome this problem some systems do not update QoS link state periodically and only gather this information at reservation time [17]. This has a big signaling cost. Hence, the contribution of this paper is a QoS-aware event router based on the Bamboo DHT [18], called "*IndiQoS*", that avoids the problems that are common to the aforementioned solutions. *IndiQoS* does not flood *any* QoS link state information. Instead, it makes a constant number of deterministic attempts to find a path capable of ensuring the QoS requirements. *IndiQoS* allows QoS parameters to be treated in a uniform way with regard to other event attributes in publish-subscribe systems. Additionally, *IndiQoS* automatically performs the QoS reservations on behalf of publishers and subscribers and uses network-level QoS architectures, such as the Integrated services [5] and the Differentiated Services [4] to enforce the reservations. For these reasons, *IndiQoS* should be fault tolerant, scalable, both as an event router and as a QoS-aware system and should fit transparently in publish-subscribe applications. In fact, 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 [9]: *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[1], we address a fourth dimension of decoupling, *QoS decoupling*, that captures the separation of QoS parameters from the type or content of events. Publishers and 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). It is up to the message broker to match the advertisements with the subscriptions and to ensure the QoS requirements.

We propose an architecture where publications and subscriptions are augmented with QoS attributes that define filtering conditions in a similar way to that of content-based filtering. To do so, the subscriber must include a *profile* of the events that it wants to receive (the purpose of the bandwidth will be seen ahead). We consider the example of a sensor that measures the speed of vehicles in a road. This sensor acts as a publisher. A photographic camera with timeliness requirement subscribes to this information as follows:

```
01  Subscriber s = subscribe VehicleInfo
02                 where ((type = any)
03                        and (speed > 50))
04                 withQoS ((bandwidth = any)
05                        and (latency < 100))
```

The condition *latency* $<$ *100* will eliminate all the paths from publishers that cannot meet this timeliness requirement. On the other hand, there are two reasons to make the publisher advertise the *profile* of the events. First, because receivers may use it to specify the type of events that they want. Consider for instance low quality and high quality voice. In this case, both kinds of events could use the same type with a different value in the attribute *bandwidth*. The other reason to advertise the *profile* of the events is to let the underlying middleware system determine the requirements of the communication and act accordingly. Hence, in the case of the speed sensor, the advertisement would look like this:

```
01  Publisher p = new Publisher
02                of VehicleInfo
03                withProfile (type = 1,
04                             speed = any)
05                withQoSProfile (bandwidth = 1)
```

Advertisements are very important in a QoS-aware system. In fact, some QoS related information, such as the occupied bandwidth is not a characteristic of each individual event but of the *shape* of the traffic produced by the publisher.

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

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. Also, note that a subscription may be refused due to lack of system resources.

3 QoS-Aware Distributed Message Brokers

Some QoS parameters are already supported in some publish-subscribe models or systems, such as CORBA Notification Service [14], Java Message Service [21] 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. Unlike message reliability or message ordering, the sort of QoS parameters that we aim to ensure requires a 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, 22, 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 with 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* [13], the *CORBA Notification Service* [14] and the *Java Message Service* [12] are examples of models that use a broker that is conceptually centralized. Centralized implementations of these paradigms are not scalable in the number of applications and subscriptions supported. As examples of architectures using decentralized brokers, we have the *Scalable Internet Event Notification Architecture* (SIENA) [8], the Scribe [20] and the *Hermes* [16].

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 [19] (DHT), together with the notion of rendezvous nodes. The fundamental idea of these systems is that subscriptions and advertisements meet at the rendezvous node of the specified type. In this way, 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,

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

a similar approach is used to support traffic-engineering [15] inside a single autonomous system.

A radically different approach is followed by the algorithm in [17] 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.

