Using Dynamic Configuration to Manage A Scalable Multimedia Distribution System ⋆

Fabio Kon, Roy H. Campbell, Klara Nahrstedt [1]

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue, Urbana, IL 61801-2987 USA
http://choices.cs.uiuc.edu/2K

Abstract

Multimedia applications and interfaces will change radically the way computer systems will look like in the coming years. Radio and TV broadcasting will assume a digital format and their distribution networks will be integrated to the Internet. Existing hardware and software infrastructures, however, are unable to provide all the scalability, flexibility, and quality of service that these applications require.

We present a framework for building scalable and flexible multimedia distribution systems that greatly improves the possibilities for the provision of quality of service in large-scale networks. We show how to use architectural-awareness, mobile agents, and a CORBA-based framework to support dynamic (re)configuration, efficient code distribution, and fault-tolerance. This approach can be applied not only for multimedia distribution, but also for any QoS-sensitive distributed application.

Key words: multimedia distribution, dynamic configuration, middleware, CORBA, QoS-aware resource management

1 Introduction

Multimedia interfaces will play a fundamental role in human-computer interaction in next generation computer systems. Applications will gradually abandon dull interfaces based on text and static graphics and move towards

⋆ This research is supported by the National Science Foundation, grants CCR 96-23867, 98-70736,99-70139, and EIA 99-72884EQ. Fabio Kon is supported in part by a grant from CAPES, Brazil, proc. 1405/95-2.

[1] E-mail: {f-kon,roy,klara}@cs.uiuc.edu

a more interesting look based on audio, video, and animations. Within one decade, there will probably be no distinction among the telephone, radio, TV, and data networks; and in future decades, no distinction between computer and dedicated radio and TV receivers. Everything will be integrated on an extension of today's Internet.

The Internet as we know today, however, is not suited for the distribution of high-quality multimedia to a large number of clients. A number of changes in the hardware and software that currently support the Internet will have to occur. We need systems to better control large numbers of multimedia flows, with support for Quality of Service (QoS) provision.

In the past ten years, several research groups have worked on architectures for distributing digital multimedia content through the Internet and intranet infrastructures. In particular, researchers working on the technologies related to the MBone [1,2] and to adaptive audiovisual streaming protocols [3,4] provided significant contributions. We noticed, however, that existing MBone and unicast solutions do not provide the degree of control, flexibility, and the possibilities for QoS management that the next generation applications require. In this article, we present an architecture for scalable multimedia distribution that meets those requirements. The architecture combines modern technologies from different Computer Science fields and extends them when necessary.

We applied the ideas presented in this article to build an object-oriented multimedia distribution system in C++ [5] and demonstrated that it is possible to use the existing Internet to distribute low and medium bandwidth multimedia to thousands of simultaneous users. Our early experiments, though, pointed out difficulties in managing such a large-scale system and keeping it available with an acceptable QoS. It showed the necessity for a better support for dynamic (re)configuration, code distribution, and provision of fault-tolerance.

This article describes our most recent achievement in this area, i.e., an integrated software architecture addressing the problems we encountered in the past. Our solution is based on an extensible Reflector system, mobile agents, and a CORBA framework for dynamic configuration and reconfiguration.

Section 2 presents an overview of the basic concepts involved in scalable multimedia distribution. Section 3 describes our early design, discussing its limitations. Section 4 describes our new, enhanced architecture with support for dynamic configuration and explains the synergistic relationships between dynamic configuration and QoS. Section 5 presents a concrete implementation of our architecture and Section 6 discusses performance evaluation. Finally, Section 7 discusses related work and Section 8 presents our conclusions and future work.

2 Basic Concepts

Before going into a more detailed description of our architecture, we first present a brief overview of the most important concepts related to our approach to scalable multimedia distribution.

(1) **Reflector**. It is the key element of our distribution system. It acts as a relay, receiving input data packets from a list of trusted sources and forwarding these packets to other Reflectors or to programs executed by end-users. The distribution system is composed of a network of Reflectors that collaborate with each other to distribute the multimedia data over local-area, metropolitan-area, and wide-area networks.

(2) **Reflector administrator**. A privileged user that is responsible for managing a Reflector network. Different portions of the network may be managed by different administrators, making large systems more manageable and helping to deal with different security domains.

(3) **Dynamic (re)configuration**. As computer environments become more dynamic, a major requirement for next generation computer systems is the ability to customize the system to the characteristics of the environment in which it executes. This property is known as *dynamic configuration*. Also of extreme importance is the ability of a system to change, on-the-fly, its internal components and configuration parameters to adapt to changes in the environment. This is called *dynamic reconfiguration*[2].

(4) **Quality of service (QoS)**. Multimedia applications have stringent requirements with respect to computational resources such as memory, CPU, and network [6]. If these requirements are not met, the quality of the multimedia service is degraded. This quality can be measured according to different metrics. A videoconference service, for example, can be evaluated according to the size and the number of colors of the video frames, the audio sampling rate, the video frame rate and jitter, the synchronization of audio and video, the delay from the sender to the receiver, the number of frames that are lost or defective, and so on.

(5) **CORBA**. The OMG *Common Object Request Broker Architecture* is an architecture for distributed object communication that is both language-independent and platform-independent. It is based on a standard interface definition language (IDL) and includes standard definitions for interoperable distributed communication [7]. It also defines standard interfaces for services such as naming, trading, real-time, persistence, and transactions [8].

[2] For simplicity, the term *dynamic configuration* may be used to denote both *dynamic configuration* and *dynamic reconfiguration*.

3 Scalable Distribution

Using traditional, centralized video-on-demand servers, it is possible to stream video to hundreds of simultaneous clients [3]. Exploiting IP-Multicast [9], this number increases to several thousands or, maybe, a few millions. However, when using IP-Multicast to send video to a large number of clients, one has little control over the transmission. It becomes difficult to provide support for a number of desired features such as QoS, security, accounting, and reliability. To address this problem, we developed a scalable multimedia distribution framework whose architecture is described in this section.

3.1 The Reflector

As noted above, the Reflector is the key element of the distribution framework. It is a user-level program that, at the application level, performs activities similar to the ones performed by network routers at the hardware level (which is where IP-Multicast is implemented). Since it is implemented in software, not hard-wired into the router, the Reflector is more flexible, easy to deploy and evolve, and can be dynamically customized to users, applications, and environments. Reflector data packets are encoded with RTP, a user-level protocol for real-time applications [10] defined by the Internet Engineering Task Force (IETF). RTP packets can be transmitted over different kinds of low-level transport protocols such as TCP, UDP, and IP-Multicast.

The Reflector combines data packets into logical groups called *channels*. It examines a 2-byte *channel_ID* field in the packet header and forwards the packet to the clients and Reflectors that subscribed to that particular channel. Each channel can contain multi-track data for different applications such as video and audio conferencing, live high-bandwidth MPEG broadcasting with TV quality, stored low-bandwidth H.263 video and GSM audio, HTML news, or current stock values. Clients and administration programs can connect to a Reflector in order to get information about available channels, Reflector load, bandwidth utilization, historical statistics, etc.

The Reflector network topology is determined by each Reflector's configuration. This information specifies input and output connections, access privileges, maximum allowed number of users, etc. The information is stored in a database controlled by the Reflector administrator. Figure 1 depicts a generic Reflector network that distributes two video streams in different channels. In this Figure, two capture stations send their video streams to "master" Reflectors; the streams may traverse several "intermediate" Reflectors until they reach "public" Reflectors to which end-user clients can connect and receive

the video streams. All the Reflectors in the Figure are initiated with exactly the same code. As explained in Section 4.1, the system then customizes each Reflector according to their individual requirements.

