A Mobile Agent Infrastructure for QoS Negotiation of Adaptive Distributed Applications*

Roberto Speicys Cardoso and Fabio Kon

Department of Computer Science
University of São Paulo
{speicys,kon}@ime.usp.br
http://gsd.ime.usp.br

Abstract. QoS-aware distributed applications such as certain Multimedia and Ubiquitous Computing applications can benefit greatly from the provision of QoS guarantees from the underlying system and middleware infrastructure. They must avoid execution glitches that affect the user's perception of the application output.
Most research in QoS support for distributed systems focuses on three aspects of QoS management: admission control, resource reservation, and scheduling. However, in highly dynamic distributed environments, effective means for QoS negotiation and re-negotiation are also essential.
We believe that mobile agents, due to its inherent flexibility and agility, can play an important role in this scenario, specially during the application adaptation process. We designed a mobile-agent-based infrastructure that provides services such as resource monitoring, QoS brokering, and QoS enforcement. Furthermore, our infrastructure offers a powerful mechanism for QoS negotiation.
In this paper, we describe the architecture and prototype implementation of this infrastructure. First, we discuss the motivations and related works. We, then, present the architectural design and discuss implementation issues concerning the infrastructure prototype. Finally, we introduce a sample application called ReflectorAglet – a QoS-aware adaptive audio reflector – and present preliminary experimental results.

1 Motivation

Multimedia applications (such as video and audio streaming) and Ubiquitous Computing services are examples of distributed applications that have a clear need for quality of service guarantees. The success of this class of applications is tightly related to the user's level of satisfaction. These systems must avoid, at all costs, performance losses that might affect the user's perception of the application output. As a consequence, QoS-related services have become an important part of the infrastructure of distributed systems to enable the development of applications such as the ones described above.

* This work is supported by a grant from CNPq-Brasil, process #552028/02-9

Ongoing research in QoS management focuses mainly on problems related to admission control, resource reservation, and QoS scheduling. However, in highly dynamic distributed environments such as ubiquitous computing, active spaces[1], or mobile multimedia, these services are not sufficient to achieve an effective QoS management infrastructure. In such systems, issues associated with QoS *negotiation* become very important, therefore demanding a special treatment from the part of the underlying middleware.

QoS negotiation mechanisms are a fundamental requirement for highly dynamic distributed environments. On such environments, resource availability and end-user QoS needs vary greatly along time. Therefore, QoS-enabled applications may need to negotiate and re-negotiate their QoS requirements to keep the quality of its output. For instance, an Ubiquitous Computing Location Service must respond quickly to client requests to avoid unresponsiveness of the whole system. Its resource requirements may change according to the quantity of clients, the number of entities and the overall resource availability. The underlying system must provide mechanisms to allow the Location Service to negotiate its QoS requirements and to adapt to changing conditions.

The QoS negotiation process in a distributed system comprises the exchange of several messages among negotiating hosts. The traditional client/server approach for QoS negotiation is problematic in highly dynamic distributed environments for many reasons. A client/server based QoS negotiation would impose a heavy traffic over the network, impacting the distributed environment as a whole. Furthermore, because of the high number of possibilities for QoS configurations on a highly distributed environment, the client/server model could lead to a very inefficient system.

The inherent characteristics of mobile agents [LA99] make them a suitable technique to solve many problems related to QoS management in dynamic distributed systems, particularly the ones associated with QoS negotiation. Mobile agents are programs that can suspend their execution and migrate to other hosts in the network, resuming their execution from the point in which they were interrupted [KG99]. The architecture presented in this paper leverages the properties of mobile agents to provide efficient mechanisms for adaptation and QoS negotiation to applications. It also provides additional QoS-related services, namely QoS specification, QoS brokering, and QoS enforcement.

2 Related Work

Nahrstedt et al. [NXWL01] describe a powerful architecture for a QoS-aware middleware for Ubiquitous Computing. Their work focuses on four key aspects of such a system: QoS specification, QoS translation, QoS setup, and QoS adaptation. Our approach differs from theirs on a few central points. They propose a QoS compilation phase that generates a QoS profile based on the application QoS specification. The application then uses this QoS profile at run-time for QoS

[1] Active spaces are physical spaces augmented with digital devices that extend the user perception and interaction with the surrounding environment

setup. As a consequence, the application requires a new QoS profile whenever its QoS specification changes. We propose an interpreted-oriented approach, which enables the application to change its QoS specifications at run-time without service interruption. Furthermore, we extend their approach to QoS adaptation introducing a mobile-agent-based QoS negotiation mechanism. This enables the application not only to change its behavior according to the fluctuation of the QoS levels provided by the middleware, but also to search for another host on the distributed system that is able to satisfy its QoS requirements when the current host QoS level becomes unacceptable.

Hafid and Fischer proposed an agent architecture for QoS management [HF98]. Even though their motivations are similar to ours, their approach is somewhat different. Their work is focused mainly on the network aspects of QoS management. Their infrastructure rely on the existence of agents in routing points of the distributed environment to provide QoS management services to users and applications. Our work, on the contrary, faced the issues related to QoS guarantees to resources such as CPU and memory, which are host-related more than network-related. Furthermore, their architecture relies on agents running at routing points of the system, such as network routers, which may be hard to deploy.