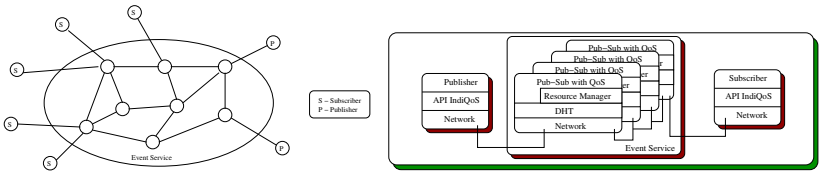
In a way that is similar to our own approach, there are protocols that try to find a compromise between these two extreme solutions. This is the case of protocols that build trees taking into account QoS parameters. For instance, the *QoS Manager for Internet Connections* (QoSMIC) builds the tree restricting connections based on available bandwidth [23]. 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 [17], 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 [16]. However, *IndiQoS* includes mechanisms to manage the QoS resources.

5.1 Decentralized Message Broker

The *IndiQoS* message broker is comprised of a set of nodes connected using an overlay network. These nodes behave as event routers that cooperate to form



(a) Overview

(b) Interaction between the components of the system

Figure 1: *IndiQoS* architecture.

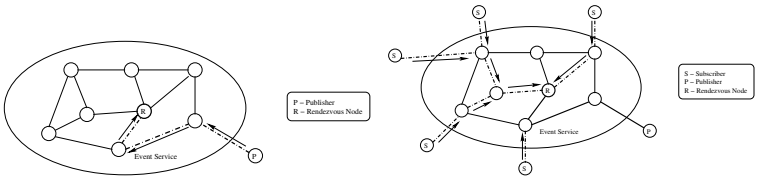
event dissemination trees able to satisfy the QoS requirements requested by the applications. The routing functions required to build the tree are provided by a DHT.

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 distributed applications. Basically, it allows a group of distributed hosts to collectively manage a mapping from keys to values using a hash function. The DHT used in the current implementation of *IndiQoS* is *Bamboo* [18]. Bamboo is based on Pastry [19] 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. The rendezvous node for a given type T is the node numerically closest to the output $hash(T)$. It is up to the DHT to route messages targeted to node $hash(T)$ to the correct destination. Applications (publishers and subscribers) interact with the message broker using one node as the gateway. Gateways try to route advertisements (subscriptions) to the rendezvous node that corresponds to the type of that advertisement (subscription). In this way, advertisements and subscriptions of the same type must always meet at some node. This is illustrated in Figure 2. Additionally, one positive aspect of this architecture is that the hash function automatically distributes the load generated by different types among the nodes of the network. This reduces the

[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.

possibility of getting bottlenecks.

5.2 Event Distribution Trees

IndiQoS creates event distribution trees where the publisher is the root of the tree and the subscribers are the leafs. Nodes of the DHT, including the rendezvous node, can act as splitting points for that tree. 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 rendezvous. Only the first subscription needs to reach the publisher (through the rendezvous node). Subsequent subscriptions that cross this path do not need to be forwarded up to the rendezvous: instead, if they can be merged at some crossing node, this node will create a new branch of the tree. As the system evolves, new branches are more likely to be found closer to the subscribers. This is illustrated in Figure 2. In the final tree, splitting of events may occur, not only in the rendezvous node R, but also in other routers.

To maintain the event distribution tree, publishers (subscribers) need to periodically refresh their advertisements (subscriptions).

5.3 Quality of Service

We assume that it is possible to establish QoS reservations in the links of the *IndiQoS* overlay network. Clearly, it would be impossible to create a publish-subscribe system with QoS support without an underlying QoS-aware data network that supports such QoS links. However, a node of the *IndiQoS* network is not required to establish a QoS reservation to every other node of the network. Instead, it must only establish a QoS reservation with its direct peers in the overlay. Furthermore, it is not required to establish an individual reservation for each publisher/subscriber flow. Instead, *IndiQoS* is based on letting direct peers to establish a single reservation for the aggregate traffic managed by *IndiQoS*. The division of the aggregate reservation among the individual publisher-subscriber flows is managed directly by the *IndiQoS* nodes.