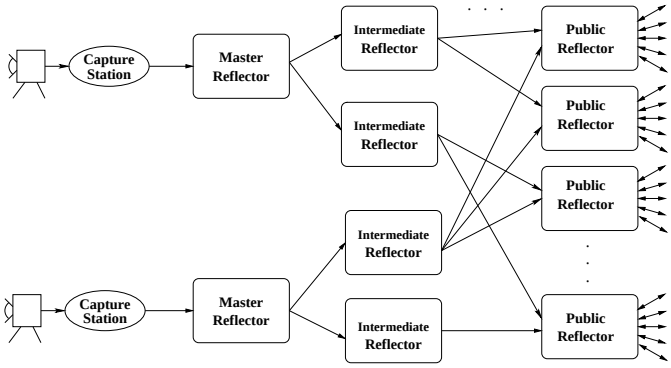


Fig. 1. A Reflector network distributing two video streams

To provide fault-tolerance Reflectors can accept *redundant inputs* for the same channel from several sources. It assumes that one of them is the primary source and ignores the data coming from other redundant sources. If an error in the connection with the primary source is detected or if it remains silent for a pre-defined period of time, the Reflector automatically switches to the next source in its redundancy list.

Software routing at the Reflectors may introduce extra latency and jitter. Typically, latency will only be a problem for conferencing applications and only if a long chain of Reflectors is used. Jitter can be minimized by reserving CPU and memory as described in Section 4.1.

3.2 Data Distribution Protocols

In order to support different types of inter-Reflector *data distribution protocols* transparently, the Reflector framework encapsulates the concept of a network connection into an abstract C++ class named *Connection* that defines the basic interface for all types of network connections used by the Reflector. This abstract class implements some of its own methods, but the majority of the connection-related code is implemented by its subclasses: *TCPConnection*, *UDPConnection*, *MulticastConnection*, etc. Reflector control information and meta-data is sent through reliable TCP connections. Multimedia data may be sent through unreliable connections in order to achieve a higher throughput. Also, different channels may use different connection types; for example, in a single Reflector a video channel may use UDP while a text channel with news articles uses a TCP connection.

Figure 2 depicts a concrete example of a highly-heterogeneous Reflector network. In this example, the network distributes two audiovisual streams. The

first comes from a mobile camera, mounted on a helicopter, that sends its stream to a "master" Reflector over a wireless link. This kind of link presents a high rate of packet loss not related to congestion, which makes protocols like TCP perform poorly. Thus, it is desirable to use a protocol optimized for this kind of situation like, for example, WTCP [11]. The second stream is sent to its "master" Reflector through a dedicated ISDN line. To optimize the bandwidth, one can use UDP as the communication protocol since its overhead is lower than that of TCP and the link offers a low loss rate. Reflector C sends its streams to Reflector D over the public Internet through a transatlantic satellite link. Even though this is a high-bandwidth link, its loss rate may be high, so it is more appropriate to use TCP. Reflectors A and D introduce the video streams into two distant points of the global MBone.

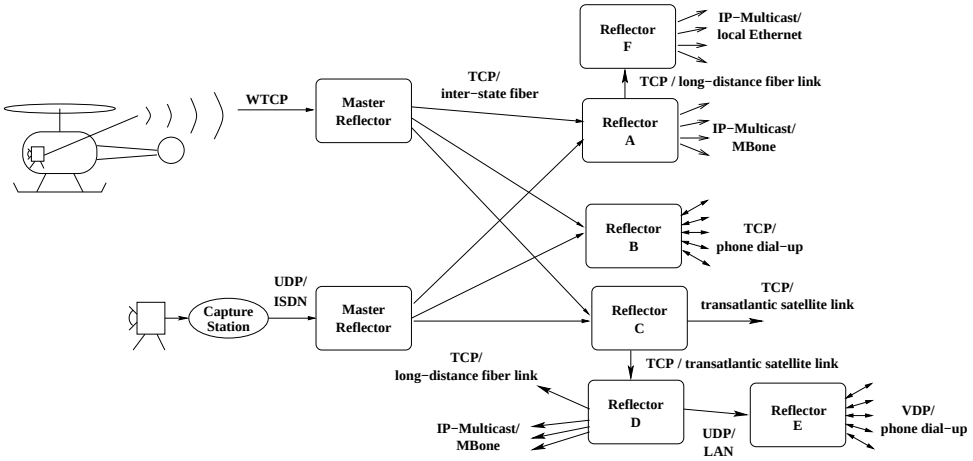


Fig. 2. A heterogeneous Reflector network

Selecting the appropriate protocol for each situation, the Reflector administrators improve the quality of service offered to the end-users, optimizing the utilization of the available network bandwidth and minimizing packet losses.

Most of the Reflector's code deals with objects of type *Connection* and is not aware of the *Connection*'s underlying implementation. The actual connection type is specified when the connection is created and does not need to be known after that. This approach allows programmers to plug in customized *Connection* subclasses by providing their own implementation of the *Open*, *Close*, *Bind*, *Connect*, *Send*, and *Receive* methods. In this manner, it is possible to incorporate, into the Reflector, *Connection* subclasses that implement different transport protocols (such as the VDP QoS-aware adaptive protocol for the Internet [3] and the xbind QoS-aware reservation protocol for ATM networks [12]). Developers also use this mechanism to implement *Connection* subclasses that perform various operations on the data such as encryption, transcoding, mixing, and downsampling. Finally, one can create new *Connection* types by composing existing ones. For example, one can create a *CryptoMulticast* connection type – that encrypts the data and sends it out using Multicast – by composing a *Crypto* connection with a *Multicast* connection.

3.3 Experience and Lessons Learned

This technology was utilized in the live broadcast of NASA's JPL Pathfinder mission [13]. During this broadcast – which lasted for several months – more than one million live video sessions were delivered to dozens of different countries across the globe by a network of more than 30 Reflectors spread across five continents. The Reflectors ran in five different operating systems (Solaris, Linux, Irix, FreeBSD, and Windows) and transmitted their streams over different kinds of network links. End-users could watch the video stream simply by pointing their web browsers to selected locations, causing their browsers to download a Java applet containing the video client. The applet connected to the Reflector automatically, received the video stream, decoded it, and displayed it to the user in real-time.

During this broadcast, we experienced three major problems:

(1) As the code had not been tested on such a large scale and on so many different platforms, we found many programming errors both in the Reflector code and in the client applet. Surprisingly, fixing the error was, sometimes, easier than updating the code in the dozens of machines that formed the distributed system. The same problem occurred when a new version of the Reflector, with added functionality, was released. System administrators had to manually connect to dozens of machines, upload the new code, shutdown the old version, and start the new one.
(2) Often, we had to reconfigure the reflector network by dynamically changing the distribution topology, or by setting new values to the reflector configuration parameters (e.g., maximum number of users, number of multimedia channels). The configuration information for the reflectors was stored in a centralized location. After updating this centralized database, we had to connect to each of the reflectors and instruct them to download their updated configuration. This process was tiresome and error-prone.
(3) The only mechanism the Reflector provided to support fault-tolerance was the redundant inputs described in Section 3.1. But this mechanism leads to a large waste of bandwidth. The redundant streams are always transmitted even though they are seldom used.

With this experience, we learned that a user-level Reflector system is, indeed, a powerful tool for managing large-scale multimedia distribution. It gives Reflector administrators a tight control over the distribution, allowing for a better control of QoS. It achieves that through the definition of the distribution topology, the selection of appropriate communication protocols, and the possibilities for limiting the number of clients according to the available resources.

We also learned, however, that it was important to provide better mechanisms

for distributed code updates, dynamic reconfiguration, and fault-tolerance. In the next section we describe a novel architecture for multimedia distribution that addresses the problems encountered in the previous approaches, leading to a more efficient, manageable, and reliable system.

4 Architecture

With the exponential growth in the number of workstations, laptop computers, and PDAs connected to the Internet, and with the extra levels of mobility and dynamism that these devices bring, it becomes increasingly important for multimedia systems to be able to cope with dynamic changes in the environment. They must react to variations in resource availability, dynamically adapting their algorithms, updating parts of the system, and replacing software components when needed.