Cavanaugh et al. [CWS+00] describe a QoS negotiation architecture for real-time distributed applications. In their work, a QoS Manager is responsible for maintaining the QoS levels agreed with the applications and a Resource Manager is responsible for controlling the environment resource usage. If a QoS violation is detected, the QoS Manager and the Resource Manager negotiate a repairing action that is executed without the knowledge of the applications. This approach suffers from two drawbacks. In highly dynamic distributed environments, the list of repairing actions might be very large. This would demand a long negotiation process between the QoS Manager and the Resource Manager, imposing unnecessary high traffic over the network. Besides, negotiation and adaptation are carried out without application acquiescence. This technique prevents the applications to adapt themselves to changes in the environment and to select the best adaptation method with regard to their particular purpose.

QuO [VZL+98,LSZB98] is a framework for the development of applications that adapt to different QoS levels provided by the underlying system. It adds a new layer to the CORBA architecture both on client and server sides. This layer is responsible for managing QoS contracts before sending requests among the distributed objects. QuO supports negotiation and adaptation, changing the application QoS level according to conditions detected in the environment. We extend this concept by providing middleware support to allow applications not only to change its QoS levels but also to maintain it by migrating to other hosts. The negotiation process is not only local as in QuO, but system-wide. Furthermore, QuO's Description Language (QDL) for QoS specification works much in the same way as CORBA IDL, generating code stubs at compilation time. Our interpreter-based solution for QoS specification does not require a compilation process, thus allowing modifications at run time.

3 Architecture

The overall objective of the architecture presented in this paper is to be a solid QoS management infrastructure for the development of QoS-aware adaptive distributed applications. To assess the effectiveness of the proposed architecture, we developed a sample application capable of evaluating all of the services provided by the infrastructure. In the following subsections we discuss the requirements for the architecture and its conceptual design.

3.1 Architectural Requirements

Our focus was to design a QoS management solution for highly dynamic distributed environments. This class of systems present many unique challenges. We designed our infrastructure based on three fundamental requirements, namely scalability, flexibility, and completeness. Our infrastructure must be scalable because new generation distributed systems, such as active spaces, are highly dynamic and distributed. On such systems, hundreds or even thousands of computers are connected by channels with unpredictable behavior. The infrastructure therefore must support a great number of connected devices and must be able to include all of them on the QoS management process.

The infrastructure must also be very flexible. Applications with different purposes and distinct adaptation mechanisms should be able to use the QoS management services in the most suitable way. The process of integrating applications to the infrastructure should not cause major architectural changes to neither of the systems involved. Previous work [VZL+98] and our experience on developing distributed systems suggests that to broaden the range of applications able to use the infrastructure and to increase its flexibility, it is necessary that environmental changes that may trigger adaptation actions be known to applications. Having full control of the adaptation process, applications can decide the best adaptation strategy to use. This decision is strictly related to the application purpose and the infrastructure cannot take the responsibility of choosing the best adaptation method on behalf of the application.

Finally, our infrastructure should provide a complete solution for QoS management. Our vision of a complete solution is similar to the ones presented in [VZL+98,NXWL01]: applications should be able to specify QoS requirements, monitor the provided QoS level, and adapt to QoS changes. The infrastructure, thus, must provide the means for QoS specification, enforcement, monitoring, and adaptation. Ideally, this solution should also provide mechanisms for end-to-end QoS enforcement, covering network-related QoS guarantees as well. This subject, however, is too broad and a lot of quality research [GP99,CC97] has already been made concerning this topic. We decided, therefore, to focus our work on host-related QoS issues only.

There are also additional requirements specific to each service provided by the infrastructure:

- QoS monitoring must not cause a heavy load on network resources. Networking is one of the most energy consuming resources of a mobile computer. Also, the monitoring process must not affect application performance by consuming resources intended to applications.
- QoS specification must be separate from application source code, allowing for QoS specification changes at run-time without service interruption.
- An admission control mechanism must be part of the QoS enforcement system to inform applications when their QoS level request cannot be met by the node, and to propose alternative QoS contracts for application review.
- The QoS adaptation system must provide means for QoS negotiation and re-negotiation. Applications supporting multiple QoS levels can use QoS negotiation to define the best QoS level available to work with. Besides, in case of environmental changes, the re-negotiation mechanism can be used to define a new QoS level which might be better or worse than the previous one.
- The QoS negotiation mechanism must be effective and flexible. Applications will use this system in multiple ways to negotiate QoS requirements with several nodes of the distributed environment. The QoS negotiation system, thus, cannot waste resources such as networking and processing time.

Security is also a concern whenever mobile agents are used. However, on this particular work, the security level provided by mobile agents is comparable to the one provided by related technologies such as RPC calls. This subject has been already widely studied, and an extensive discussion is out of the scope of this paper. More details concerning security with mobile agents can be found in [SV98].