Therefore, an *IndiQoS* node must perform the following steps to join the network: *i)* join the overlay network (in the current version, the Bamboo overlay); this step will define which other nodes of the *IndiQoS* network will be the direct peers of the joining node; *ii)* establish with the underlying data network a QoS reservation in the link to each of its peers for the aggregate *IndiQoS* traffic. It should be noticed that, in practice, the number of requests made by each node is small, because nodes of Bamboo have only $O(\log n)$ neighbors.

In each node of the *IndiQoS* system there is a *Resource Manager* that is aware of the available bandwidth and expected latency in each link of the overlay network. Hence, the Resource Manager must locally keep track of all the reservations already made by the publish-subscribe system that go through its node. As the paths are bound in the rendezvous node, the Resource Managers at the nodes along the path have to verify if the requested QoS can be satisfied. The Resource Manager gets the next hop information from the DHT and adds the QoS-related information to the advertisements and subscriptions in their path to the rendezvous node. This allows the rendezvous node to determine for a given path *i*) if there is enough bandwidth and *ii*) what is the expected latency. Hence, the rendezvous node contacts the subscriber to establish the reservations upstream to itself, while it makes the same thing toward the publisher. Note that this does not result in an implosion of messages toward the publisher, because reservations can be shared when they meet at a given node. It is up to the Resource Manager at each node to reserve the necessary resources at each link and distributes resources by publications and subscriptions as they arrive. The reader should notice that the Resource Manager makes reservations locally in each event router and only when paths are already defined.

Our approach to build *IndiQoS* has a number of advantages. First of all, Bamboo already tries to set neighbors of nodes in a way that reflects data network proximity. As a result, *IndiQoS* makes a rational use of data network resources. Additionally, most QoS reservations made by the *IndiQoS* applications will not pass to the data network, because *IndiQoS* nodes manage their own resources locally. This means that Bamboo only needs to set up QoS links when nodes enter or depart from the network or when resources of some link become exhausted.

5.4 Replication of the Rendezvous Points

The fundamental problem of QoS routing is to find a compromise between quantity of state update information and quality of routing decision. A perfect decision requires too many information (which may overwhelm the network with traffic), while too few information may result in very bad decisions or even in unfeasible paths. Our solution to this problem is the replication of the rendezvous nodes. The idea is to explore several routing alternatives, keeping QoS information local to the links. Since routing operations in a DHT like Bamboo need a small number of hops (lookups take $O(\log n)$ hops with a small constant) the additional traffic cost associated with each rendezvous node is small. Additionally, this has the advantage of increasing performance, scala-

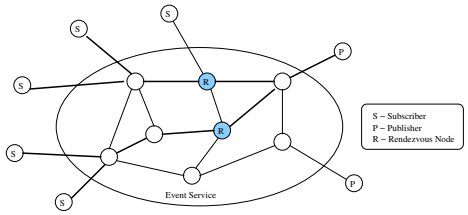


Figure 3: Using a replicated rendezvous.

bility and fault tolerance of the system. Assuming that T is a number that represents the type, the k rendezvous nodes are the nodes numerically closest to $hash(T), hash(T+1), hash(T+2), \ldots, hash(T+k-1)$. Publishers and subscribers send their messages to all the k rendezvous nodes. The difference to the single rendezvous node is that now the subscriber collects QoS information from all the possible paths. It chooses the best option and only after this moment the reservations take place, going upstream from the subscriber to the publisher through the preferred rendezvous. As with a single rendezvous node, tolerance to node failures or departures is transparently ensured by periodic refreshment of advertisements and subscriptions.

Figure 3 shows a distribution tree for one event type with two rendezvous replicas. A significant advantage of 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. As the number of rendezvous replicas grows, the system has more alternative 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 [18]. 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.