The Reflector is no exception to this trend as it needs to cope with different kinds of devices that may connect to it. A client running on a PDA, for example, may need to dynamically add a proxy into the Reflector so that it can transcode the multimedia stream into a format suitable for the PDA (as done in the PalmPlayer video system [14]). These new requirements, together with the lessons learned in the past experiences (see Section 3.3), point out the necessity of supporting dynamic configuration of the Reflector system.

Although dynamic configuration is a field that has been studied intensively in the past decade [15], the dynamic configuration of QoS-sensitive systems pose completely new challenges that have not yet been thoroughly explored. In this case, the system must not only be able to reconfigure correctly, but also be able to carry out this reconfiguration with minimal impact on QoS.

The synergistic relationships between dynamic configuration and QoS are clear. Dynamic configuration allows the use of the best policies for each situation. For example, a mobile computer displaying a video clip to its user could use a protocol optimized for wireless connections when the computer is using a wireless link, but dynamically reconfigure itself to use a TCP connection when the computer is hooked to a wired Ethernet connection. However, if the reconfiguration process, itself, affects the quality of service negatively, it may not be worthwhile to do any reconfiguration at all. Coming back to the example, if the dynamic reconfiguration to the TCP connection is so expensive that the video is interrupted for several seconds, it is better to keep using the wireless link even when the wired link becomes available.

We, now, present an enhanced architecture that greatly improves the possibilities for QoS management with new mechanisms for automatic configuration

(Section 4.1), scalable code distribution and dynamic Reflector reconfiguration (Section 4.2), and dynamic reconfiguration of the network topology to provide fault-tolerance (Section 4.3). Furthermore, the dynamic configuration is carried out with no negative impact on the QoS of the multimedia streams.

4.1 Automatic Configuration

To solve the problem of maintaining the Reflector instances up-to-date as the code of the Reflector program evolves and to customize each Reflector according to its role, we adopted the *automatic configuration* approach [16,17]. The Reflector is divided into a minimal core and components that are linked to the core at runtime to extend its functionality. These components may include the implementation of the different data distribution protocols and mechanisms for encryption, access control, accounting, down-sizing, and transcoding.

The core is the only code that is initially shipped to the nodes that are going to execute the Reflector. This is enough to initiate the bootstrapping process. The configuration information for all the Reflectors in a certain administrative domain are stored in a *configuration service*. Figure 3 depicts a schematic view of the automatic configuration process, which is driven by a module of the Reflector core called *dynamic configurator*. All the code for configuration is encapsulated in the dynamic configurator.

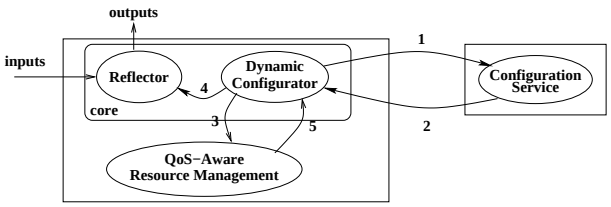


Fig. 3. Automatic configuration architecture

When a Reflector starts, it first contacts the configuration service to retrieve its configuration (step 1 in Figure 3). The configuration service provides three kinds of configuration information to the Reflector:

(1) A list of the components that must be dynamically linked to the core to customize this instance of the Reflector.
(2) A specification of the QoS parameters required by this Reflector.
(3) The input and output connections to be created at startup.

In step 2, the Reflector not only receives its configuration information, but also the actual code for the components to be linked to the core. In step 3, it passes the QoS specification to an underlying *QoS-aware resource management service* that is responsible for managing local machine resources such as CPU and memory, and, when possible, distributed resources such as the network.

The QoS specification may be as simple as an SPDF specification [17] like the one in Figure 4, or as elaborated as a QML specification [18].

```
min_ram                          1  MB
max_ram                          5  MB
average_ram                      2  MB
period                         100  ms
reference_cpu_speed            360 MHz
reference_cpu_type       UltraSPARC-II
min_cpu_share                    5%
max_cpu_share                   30%
average_cpu_share               10%
```

Fig. 4. Low-level QoS specification using the Simple Prerequisite Description Format

The QoS-aware resource management service can be provided by systems like SMART [19] or the Dynamic Soft Real-Time Scheduler (DSRT) [20]. DSRT, for example, can use the QoS specification to perform QoS-aware admission control, negotiation, reservation, and scheduling. Since the architecture relies on user-level processing of data packets, the use of a soft real-time scheduler is important to guarantee that the Reflectors do not impose additional jitter to the multimedia streams.

The configuration is completed in step 4, when the dynamic configurator instructs the Reflector to open the required input and output connections. Afterwards, when the Reflector is running, the resource management service may detect exceptional changes in the resource availability or violations of the QoS requirements that may require modifications in the Reflector configuration. In such cases, it issues a call-back to the dynamic configurator with information about the exceptional conditions (arrow labeled 5 in Figure 3).

This automatic configuration process greatly simplifies system management since the Reflectors always download the most up-to-date version of each component from the configuration service. In that way, it eliminates the need to upload components to the entire network each time a component is updated.

4.2 Scalable Code Distribution and Dynamic Reconfiguration

The automatic configuration process described above consists of a pull-based approach for code updates and configuration. In other words, the Reflectors take the initiative to *pull* the code and configuration information from a central location. In large-scale systems, this may not be enough. We also need a mechanism to allow system administrators to *push* code and configuration information into the Reflector network efficiently. Our architecture achieves that by using the concept of mobile agents [21].

Using a special *configuration agent builder* tool, Reflector administrators are able to create a mobile agent and inject it into the Reflector network. The agent

traverses the Reflector network and, in each node, is able to perform three kinds of actions: reconfigure the Reflector changing its working parameters (e.g., maximum number of clients, security keys, access control lists), install the code for new components (e.g., new data distribution protocols, security mechanisms), and collect information about the Reflector dynamic state (e.g., bandwidth utilization, current number of clients, total number of clients since startup). After visiting the nodes specified by the administrator, the agent returns and displays the results of its actions and the collected information.

Although the introduction of mobile agents promotes significant benefits, it also brings additional security concerns. This is especially worrisome for QoS-sensitive applications since malicious or defective agents may utilize excessive resources and degrade the Reflector QoS. Luckily, the problem of mobile agents security has been studied extensively in recent years and good results are starting to appear. In the Reflector case, it is important to (1) restrict the sources of mobile code to trusted, authenticated entities [24], (2) limit the mobile code's resource consumption [22], and (3) limit its capabilities by confining it to secured environments like sand boxes for Java byte code or the Janus sand box for native code [23].

4.3 Fault-Tolerance

Reliability is an important aspect of QoS. It is difficult to maintain a system working properly in the presence of network failures, node failures, software failures, and partial system shutdowns for maintenance. In addition, keeping the desired level of QoS irrespective of all these disruptive events is a major challenge for modern QoS-sensitive systems. The architecture addresses this problem by using architecturally-aware dynamic reconfiguration based on a framework for representing dependencies in distributed systems [16,17].

In multimedia distribution, the goal with respect to fault-tolerance is to maximize availability without relying on redundant transmissions, which lead to waste of bandwidth. To achieve that, the architecture supports the dynamic reconfiguration of the Reflector inter-connections when failures occur. The distributed system has a knowledge of its own structure and is able to establish alternative routes to deliver the streams to its users. Furthermore, whenever possible, the system performs these reconfigurations without affecting the QoS perceived by its users.

4.3.1 Fault-Recovery Models

In order to build an alternate distribution topology, the system must store enough information so that alternate routes can be found when failures occur.

The question is: where to maintain this information? We considered, initially, a solution in which all the information about fault recovery would be placed in a centralized database accessible by every Reflector. When a Reflector R_1 detects that one of its inputs failed or that it is silent for too long, it would contact a *configuration server* with access to the centralized database and request the address of a reflector R_2 from which it could receive the missing streams. The configuration server would return this information and contact R_2, reconfiguring it to send data to R_1. The advantage of this approach is that very little information is stored on the Reflectors, all the fault-recovery information and policies are centralized in a single location, facilitating the manipulation of this information by a single entity: the configuration server.