To accomplish the goal of designing an infrastructure compliant with the requirements described above, we decided to use, whenever possible, systems already developed filling the gaps with novel work. We now describe the design of the infrastructure. In the next section we introduce the architecture overall design and in Sect. 4 we present the systems used and the original contributions developed during this research.

3.2 Architectural Design

A QoS management infrastructure for highly dynamic distributed systems faces many challenges that are not appropriately solved with traditional client/server techniques. Mobile agents can help overcome those issues in a number of ways with flexibility and agility benefits. Mobile agents [LA99] can encapsulate protocols, hiding from the application implementation issues of the infrastructure; may execute asynchronously, avoiding long periods of connected activity and thus reducing network load; and are composed of code and data, so applications can easily send agents throughout the distributed environment to perform complex actions on their behalf.

Our architecture relies on mobile agents to perform various tasks related to QoS negotiation and adaptation. The architectural design is better summarized

in Fig. 1. A central monitoring management node, which may be replicated, is responsible for concentrating monitoring data from networked hosts. Using the publisher/subscriber pattern, applications receive event notifications from this node with information about the status of entities spread throughout the distributed system (such as CPU utilization on host A, memory utilization on host B, etc.). The monitoring system allows applications to keep track of system-level QoS and provides hooks to trigger application adaptation. Changes in the QoS level might be caused by a host being unable to fulfill application resource needs or by an increase in service demand. In both cases, the system monitors resources to detect these conditions and notifies the interested parties so they can change their behavior to recover or enhance their QoS level.

If proper care is not taken, the central monitoring management node can become both a bottleneck and a single point of failure. In previous work, we have addressed these issues in a similar system, showing how this node could be replicated for fault-tolerance with little performance penalty [KYH+01]. In addition, previous experiments with that system showed that each monitoring management node could easily manage up to 100 networked hosts [MK02] and we are currently investigating new protocols for scaling this up to thousands of nodes. Thus, we believe that the relevant issues of fault tolerance and scalability can be resolved; however, they are out of the scope of this paper. We here focus on describing our new results in the area of QoS management and mobile agents, which were not present in our previous works.

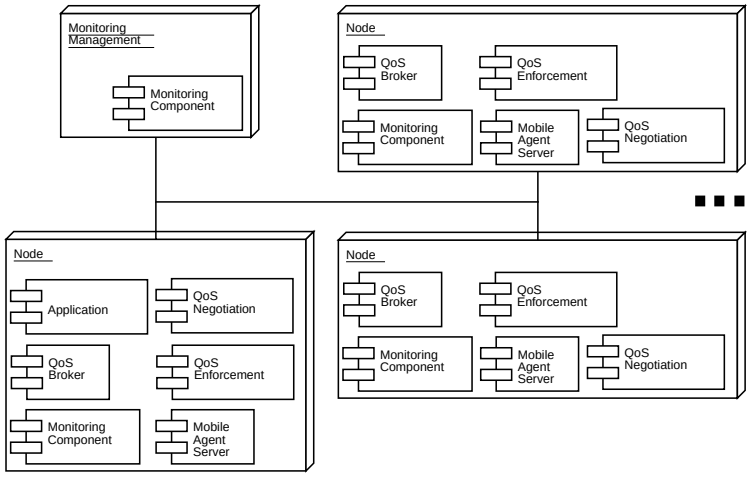

Fig. 1. Architectural design

Each node capable of hosting applications must be able to accept QoS reservation requests and mobile agents. They must also be able to offer QoS guarantees to hosted services. Note that not every node of the distributed system must

comply with these requirements. Mobile nodes with low and expensive computational power will not be used to host services. Hence, they might not need to offer QoS guarantees or receive mobile agents, although, if desirable, they may.

A QoS broker is also required on each host to mediate reservation requests between concurrent applications. It acts as a single point of negotiation for all programs that require QoS guarantees from that host. As such, it can perform admission control tasks refusing QoS contracts that it cannot honor. A deadlock prevention mechanism must also exist to avoid situations where a single application requests resources from every node in the system, denying to other applications the possibility of using these resources.

The QoS negotiation process is completely based on mobile agents. Whenever an application requests a QoS negotiation, a mobile agent reads the application QoS specification file and sends other agents across the network, according to the negotiation strategy selected by the application. These agents negotiate locally on each host and send the results back to the application, which, in turn, analyzes the results received and chooses the best adaptation strategy to use, according to its operational requirements.

To benefit fully from the services provided by the infrastructure, applications should also be designed using the mobile agent paradigm. This would increase their flexibility, since they would be able to adapt easily to changes in the environment using migration and cloning strategies. These two techniques combined with already existing application adaptation mechanisms can encourage the development of QoS-aware services enabled with multiple sophisticated forms of seamless adaptation. Legacy applications can also be wrapped into mobile agents and achieve the same benefits. Actually, during the course of our research we wrapped the JacORB[2] Naming Server and a multimedia reflector into mobile agents.

4 Prototype Implementation

Our approach to implement this architecture was to reuse, whenever possible, systems and tools already developed by our another research groups. Our focus was to identify possible shortcomings of existing technologies when applied to highly dynamic and distributed environments such as typical ubiquitous computing scenarios. The implementation of the proposed architecture required an extensive search for systems to support adaptation and QoS management for distributed systems. We evaluated these systems in terms of flexibility, ease of use, documentation, and stability.