In our experiments, we compare *IndiQoS* with two different solutions described in Section 4.2. One of them keeps the QoS state information local to the links and floods the routing requests (this is called "Flooding Requests"). The second one follows a solution, which, in a sense, is the extreme opposite: it floods each link state update and excludes all links without enough bandwidth before computing the shortest path in terms of latency (this is called "Flooding Network State"). To ensure a fair comparison, all the three methods that we compare, *IndiQoS* and the two flooding alternatives, use all the nodes of the same overlay QoS network.

6.1 Benefits of 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 6.3 shows the increase in network utilization as the number of event-types increases. Percentage of bandwidth refers to the ratio between the sum of the bandwidth occupied in all the links of the QoS paths and the sum of the bandwidth of all the links of the network. With a single event type, all advertisements and subscriptions 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 under-utilized. 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 6.3 depicts the maximum achievable network utilization as a function of the number of replicas of the rendezvous node for a single event type. In the experiment we did not allow subscriptions to merge. Therefore, each subscriber has its own individual flow. As expected, it is observed an increase in the number of subscriptions that are satisfied as the number of replicas of the rendezvous increases. Interestingly, with a single event type it is possible to make better use

of resources than with multiple event types. This results from the fact that it is easier for a subscriber to reach the distribution tree of the single type (unlike the case of multiple different trees). Hence, this points to the conclusion that fewer types in the network lead to a better utilization of resources if several rendezvous nodes are provided.

Figure 6.3 depicts the average latency between the publisher and each subscriber 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.

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 forwarded to the different replicas of the rendezvous node. Figure 6.3 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.

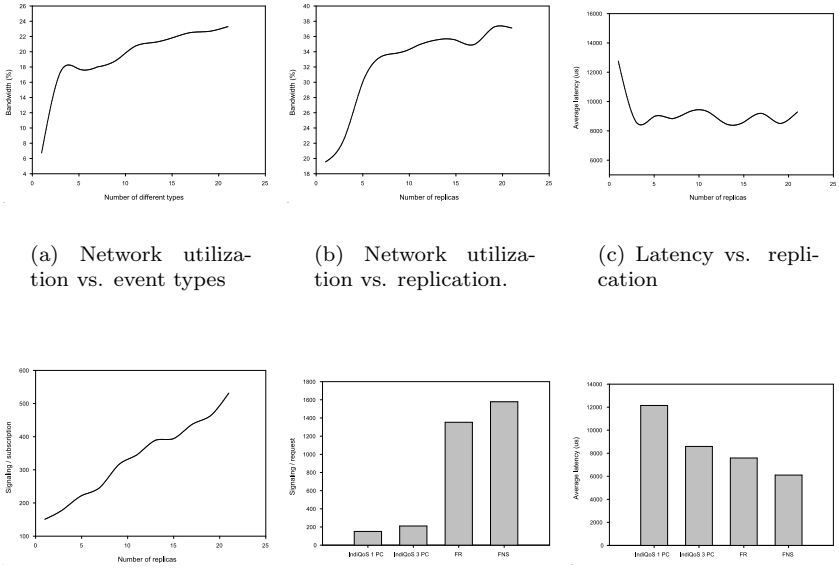
6.3 Comparison with other Strategies

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 [15]): 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 [17], 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.

Figures 6.3 and 6.3 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.



(a) Network utilization vs. event types

(b) Network utilization vs. replication.

(c) Latency vs. replication

(d) Signaling vs. replication

(e) Signaling

(f) Latency

Figure 4: Performance

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. To support such model, the *IndiQoS* includes a decentralized message-broker based on a DHT 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, Aug. 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*, Mar. 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, Aug. 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] OMG. *Event Service Specification*. Object Management Group, Mar. 2001.

[14] OMG. *Notification Service Specification*. Object Management Group, Aug. 2002.

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

[16] 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.

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

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

[19] 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.

[20] 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.

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

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

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