The second solution is to store fault-recovery information in the Reflectors. That is, each Reflector would store a list of alternate Reflectors that could be used as backups. The advantage of this approach is that it does not lead to a single point of failure and does not impose an extra load on a possibly already overloaded configuration server. This solution may be more difficult to implement but it tends to be more scalable.

We believe that the optimal solution to this problem is one that encompasses both models. On the one hand, each Reflector should have some knowledge about its local surroundings and should be able to perform reconfigurations by communicating with its immediate neighbors, without being a burden to the centralized configuration server. On the other hand, the configuration server should maintain global knowledge about the system topology. This centralized, global knowledge should be used not only as backup, in case the Reflector's localized information is not enough to keep the system functioning, but also to perform optimizations such as dynamic changes in the network topology to improve the quality of service and promote load balancing.

Our architecture adopts the hybrid model described above. It distributes the knowledge throughout the Reflector network and makes each Reflector aware of its dependence relationships with other Reflectors. Thus, the Reflectors are able to make reconfiguration decisions by themselves, without relying on a centralized entity. In addition to this, the global system topology is maintained in the configuration service so that a Reflector administrator or an "intelligent" software module can perform global optimizations in the distribution network.

Each Reflector contains an object of the type *ComponentConfigurator* [16] which stores a list of entities on which the Reflector depends (its source Reflectors) as well as a list of entities that depends upon the Reflector (client Reflectors or end-user clients). In addition, a *ComponentConfigurator* may also contain a list of *alternatives*, which could serve as alternate inputs for the multimedia streams in case the input for a given channel becomes silent.

5 Implementation

We implemented a prototype of the architecture described in the previous section as a CORBA-based framework, which brings two immediate benefits. First, we are able to utilize a number of standard CORBA services implemented by the CORBA community such as the Naming Service and interact with existing CORBA tools. Second, we have implemented our mechanisms for automatic configuration and dynamic reconfiguration as CORBA services exporting well-defined IDL interfaces. This makes them easily accessible to other systems and reusable in other contexts.

5.1 Implementing Automatic Configuration

The first major change we had to do in the Reflector implementation to accommodate the new design was the adoption of a component-oriented model. We reorganized the implementation of the Reflector program, breaking it into dynamically loadable components. Surprisingly, this proved to be not so difficult, thanks to the original object-oriented design of the Reflector that was based on loosely coupled objects interacting via well-defined interfaces. This component-based model facilitates the customization of the Reflector program at startup, allowing the system to select the components that are best suited for a specific environment at a certain time. It also facilitates the dynamic reconfiguration of running Reflectors to adapt to changes in the environment and to install new versions of components on-the-fly.

Figure 5 presents a schematic overview of the Reflector bootstrapping process in which the Reflector configures itself.

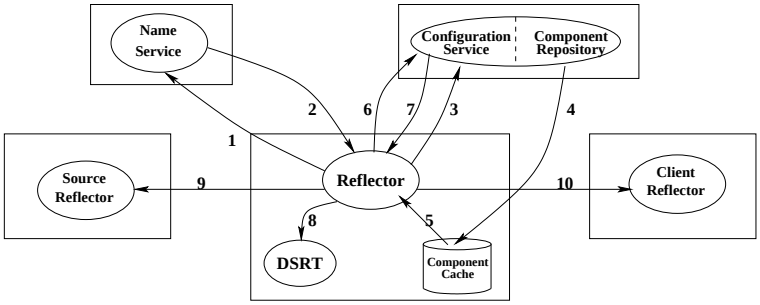


Fig. 5. Bootstrapping a Reflector

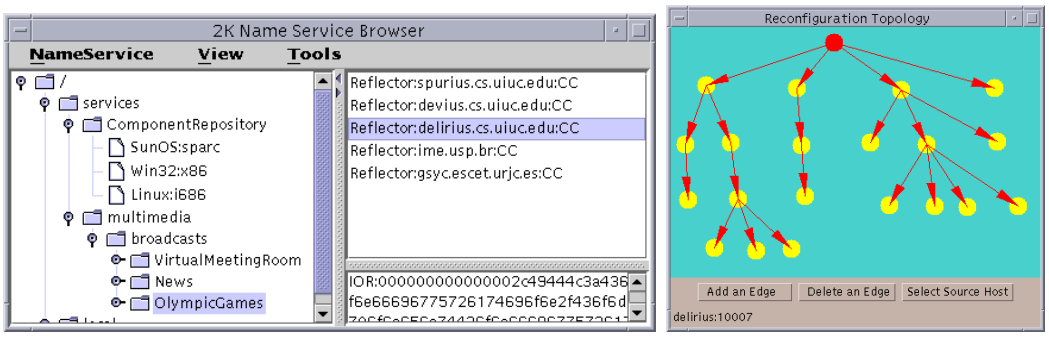
At startup time, each Reflector contacts the CORBA Name Service to locate the configuration service (steps 1 and 2 in Figure 5). From the configuration service, it retrieves its specific configuration information, which may contain three kinds of specifications: (1) the components that must be dynamically loaded to build this instance of the Reflector, (2) the amount of physical

memory and the share of the CPU that should be reserved for this instance of the Reflector, and (3) the input and output connections to be created at startup.

The Reflector uses the information of the first kind (which is obtained in step 3) to contact our CORBA Component Repository and fetch the code implementing the desired components[3] (step 4). It then dynamically links these components into the runtime system (step 5). If configuration specifications of the second kind of are present, they are fetched in steps 6 and 7. Next, the Reflector contacts the underlying middleware to request the reservation of the required resources, which is achieved with the help of the Dynamic Soft Real-Time Scheduler (DSRT) [20].

After loading all the required components, the Reflector registers itself with the Name Service, so that it can be easily located by other system entities, and opens all the input and output connections using the specified protocols (steps 9 and 10).

Administrators of the Reflector system can look at the available broadcast and videoconference sessions by using a graphical user interface that interacts with the CORBA Name Service. Figure 6(a) is a screen shot that exemplifies the use of this interface. It shows three independent Reflector networks: the first called *VirtualMeetingRoom* reserved for audio and videoconferencing, possibly divided according to interest groups; the second called *News* that could contain several news channels; and the third called *OlympicGames* that could contain audio and video broadcast channels related to Olympic events.



(a) Screen shot of the Name Service GUI (b) Agents GUI

Fig. 6.

The administrator can also see that there are three available component repositories for three different kinds of architectures and can use the GUI to upload, download, and remove Reflector components from the repositories. Finally, the

[3] To minimize startup time and network load, the components fetched from the Component Repository can be cached locally.

administrator can see that the *OlympicGames* network is composed of the five reflectors shown in the upper right-hand window. The CORBA IOR (Interoperable Object Reference) of the Reflector at `delirius.cs.uiuc.edu` is shown in the bottom right-hand window.

5.2 Mobile Reconfiguration Agents

To support efficient code distribution and the dynamic reconfiguration of large-scale Reflector networks, we use a generic middleware infrastructure for mobile agents [24]. This infrastructure is based on *dynamicTAO* [25], an enhanced CORBA Object Request Broker (ORB) that exports a reconfiguration interface that can be used to (1) upload executable code to remote ORBs, (2) dynamically load components into the system runtime, (3) change the configuration parameters of application components, and (4) modify the internal architecture of component-based applications.

The infrastructure includes the support for *reconfiguration agents* which work as "smart capsules" that traverse a network of distributed ORBs specified by the application administrator. These agents contain a directed graph representing the order in which the ORBs are to be visited and a script (or Java program) specifying reconfiguration or inspection commands that are processed by each of the ORBs in their path. The results are collected and returned to the application administrator by following the reverse direction in the directed graph.