We decided to use Aglets [LA99] as our mobile agent platform. It is an already established technology, with plenty of documentation available. Furthermore, unlike systems that use a modified virtual machine to obtain access to the memory stack, the aglets platform uses the native Java Virtual Machine, what increases its stability. It is also an open-source platform, allowing easier integration with other technologies. The monitoring mechanism we employed is part

[2] http://www.jacorb.org.

of the Framework for Dynamic Adaptation of Distributed Systems developed earlier in our research group. We discuss it with further details in Sect. 4.1.

To provide QoS enforcement features on hosting nodes, we used the Dynamic Soft Real Time Scheduler - DSRT[3] [NCN97]. This system offers soft real-time guarantees to applications without modifications to the original operating system kernel. More details regarding DSRT are presented in Sect. 4.2. The QoS definition language used for this prototype was the QoS Modeling Language - QML [FK98]. The reasons that led us to use this language along with specifics of the language are presented in Sect. 4.3.

4.1 Performance Monitoring

QoS-aware applications must have knowledge of the system-level QoS provided by the underlying infrastructure and also the application-level QoS provided to their clients. This kind of feedback helps applications to find out whether it is working according to what is expected. Applications use this information to identify environmental changes that might affect their performance, such as resource shortages or request bursts.

However, the monitoring process itself can be resource consuming. It is undesirable that it use a great amount of shared resources affecting application performance. Hence, it is important to find the correct balance between monitoring granularity and the imposed overhead. The Framework described in [SEK02] is a very flexible and highly configurable system whose focus is to ease the development of distributed applications providing efficient monitoring mechanisms and adaptation hooks.

A specific node is responsible for managing monitoring data. This node contains the entity repository, which defines the monitored nodes of the distributed system, the node resources subject to monitoring and the ranges of operation for each resource. Applications subscribe to simple events (e.g., notify me if CPU usage is greater than 60% for 15 sec.) or compound events (e.g., notify me if CPU usage is between 70% and 80% and memory usage is over 90% for 30 sec.). Monitored entities do not send their status periodically to the management node. Instead, information regarding monitored resources and subscribed events are sent to monitored hosts, responsible for evaluating this information and for contacting the central node only when an event of interest happens. After that, the central node sends the event to the subscribers, allowing applications to take adaptation actions.

This framework can be extended easily to monitor many different kinds of distributed resources. It is also very flexible, providing simple and yet powerful hooks for application to use. More importantly, applications have full control over the adaptation process, since the framework only notifies the application of changes on the environment, leaving the adaptation strategy to the application itself.

[3] http://cairo.cs.uiuc.edu/software/DSRT-2/dsrt-2.html

4.2 QoS Enforcement

To provide QoS guarantees to distributed applications using the infrastructure, a QoS enforcement system is needed. It is responsible for accepting application descriptions of resource requirements and managing the use of shared resources so that reserved resources are always available to applications.

To build our prototype we used DSRT, which supports processing time guarantees via its CPU Server and memory use guarantees via its Memory Server. It also features a Resource Broker to mediate QoS negotiations between the application and the system, supports advanced reservation scheduling, and provides QoS re-negotiation mechanisms. DSRT runs on top of several operating systems, including SunOS, IRIX, Linux, and Windows.

DSRT does not use a language for QoS definition. The application specifies its QoS requirements directly in its source code issuing calls to the DSRT APIs. It is an open-source system, what increases its flexibility allowing developers to change its behavior to better adapt to their necessities. In fact, we actually had to perform a few source code modifications so that DSRT could support Linux kernel 2.4, since it previously supported only kernel 2.2.

Even though we used DSRT to provide QoS enforcement services, our infrastructure was designed to be decoupled from the QoS enforcement system, and to support integration with other QoS systems. As we will show in the following sections, applications do not interact directly with the underlying QoS enforcement system, they communicate only with our infrastructure via high-level interfaces. The same application could work with a different QoS enforcement system, if the proper infrastructure support is developed. Even if the QoS enforcement features provided by a given system are different from the ones provided by DSRT, the application would only need to adjust its QoS specification file.

4.3 QoS Definition

Application QoS requirements may change across time. A program under demonstration usually requires less resources than when it is on a production environment, for example. It is important that QoS specifications be separate from the source code, otherwise any change in the requirements would require a recompilation of the whole application, which is undesired in terms of productivity and flexibility.