This infrastructure was all we needed to solve the two initial problems (see Section 3.3) in the previous Reflector version. The reconfiguration interface supports the operations *upload_implementation* and *configure_implementation*, which can be used for code distribution and for sending configuration messages to application components, respectively.

Using a graphical user interface, the Reflector administrator can build a reconfiguration agent that carries new implementations of Reflector components. One of the GUI windows (see Figure 6(b)) allows the administrator to draw the vertices and edges of a directed graph that specifies the collection of Reflectors that will receive the new code and the path through which the code will be transmitted. Each node in Figure 6(b) refers to a Reflector running on a different location. The directed edges specify the paths the agents traverse. Using the GUI, the Reflector administrator can send state inspection or reconfiguration commands to subsets of the Reflector network. The operation results are combined and returned to the administrator who can verify the results of all operations in the Reflectors and take care of individual failures, if any.

5.3 Supporting Fault-Tolerance

The architecture supports fault-tolerance by using *ComponentConfigurator* objects to represent the dependencies between Reflectors. When failures occur, the system uses the dependence information to locate alternate routes and keep the system functioning. The *ComponentConfigurator* implementation stores the dependencies as a list of CORBA IORs, which allows for prompt communication no matter where the objects are located.

A subclass of *ComponentConfigurator*, called *ReflectorConfigurator*, contains the policies for reshaping the network topology in case of failures and encapsulates all the code to deal with these reconfigurations. This approach proved to be very effective in keeping a clear separation of concerns in the Reflector code. The classes that deal with the Reflector's normal operation are totally unaware of the *ReflectorConfigurator* and of any code that deals with reconfiguration. This clean separation also makes it easy to plug different kinds of *ReflectorConfigurator*s to support different reconfiguration policies.

5.3.1 Triggering Reconfigurations

Four kinds of events can trigger dynamic reconfiguration:

(1) A Reflector shutdown message sent by the Reflector administrator or a *kill* command executed by the local system administrator.
(2) Software errors that lead to a segmentation fault or bus error.
(3) A reconfiguration order sent by the Reflector administrator.
(4) Sudden machine crashes or network disconnections.

In the first two cases, the Reflector captures those events using signal handlers installed with the UNIX *signal* function or the Windows *SetConsoleCtrlHandler* function. In the UNIX implementation, for example, the administrator can kill a Reflector by pressing `Ctrl-c` on the terminal executing the Reflector, by sending a shutdown message to the reflector using *telnet*, or by using the *kill* command. The Reflector captures the events generated by `Ctrl-c`, *kill*, segmentation faults, and bus errors by implementing signal handlers for the SIGINT, SIGTERM, SIGSEGV, and SIGBUS signals, respectively.

In the third case, the Reflector contacts the configuration service to retrieve its new configuration information and reprocesses it, reconfiguring its input and output connections. Finally, the fourth case is the only one in which it is not possible to keep the client multimedia stream uninterrupted without relying on redundant streams to the same Reflector. The solution in this case is to detect when the input for a given channel has failed or has been silent for too long and then locate an alternative input either by using the local list

of alternatives or by contacting the configuration server. As described bellow, our current implementation focuses on supporting dynamic reconfiguration in the presence of the first two kinds of events.

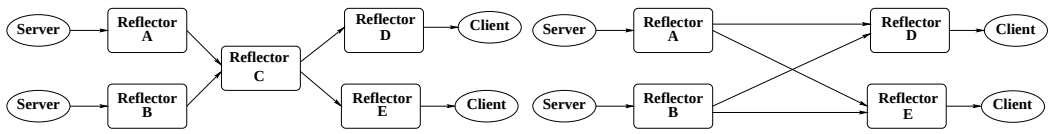
5.3.2 The Reconfiguration Process

When an administrator (or other system entity) requests to kill a Reflector, the system executes a special event handler called *abandonReflectorNetwork*. This handler takes the following three actions.

(1) Unregisters the Reflector from the Name Service.
(2) Using CORBA, sends a FINISHED event to the *ReflectorConfigurator*s of all the sources (inputs) of this Reflector; the event carries a list of the clients of the finishing Reflector.
(3) Sends a FINISHED event to the *ReflectorConfigurator*s of all the clients (outputs) of this Reflector, carrying a list of its sources.

When a *ReflectorConfigurator* receives a FINISHED event from a source Reflector, it adds all the Reflectors in the list of sources of the finishing Reflector to its list of inputs. Conversely, when a *ReflectorConfigurator* receives a FINISHED event from a client Reflector, it adds all the Reflectors in the list of clients of the finishing Reflector to its output list.

Figure 7(a) shows a sample Reflector network where Reflector C has two inputs and two outputs. When C is killed and the reconfiguration process described above completes, the new configuration becomes the one in Figure 7(b).



(a) Network with 5 Reflectors (b) After Reflector C is killed

Fig. 7.

In order to be able to carry out the reconfiguration without any glitches in the multimedia streams and without affecting the system quality of service, we had to adopt a multithreaded solution which we describe in Section 6.3.

5.3.3 The Recovery Process

When a Reflector starts its execution for the first time or when it is restarted after being shutdown for some reason, it executes an initialization process. In this process, in addition to performing the actions described in 5.1, it performs the following three actions.

(1) Registers the Reflector with the Name Service.
(2) Using CORBA, sends a STARTED event to the *ReflectorConfigurator*s of all the clients (outputs) of this Reflector; the event carries a list of the sources of the new Reflector.
(3) Sends a STARTED event to the *ReflectorConfigurator*s of all the sources (inputs) of this Reflector, carrying a list of its clients.

Upon receiving a STARTED event from a new source Reflector, the client Reflector opens a new input connection to the new Reflector. If it is also receiving input from one of the sources of the new Reflector, it closes that input connection as soon as the data from the new source is available. An analogous process happens upon receiving a STARTED event from a new client Reflector. These mechanisms allow the distribution system to recover its original topology after a faulty reflector restarts. Therefore, if the system configuration is the one in Figure 7(b) and the Reflector C recovers, then the configuration automatically switches back to the one in Figure 7(a).

Note that we do not have an automatic mechanism for restarting faulty Reflectors. This requires the administrator's intervention, which seems to be natural since a Reflector goes out of service either because of an administrator's command or because of a failure in the system. In both cases, the administrator's attention is advisable. Alternatively, if desired, the Reflector can be added to the list of daemons that are executed when a machine boots, eliminating the need for manual intervention when a machine crashes and restarts.

6 Performance Evaluation

In this section, we describe the experiments we performed with the Reflector system in the course of the last four years and draw conclusions about its performance. The experiments are divided into the following groups. (1) Evaluating the system's reliability and usability in large-scale, wide-area broadcasts. (2) Measuring the capacity of a single Reflector with respect to the maximum number of clients and maximum bandwidth it can support. (3) Evaluating the impact of reconfiguration of the network topology on the QoS perceived by the end-users. (4) Measuring the performance improvements obtained by using mobile agents for code distribution and reconfiguration of Reflector networks.

6.1 Large-Scale, Wide-Area Broadcast

In March 1997, we performed the first experiments with the initial implementation of the Reflector. At that time, we ran our *TVStation* application in

three SPARCstations 20 running Solaris 5.5 and using the SunVideo interface for video capturing. Using an H.263 software encoder, the *TVStation* was able to generate an average of three frames per second while still allowing the machines to be used as personal workstations. For more than one month without interruption, the machines sent live video from our offices to a small Reflector network. Users could receive the video by pointing their Web browsers to our personal home pages on the web. An applet implementing an H.263 decoder was automatically loaded by the web browser and was able to receive the video even over standard telephone lines. Since H.263 decoding is much less computationally intensive than encoding, a simple 133MHz PC was able to decode and display the three video sessions simultaneously, totaling more than 9 frames per second. Although rather small in the number of Reflectors and bandwidth, this initial experiment showed that it was possible to keep a Reflector network running for several weeks without any interruption.

Our technology was later chosen to broadcast, live over the Internet, the NASA JPL Mars Pathfinder mission [13,5] 24 hours a day from July to October, 1997. The capture station at NASA's Jet Propulsion Laboratory was composed of a 133MHz Pentium-based PC equipped with an Osprey-1000 card for capture and encoding H.263 video. The data rate was set to 24kbps so the stream could be received by 28.8kbps modems. It contained half-rate GSM audio and 3 to 5 frames per second of H.263 video, depending on the encoded images. Figure 8 shows the TV client applet running within Netscape.

Fig. 8. The TV Client Applet

In order to carry out a broadcast of such an enormous magnitude and avoid the complete collapse of our networks, we requested collaboration from other institutions. In a few days, we were able to establish a global network utilizing resources from several universities, research laboratories, space agencies, Internet service providers, computer manufacturers, and mass media corporations.

Our group organized a distribution tree composed of more than 30 Reflectors

spread across five continents. Figure 1 depicts a network similar to the one we utilized. "Master" reflectors received data from the capture station and forwarded them to "intermediate" Reflectors across the globe. These Reflectors forwarded the data to "public" Reflectors which were associated with Web pages and accepted connections from end-user applets. This scheme provided live audio and video to dozens of countries around the globe. Each of the Reflector sites was capable of serving from 30 to 300 simultaneous clients depending upon the bandwidth of its connection to the Internet. This network could potentially serve more than 3,000 simultaneous clients 24 hours a day.

During the initial four days of the broadcast, the Reflector network was able to deliver half a million multimedia streams. Within two weeks, usage climbed to more than one million video and audio sessions. After the first few weeks, interest in the Pathfinder mission decreased, but the broadcast of the NASA *Select TV* over the Internet continued for several months.

During the course of the broadcast, some reconfiguration of the network topology was required in order to accommodate changes in the availability of Reflector hosts. We had to redirect streams to different primary Reflectors and often add or remove public Reflectors. All these modifications were performed remotely without stopping the transmission.

6.2 Single Reflector Capacity

The only bottleneck in the NASA experiment was the bandwidth of Internet connections on the sites hosting Reflectors. In order to guarantee a good quality of service to our clients, we were forced to deny approximately one million connection requests. During a different broadcast (featuring the *Indy 500* race), a single Reflector running on a Sun Ultra 2 was capable of serving near 800 simultaneous clients. This showed that a much larger number of users could be serviced if enough bandwidth were available.

Recent experiments in our laboratory show that a Reflector running on a Sun Ultra-60 with Solaris 7, two 360MHz processors and 1280MBytes of RAM requires less than 30% of one of the processors' time to stream a 26kbps video to 1020 simultaneous clients using TCP over a 100Mbps Fast Ethernet. The bottleneck in this experiment is the maximum number of sockets that a Solaris process is allowed to open in our test machine: 1024. In machines that execute Solaris 7 in 64-bit mode, this limitation no longer exists. The same machine transmitting a 1Mbps stream to 85 clients, uses less than 10% of one of its processors' time. The bottleneck in this case is the network which is not able to sustain a TCP payload data rate of more than 85Mbps.

Assuming that enough machines with high-bandwidth connections were avail-

able, we could use the software technology described in this article to build, for example, a network of 300 Reflectors serving 1000 clients each, totaling 300,000 simultaneous unicast clients. Since the Reflector also supports IP-multicast, the actual number of clients could be much larger.

With large networks composed of more than 100 Reflectors, the problem becomes the manageability of such a large number of Reflectors from a single point. We developed the mechanisms for automatic reconfiguration of the Reflector network in case of failures to address this exact problem. Even with this automated facility, it is advisable to divide the Reflector network in collaborating groups of 100 to 200 Reflectors, each group being managed by a different administrator.

6.3 Reconfiguration Impact on the QoS

More recently, we conducted experiments to evaluate the impact of the automatic reconfiguration of the Reflector network on the quality of the multimedia streams received by the end-users. We carried out this set of experiments on seven Sun Ultra machines (two Ultra-60 and five Ultra-5) connected by a 100Mbps Fast Ethernet. As depicted in Figure 9, three of these machines executed our Reflector program. The fourth machine executed TestServer, a program we created to synthesize an RTP stream with bandwidth and packet rate specified in its command line. A fifth machine executed TestClient, a program that receives the packets from a Reflector and logs the packet arrival times into a file. The sixth machine executed the CORBA Name Service and the last one, the configuration service and component repository.

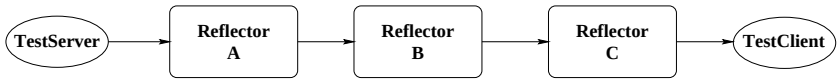


Fig. 9. Reconfiguration experiment testbed

As explained in Section 5.3, when the Reflector B goes down, the system reconfigures itself automatically and adopts the new topology shown in Figure 10. When the Reflector B recovers, the system reconfigures itself back to the topology shown in Figure 9.

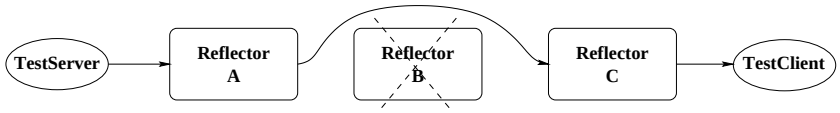


Fig. 10. Reconfiguration when Reflector B goes down

We evaluated the impact of these reconfigurations on the QoS perceived by the end-users by using the information in the log file to compute the packet

inter-arrival time at the TestClient. The following figures plot packet inter-arrival time (in seconds) over time (in seconds). Each experiment lasted for 16 to 18 seconds.

Figure 11 shows the packet inter-arrival times at our testbed client when no reconfigurations take place and with the TestServer sending an RTP/UDP stream of 1.2Mbps at 30 packets per second.

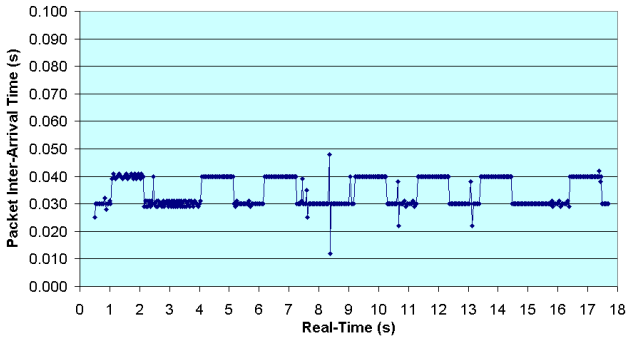


Fig. 11. No reconfigurations. Bandwidth = 1.2Mbps, packet rate = 30pps

The TestServer program uses an adaptive algorithm to keep its average output bandwidth as close as possible to the output bandwidth specified at its command line. Once every K seconds, the program checks its output bandwidth and modifies its packet inter-transmission time to adapt to the changes in its measured output bandwidth. The packet size is fixed throughout each experiment and it is computed by dividing the bandwidth by the packet rate.

In our first experiments, we set $K = 1$ and the desired packet rate to 30 packets per second. The TestServer, then, tries to send one packet every 33.3 milliseconds. But, since the Solaris clock tick is set to 10 milliseconds, it does not let a program sleep for exactly 33.3 milliseconds. Hence, our adaptive program ends up sending one packet every 30 or 40 milliseconds which explains most of the variations throughout the experiment in Figure 11. The small and the localized variations (such as the one at $8.5s$) are due to changes in the network and machine loads caused by other applications.