A DSRT shortcoming with regard to our architectural requirements concerns its QoS specification mechanism. The only mechanism for QoS specification provided by DSRT is API calls performed by the application source code. To overcome this weakness, we decided to use QML [FK98] as a high-level QoS specification language and used JFlex[4] to develop an interpreter that translates QML definitions into DSRT API calls. An example of a simple QML specification for a DSRT contract is shown in Fig. 2. It creates a type of contract called `CPU_RT_PCPT` composed of two dimensions: processing `period` and

[4] http://jflex.de

peakProcessing_per_period, used to define at most how much processor time the application will require in each period. The decreasing qualifier means that lower values are stronger guarantees while increasing means that higher values are stronger. After that, an instance of this contract, called idealProcessing, is created. Finally, this instance is associated with the mpeg_player application.

```
type CPU_RT_PCPT = contract {
    period: decreasing numeric usec;
    peakProcessing_per_period: increasing numeric;
};
idealProcessing = CPU_RT_PCPT contract {
    period = 40;
    peakProcessing_per_period = 30;
};

idealProfile for mpeg_player = profile {
    require idealProcessing;
};
```

Fig. 2. Example of a QML definition for a DSRT contract

The QML Parser is called at run time by the negotiation agents to interpret the specification file and to use those definitions during negotiation. This file can be modified during application execution. The following negotiation process will then use the new definitions provided.

The choice of QML was a hard decision. QuO's Definition Language [LSZB98] also provides rich means for QoS specification. However, QDL is very tied to QuO's mechanisms for QoS management. It would be difficult to use QDL to specify a generic QoS contract, since its keywords are strictly related to QuO's internals such as callback functions. We also considered using XML for QoS contract specification. As a general purpose language, XML would allow us to define contracts irrespective of any specific QoS enforcement system. Parsing an XML file would also be easier since it is widespread and many languages have libraries that support XML processing. Unfortunately, the generality of XML was actually a drawback. The text files we created were excessively verbose and hard to understand by humans comparing to similar files in QML. XML specifications would be difficult to write by developers, hard to read (e.g. for debugging purposes), and demanding to maintain.

4.4 QoS Negotiation

Applications working in highly dynamic distributed environments must adapt gracefully to constant availability changes of system resources to continuously deliver services with acceptable QoS levels to their clients. Peaks of utilization, failures of system components, and resource sharing can significantly affect the

performance of Multimedia and Ubiquitous Computing applications. QoS guarantees are fundamental for those applications, since their outputs are strictly related to human senses and users can easily perceive glitches in their operation.

This class of applications require effective means for QoS negotiation. Hosts might not honor QoS contracts due to unpredictable events and applications may have to re-negotiate their QoS contract to adapt to this situation. Peaks of utilization may cause applications to adapt by increasing their QoS requirements, using QoS negotiation mechanisms. Applications capable of working with multiple QoS levels must use some form of QoS negotiation to define on which level of QoS they will operate.

We designed an efficient mechanism for QoS negotiation using mobile agents. The whole negotiation process is described below:

1. The application subscribes to resource usage events with the monitoring system.
2. The monitoring system notifies the application when an event of interest has occurred.
3. The application requests a QoS negotiation to the Negotiator agent.
4. The Negotiator agent reads the application QoS specification file, defines the negotiation strategy to use, and creates ProxyNegotiator agents.
5. ProxyNegotiator agents travel to remote hosts to carry out the negotiation process locally. When the negotiation is over, they send back the results to the Negotiator agent.
6. The Negotiator agent collects the results and passes them to the application.
7. Finally, the application receives the QoS offers and determines which one is the best according to its specific criteria.

Applications can receive negotiation results in two different ways. They can wait for results of all hosts visited during the negotiation process and choose the best offer among all hosts. Or they can wait for an application-defined negotiation timeout, request the results obtained by the Negotiation system up to that moment, and choose the best offer among the ones received until then.

This mechanism frees the application from the complicated process of QoS negotiation. The tasks of searching for a suitable host, negotiating the best contract according to the application multi-level QoS specification, and collecting the results are solely the responsibility of the Negotiation system. The application developer must only specify QoS requirements, ask for a negotiation whenever an adaptation is required, and select the best offer among the ones received.

Figure 3 shows the messages exchanged during a negotiation. The monitoring system notifies the application that an adaptation should take place. The application then creates an object containing its adaptation strategy (the strategy will be responsible for receiving negotiation results and selecting the best offer according to application functional requirements). The application then asks the Negotiator agent to start a QoS negotiation. This system creates agents and dispatches them to the network to carry on the negotiation process on remote hosts. When this process is over, the agents send back the results to the Negotiator agent. It compiles the received results and passes to the application a

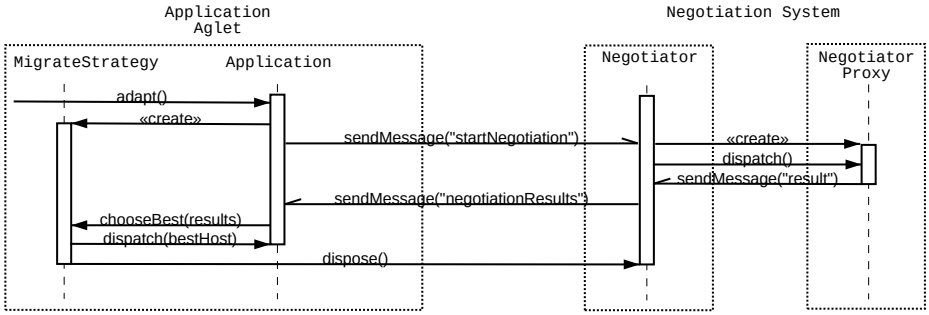

Fig. 3. Messages exchanged during a negotiation

list of available offers. The application adaptation strategy analyzes the received results, selects the best option and and takes the proper actions.

The QoS negotiation mechanism also supports application-defined roaming strategies. Currently, Negotiator Proxies can search for the best QoS offer using two different strategies, described in Fig. 4. With the linear strategy, a single ProxyNegotiator will visit each host of the environment, sending results back to the Negotiator agent. The parallel strategy, on the other hand, will send a different ProxyNegotiator to each host of the distributed system, and each agent will return its result to the Negotiator agent. One could also imagine a peer-to-peer strategy, in which a ProxyNegotiator would leave its original host knowing only its first destination. According to negotiation results in each node, the agent could select its next hop, and so on.

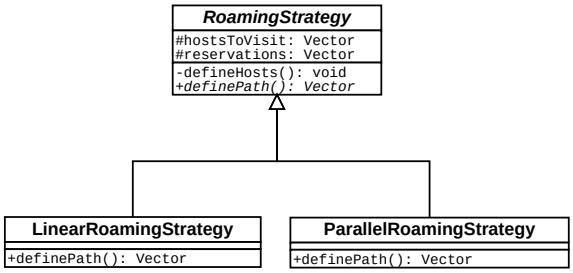

Fig. 4. Linear and parallel QoS negotiation strategies

The parallel strategy provides a better response time for negotiation results, since the process is carried on simultaneously on all hosts of the distributed system. However, this strategy also causes greater network traffic, since an agent is sent to each host. Conversely, the linear strategy generates less network traffic

(a single agent is sent), but also takes longer to provide negotiation results (the negotiation is performed sequentially).

This mechanism could be extended to support multiple application QoS negotiation. The existing design requires that each application trigger a separate negotiation process. If two or more applications on the same host request a QoS negotiation, this would trigger multiple simultaneous negotiation processes, which is unnecessary. A single agent could negotiate QoS requirements on behalf of multiple applications, thus reducing network load and increasing efficiency. Furthermore, negotiation results for an application could eliminate the re-negotiation necessity of other applications.

4.5 QoS Brokering

The architecture allows for application-initiated QoS negotiation. As such, several applications may negotiate with the same host at the same time. To avoid concurrency reservation issues, each node must have a single process responsible for coordinating the QoS negotiation for multiple applications that may request resources from that node.

Particularly, this coordinator process must avoid starvation of resources on the distributed system while searching for the best offer. An application may negotiate QoS contracts with many different hosts of the environment, but will actually request a reservation from a single node. The resources negotiated but not used by the application must be freed so that other applications may use them during their QoS negotiation process.

DSRT's original broker design [KN00] does not distinguish negotiation requests from reservation requests. Hence, when an application demands reservation of available resources during its negotiation phase, the broker actually executes the reservation. If the application does not plan to use the resources negotiated, it must explicitly contact the broker of each host and release the resources obtained during the negotiation process. This would cause an unnecessary network overhead, affecting the infrastructure performance.

To avoid this issue, we have designed a CORBA QoS Broker that applies the leasing pattern [KJ04] for reservation tickets. Whenever a QoS reservation request is accepted, a ticket is created with a predefined validation timeout. If the application will not use the reservation ticket received (because another host made a better offer e.g.), no further action is required, since the resources will be released as soon as the ticket expires, typically in a few seconds. If the application tries to use an expired reservation ticket for a resource reservation, the QoS Broker denies the request and the application must apply for a new ticket.

We designed the QoS Broker as a CORBA object to ease the interaction between negotiation agents and broker. Agents can easily locate the QoS Broker in each host using the CORBA Naming Service. Agents interact locally with a QoS Broker via broker method calls.

4.6 Adaptation Support

Self adaptive software monitors its behavior and changes it whenever required by external conditions or when performance and functionality improvements are possible. Adaptive software is specially useful in highly dynamic distributed environments, since changing availability of resources demand a myriad of adaptation techniques from applications.

To increase application flexibility and adaptability on such environments, they should also use the mobile agents paradigm. Mobile agents reduce network load, overcome network latency, can be easily integrated to heterogeneous systems and are fault-tolerant [LA99]. These features can enhance application adaptation strategies allowing for functionality and performance benefits.

With our infrastructure, applications developed using the mobile agents paradigm can profit from two adaptation techniques, as shown in Fig. 5.

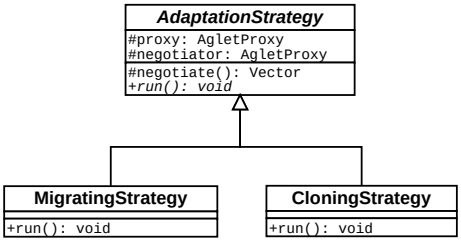

Fig. 5. Migrating and cloning adaptation strategies

As explained on Sect. 4.4, the outcome of a QoS negotiation, according to the application point of view, is a list with QoS offers received from hosts of the environment. After reviewing those offers, an application may conclude that the best adaptation alternative is to migrate to one of the hosts and accept its QoS offer. Using the migrating strategy, an application can easily change its execution to another host on the distributed system. This approach may help applications to improve its QoS contracts or simply to continue its execution in case of a host deactivation.