Figure 12 shows the impact of reconfigurations in our first implementation of the fault-tolerance mechanism. The continuous arrows point to the instants in which the Reflector B is killed and the Reflector network topology switches from the configuration in Figure 9 to the one in Figure 10. The dashed arrows point to the instants in which the Reflector B recovers and the topology switches back to the one in Figure 9. One can see that when we killed the Reflector, there was a big peak in the inter-arrival time at the client. This happened because, in that implementation, as soon as the Reflector received the termination signal, it stopped forwarding packets and sent events to its neighbors announcing that it was going to shutdown. This caused a delay until

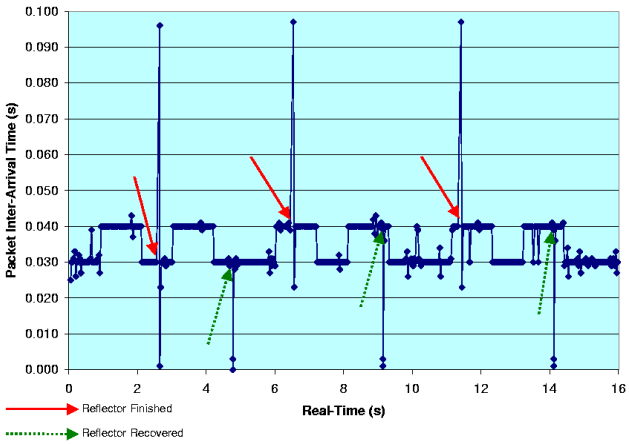


Fig. 12. Reflector reconfigurations. Bandwidth = 1.2Mbps, packet rate = 30pps

the other Reflectors were able to reconfigure their inputs and outputs. In a wide-area network, the delays would be even larger and the QoS perceived by the end-user would be degraded.

We solved this problem by creating a new thread to manage the reconfiguration. While the new thread contacts the Reflector's sources and destinations to announce the end of the service, the old thread continues to perform the Reflector's normal packet-forwarding operations. The Reflector only shuts itself down after it receives a confirmation from its neighbors that its service is no longer needed. If this confirmation does not arrive, the Reflector administrator still has the chance to kill the Reflector anyway at his or her own discretion. Alternatively, the *ReflectorConfigurator* can be programmed to timeout after a given period so that one does not need to rely on administrator intervention.

One can observe another problem in the graph of Figure 12: as the Reflector recovers, there is a sudden change in the packet inter-arrival time, which approaches zero for a couple of packets in a row. This happens because the end-user receives some repeated packets that are sent both by the "backup" connection being deactivated and the new, "recovered" connection. This problem of duplicated packets is solved very easily by implementing a new subclass of *Connection* that uses the RTP sequence number to drop repeated packets.

Figure 13 shows the result of running the same experiment after implementing these two improvements. One can see that the reconfigurations take place without affecting the QoS perceived by the end-user (there is no correlation between the arrows in Figure 13 and the variations in packet inter-arrival times). Furthermore, since the old Reflector keeps a thread performing the Reflector's normal operations until the reconfiguration is completed, the quality of service is not degraded even in wide-area networks where the latency to contact the neighboring Reflectors may be large.

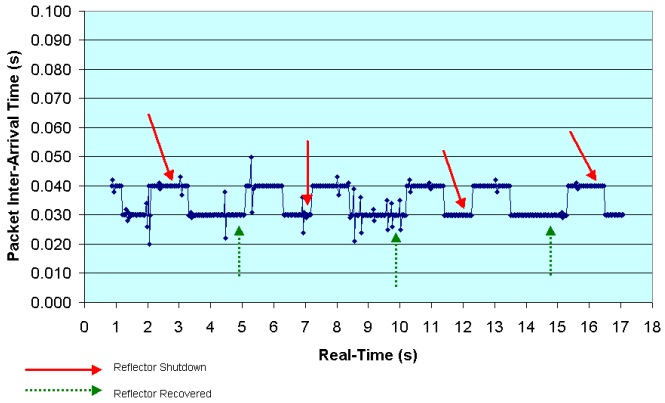


Fig. 13. Reflector reconfigurations. Bandwidth = 1.2Mbps, packet rate = 30pps

The lack of any correlation between the Reflector reconfigurations and the jitter (i.e., the variance in the packet inter-arrival time) is even more clear in Figure 14 that shows a similar experiment carried out in a period of high network and machine loads. In this case, we set K to 10 so that the TestServer would keep its transmission rate constant for longer periods.

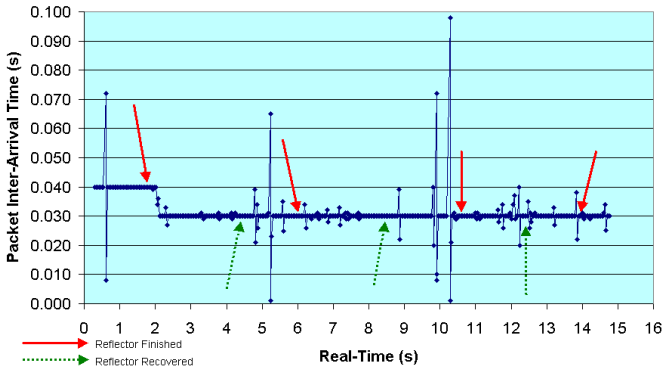


Fig. 14. High network and CPU load. Bandwidth = 1.2Mbps, packet rate = 30pps

Finally, Figure 15 shows one more experiment in which we used the Reflector network to distribute an RTP stream generated by the vat audioconference tool for the MBone. The acoustic perception of the users listening to audioconference confirms what one sees in the graph: the reconfigurations do not degrade the quality of service of the audioconference.

6.4 Code Distribution and Dynamic Reconfiguration with Mobile Agents

In order to evaluate the response time and relative performance gains made possible by our mechanisms for Reflector code distribution and reconfiguration based on mobile agents, we established an intercontinental testbed with the collaboration of the Departments of Computer Science at the Rey Juan Carlos University in Spain and at the Campinas State University in Brazil.

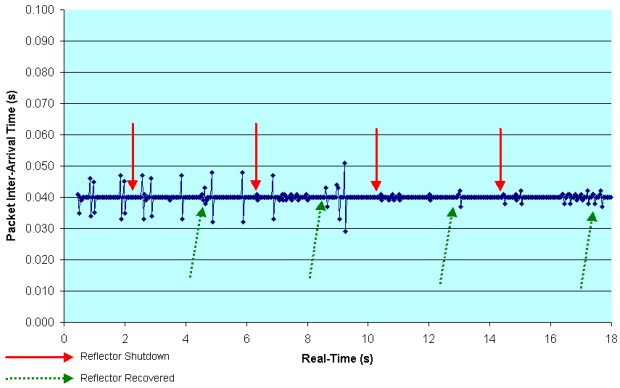


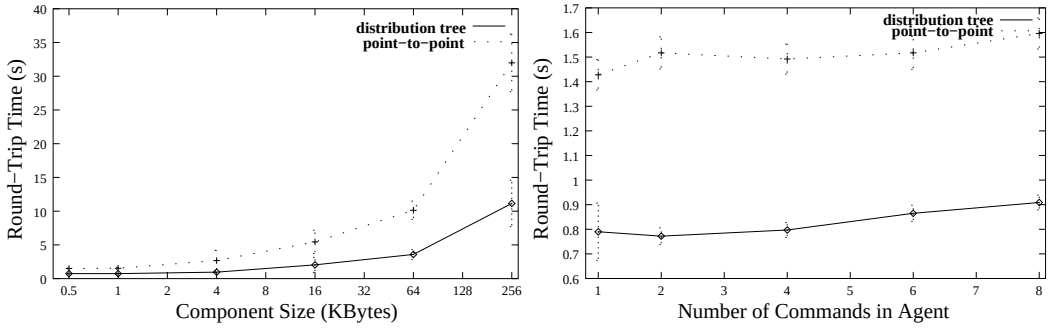
Fig. 15. Reflector reconfigurations. vat bandwidth = 45kbps, packet rate = 25pps

The testbed consisted of the following three groups of machines. (1) Two Sun Ultra-60 and one Sun Ultra-5 machines running Solaris 7 in the `cs.uiuc.edu` domain, (2) three 333MHz PCs running Linux RedHat 6.1 at `escet.urjc.es`, and (3) three 300MHz PCs running Linux RedHat 6.1 at `ic.unicamp.br`. The machines inside each group were connected by 100Mbps Fast Ethernet networks while the groups were connected among themselves through the public Internet. We executed nine instances of our middleware, one in each node, and injected different kinds of agents in this network. To avoid drastic oscillations in the available Internet bandwidth and latency, and to minimize undesired interference, we carried out the experiments during the night [4]. We measured the average bandwidth between our laboratory in Illinois and the remote ones by transferring a 100Kbytes file via FTP five times and measured the latency by using the *ping* command. The average bandwidth and round-trip latency between our lab at `cs.uiuc.edu` and the nodes at `escet.urjc.es` were 76KBps and 170ms, respectively. Between our lab and `ic.unicamp.br`, 32 KBps and 270ms, respectively.