Similarly, using the cloning strategy an application can create a copy of itself and send it to another host when suitable. This technique enables applications to adapt gracefully to service usage bursts and valleys, consuming and freeing resources according to client requests.

In this work we do not discuss issues that arises when an application changes from host, such as client reference updates. This is a much broader research field and numerous qualified works exist concerning this subject. Moreover, we believe that even though the middleware may offer mechanisms to help applications to overcome these difficulties, each application may require a different particular solution.

5 Experimental Results

To assess the flexibility and usefulness of our proposed architecture, we developed a basic sample application. The application purpose was to evaluate all services provided by the infrastructure, namely QoS definition, QoS enforcement, QoS negotiation, admission control, and dynamic adaptation. In this section we introduce our sample application, discuss implementation details and present some preliminary experimental results.

5.1 Sample Application: ReflectorAglet

Multimedia broadcast over the Internet is often a processing and networking consuming operation. Servers must capture audio and video signals, convert them to a suitable format using compression algorithms, and send large amounts of data to their clients. When the number of clients increases, this approach becomes not efficient.

The typical architecture for scalable real-time multimedia streaming over the Internet is shown at the left-hand side of Fig. 6. It uses multimedia reflectors to decrease processing and network loads generated by multimedia streams [KCN01].

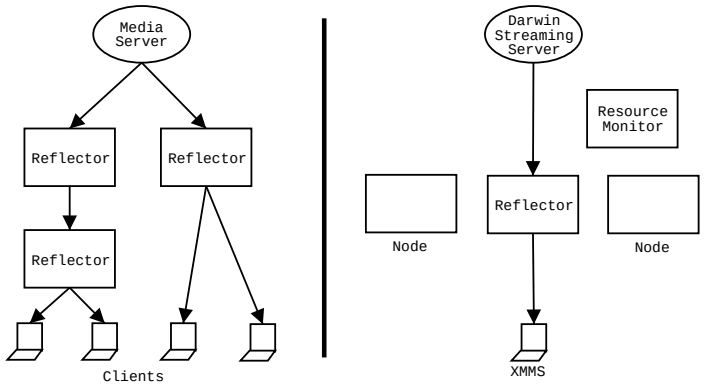

Fig. 6. Typical Internet multimedia broadcast architecture and our testing environment

Clients connect to multimedia reflectors instead of connecting directly to media servers. Reflectors are light-weight processes in terms of CPU usage, as they connect to media servers and simply forward the traffic received to other reflectors or application clients. This architecture unloads media servers and improves overall performance.

We developed a multimedia reflector, called ReflectorAglet, using the mobile agent paradigm. This aglet specifies its QoS requirements, detects processing

shortages, and uses the QoS Negotiation system to find a host capable of fulfilling its QoS requirements. The ReflectorAglet and the infrastructure source code can be found at `http://gsd.ime.usp.br/software/QoSNegotiation`. Our testing environment is presented at the right-hand side of Fig. 6.

For testing purposes, we used Apple's Darwin Streaming Server [5] to stream MPEG Layer 3 (MP3) files. Reflectors were responsible for connecting to the Darwin Streaming Server and forwarding the stream to XMMS [6] clients, in charge of playing the audio stream. The ReflectorAglet monitored its host processing usage. When high CPU utilization was detected, the reflector requested a negotiation to find another host able to accept its QoS requirements. The reflector then migrated to this host using its new QoS contract. We used a process-consuming application to cause high CPU utilization at the reflector host. All hosts used for our tests had AMD Athlon XP 1700 processors and 512MB of RAM memory. We used a QoS requirement file including the QML specification for three different QoS levels in which our test application could operate.

Our tests consisted in evaluating the negotiation process scalability when it included up to four hosts. The focus was to estimate the increase in execution time of a typical QoS negotiation in our infrastructure. We compared the performance of both negotiation roaming strategies: linear and parallel. The results are presented in Fig. 7.

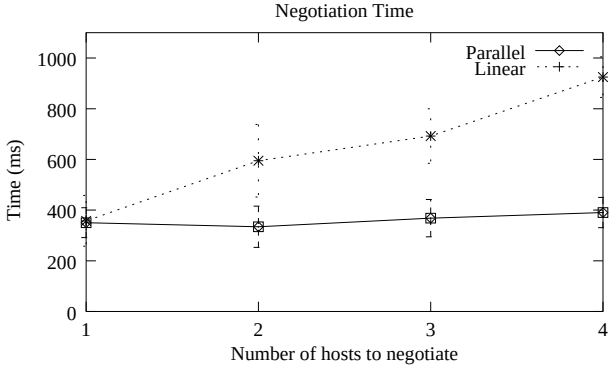

Fig. 7. Testing results

Each test was performed five times; each point in the graph represents the arithmetic mean of five runs of the experiment and the vertical error bars represent the standard deviation of the mean. We first measured the time of an aglet migration without QoS negotiation (which corresponded to negotiating with zero hosts), which was equal to 8.6 ms. After that, we measured the time overhead

[5] http://developer.apple.com/darwin/projects/streaming
[6] http://www.xmms.org/

caused by a negotiation with one to four hosts. We considered the time elapsed between the instant in which the application receives the event notification (that triggers the negotiation) and the instant in which it departs from its original host to its new host (after the result of the negotiation is known and a final decision is made).

The negotiation time for the parallel strategy remained almost the same on every run of the experiment, since the negotiation agent clones visited all hosts concurrently. The time overhead of including another host to the process is minimal but, on the other hand, the network load increase is noticeable.

Conversely, the overall negotiation time for the linear strategy increased linearly. In a linear negotiation, a single agent must travel to all target hosts. As a consequence, for each new node added to the negotiation process, the total negotiation time also increased accordingly. Even though this strategy proved slower in most cases, it could still be useful if modified to interrupt the negotiation process when an optimal (or near-optimal) QoS contract is found. This would shorten the negotiation process and unload the network when compared to the parallel strategy.

6 Conclusions

In this article we presented a mobile-agent-based infrastructure for QoS Negotiation, which is part of an entire QoS management architecture. Its goal is to be an effective mechanism for QoS negotiation of adaptive distributed applications, unloading them from the complexity of QoS negotiations on highly dynamic distributed environments. As a consequence of using multi-platform technologies such as DSRT and Java, it is also portable and deployable in heterogeneous environments.

Our first results are promising. The QoS negotiation process scaled up well when we considered one to four hosts. We are now carrying out more experiments to evaluate the QoS negotiation system performance in larger environments as well as the whole infrastructure. Future work includes further evaluation of the infrastructure using other distributed applications, the development of more sophisticated QoS negotiation algorithms and an evaluation of the infrastructure usage, defining in which scenarios it is worth to use this infrastructure and in which scenarios a simpler solution would suffice.

We think that mobile agents present many potential advantages not yet extensively explored. Even though the mobile agent concept was introduced nearly ten years ago, the related technologies are not yet widespread among distributed systems developers. We should note that other now well-established technologies such as object-orientation took over twenty years to become mainstream. We believe that more research and experience with mobile agents are necessary before this technology can become widely used by a large number of system developers.

References

[CC97] A. T. Campbell and G. Coulson. QoS Adaptive Transports: Delivering Scalable Media to the Desktop. *IEEE Network*, 11(2):18–27, March 1997.

[CWS+00] C. D. Cavanaugh, L. R. Welch, B. A. Shirazi, E. Huh, and S. Anwar. Quality of Service Negotiation for Distributed, Dynamic Real-time Systems. In *Workshop on Bio-Inspired Solutions to Parallel Processing Problems (BioSP3) at IDPDS Workshops*, pages 757–765. Springer, 2000.

[FK98] S. Frølund and J. Koistinen. Quality of service specification in distributed object systems design. *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, April 1998.

[GP99] R. Guérin and V. Peris. Quality-of-service in Packet Networks: Basic Mechanisms and Directions. *Computer Networks*, 31(3):169–179, February 1999.

[HF98] A. Hafid and S. Fischer. A multi-agent architecture for Cooperative QoS Management. In *Management of Multimedia Networks and Services*, pages 41–54. Chapman & Hall, 1998.

[KCN01] F. Kon, R. Campbell, and K. Nahrstedt. Using Dynamic Configuration to Manage a Scalable Multimedia Distribution System. In *Computer Communication Journal*, volume 24, pages 105–123, 2001.

[KG99] D. Kotz and R. S. Gray. Mobile agents and the future of the Internet. *ACM Operating Systems Review*, pages 7–13, 1999.

[KJ04] M. Kircher and P. Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley, 2004.

[KN00] K. Kim and K. Nahrstedt. A Resource Broker Model with Integrated Reservation Scheme. In *Proceedings of IEEE ICME'2000*, 2000.

[KYH+01] F. Kon, T. Yamane, C. Hess, R. Campbell, and M. D. Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proc. 6th USENIX COOTS*, February 2001.

[LA99] D. B. Lange and M. Ashima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[LSZB98] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Proceedings of ISORC '98*, April 1998.

[MK02] J. R. Marques and F. Kon. Gerenciamento de Recursos Distribuídos em Sistemas de Grande Escala. In *Proceedings of the 20th Brazilian Symposium on Computer Networks*, pages 800–813, May 2002.

[NCN97] K. Nahrstedt, H. Chu, and S. Narayan. QoS-Aware resource management for distributed multimedia applications. *Journal on High-Speed Networking*, December 1997.

[NXWL01] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-Aware Middleware for Ubiquitous Computing. In *IEEE Communications Magazine*, volume 39, Issue 11, pages 140–148, November 2001.

[SEK02] F. J. S. Silva, M. Endler, and F. Kon. Dynamic adaptation of distributed systems. *12th ECOOP Workshop for PhD Students in OO Systems*, June 2002.

[SV98] Springer-Verlag, editor. *Mobile Agents and Security*, number 1419 in Lecture Notes in Computer Science, 1998.

[VZL+98] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. In *Proceedings of Middleware'98*, September 1998.