To measure how the benefits of using a distribution tree similar to the one in Figure 6(b) varies with the size of the agent, we sent a series of agents carrying the code for components to be installed in the remote nodes. As shown in Figure 16(a), as the size of the component being uploaded increases, the relative gain of using a distribution tree instead of point-to-point connections to each node increases significantly. Figure 16(b) shows the elapsed time for executing agents carrying from one to eight reconfiguration commands. Each point in the figures represents the arithmetic mean of 10 runs of each experiment. The vertical bars represent the standard deviation.

These results demonstrate clearly that our approach for code distribution and reconfiguration based on mobile agents can provide extreme performance im-

[4] When we ran the experiments during times of high network traffic and congestion, the performance numbers were even more in favor of our mobile agents approach.



(a) Uploading component to 9 nodes (b) Agent visiting 9 nodes

Fig. 16.

provements and better predictability in the management of wide-area distributed QoS-sensitive systems. We expect it to be of great help in the management of our multimedia distribution system.

6.5 Summary

The experiments described in this section let us draw the following conclusions about the Reflector system.

(1) It provides a good level of scalability for multimedia distribution over wide-area networks and is able to keep the service running for several months without interruption.
(2) It supports, at least, 1000 unicast clients per Reflector depending upon the available network bandwidth. In our Ethernet testbed, each Reflector is able to handle streams with a TCP payload data rate up to 85Mbps.
(3) Its mechanism for dynamic reconfiguration of the distribution network to deal with Reflector failures is efficient and does not impact the quality of service as perceived by its end-users.
(4) Its mechanisms for scalable code distribution and dynamic reconfiguration based on mobile agents provide significant performance improvements when compared to traditional client/server mechanisms.

7 Related Work

Our research benefits from and builds on previous work on IP-Multicast [9] and the Internet MBone [1]. The MBone is a virtual network that is layered on top of the Internet to support routing of IP-Multicast packets. It is composed of islands that can directly support IP-Multicast linked by virtual point-to-

point links called *tunnels*. The system described in this article can be seen either as extending the MBone capabilities or, (since we also support IP-Multicast) as a set of tools for managing MBone broadcasts. The MBone relies on a multicast distribution engine that is hard-wired into the networking infrastructure, making it difficult to deploy new technologies for distribution. Our approach not only uses user-level entities that can be replaced easily, but it also supports dynamic reconfiguration of running systems, allowing for easy incremental evolution of the communication mechanisms with respect to QoS, security, and reliability.

Amir, McCanne, and Katz developed an *active service framework* [26] that provides support for the dynamic instantiation of server agents, providing the desired service, in a collection of distributed nodes running their host manager (HM) daemon. They used this framework to implement a Media Gateway (MeGa) service [2]. As our Reflector, the MeGa service can be used to transcode multimedia streams. Indeed, their research has focused on algorithms for efficient transcoding and down-sizing and, more recently, on adaptive bandwidth allocation algorithms and on the dynamic instantiation of media gateways between MBone sessions to deal with heterogeneous networks. Our research on the Reflector architecture shares some concerns with their active service framework (which has some similarities with our automatic configuration service) and with their MeGa service (which could be implemented in our architecture with customized *Connection* classes loaded into the Reflector). But, differently from them, we focus on providing scalable multimedia distribution using multiple communication protocols, reconfiguration of the network topology to deal with failures, and scalable code distribution and dynamic reconfiguration.

In the last few years, the first commercial Reflector-like systems for multimedia distribution started to appear (see, for example, VTel's *Turbo*Cast Reflector [27] and White Pine's *Multipoint Control Unit* for *Meeting Point* [28]). But, unfortunately, literature on this topic is still scarce. Baldi, Picco, and Risso designed a videoconference system based on *active networks* [29]. Their proposed architecture allows clients to customize their videoconference server (or Reflector) by uploading mobile Java code. The main difference is that, in their approach, the Reflector is designed to run in active network routers while ours is designed as a user-level application to be executed on a networked workstation. We chose to implement our system in C++ because our experience shows that Java is not yet ready to provide the high-performance and predictability that most QoS-sensitive applications require. Active networks or *programmable networks* may potentially provide a better throughput and less jitter, since routing decisions are performed at specialized routers rather than on commodity workstations. However, active and programmable networks are not available except for in a few research laboratories. It is not yet clear if these technologies will become widespread in the next years.

8 Conclusions and Future Work

Multimedia applications dealing with large-scale, real-time streaming will become commonplace in future computer environments. The existing hardware and software infrastructure, however, does not provide the high degree of quality of service, flexibility, configurability, and scalability required by these applications. Using our five-year experience with Internet multimedia streaming technology, we have designed and implemented an architecture for scalable multimedia distribution that meets these requirements. In this article, we presented a detailed description of our system and discussed how the lessons learned in early experiments motivated us to implement the enhancements that led to the current architecture.

As a future improvement to the Reflector system, it would be important to develop a service providing "intelligent" management of the data flows among Reflectors and between Reflectors and end-user clients. In the current implementation, a bandwidth control subsystem is used to measure the bandwidth utilization for each input and output connection. It could be easily extended to limit the number of users who are allowed to access a particular Reflector at a given time. With further research, we could also use the bandwidth control subsystem to (1) redirect multimedia streams to different paths in the network to optimize bandwidth utilization and (2) to select which Reflector should be used by each client based on their location and on Reflector load.

Another interesting future work would be the development of a subclass of *Connection*, called *RSVPConnection*, that would reserve network bandwidth using the standard RSVP protocol [30]. This seems to be a relatively easy task if we reuse an existing middleware supporting RSVP such as the one developed by the MONET research group at the University of Illinois.

We believe that the architecture presented in this article has a great potential to influence the next generation of scalable multimedia distribution systems. But, the methods we describe in this article apply not only to multimedia systems. They give significant insights on how to manage any QoS-sensitive distributed application with respect to scalability, dynamic configuration, and fault-tolerance.

Acknowledgments

The authors gratefully acknowledge the work performed by Miguel Valdez, Jim Wong, and See-Mong Tan in the early versions of the Reflector software. Chuck Thompson and Andrew MacGregor provided valuable system administration support.

References

[1] H. Eriksson, MBone: The Multicast Backbone, Communications of the ACM 37 (8) (1994) 54–60.

[2] E. Amir, S. McCanne, H. Zhang, An Application-level Video Gateway, in: Proceedings of ACM Multimedia, San Francisco, CA, 1995.

[3] Z. Chen, S.-M. Tan, R. H. Campbell, Y. Li, Real Time Video and Audio in the World Wide Web, in: Fourth International WWW Conference, Boston, 1995.

[4] S. Tan, R. Campbell, Z. Chen, W. Liao, D. K. Raila, F. Kon, M. Valdez, Adaptation and Synchronization in Low-Bandwidth Internet Video, in: WWW Consortium Workshop on Real Time Multimedia and the Web, Sophia Antipolis, 1996.

[5] F. Kon, R. H. Campbell, S. Tan, M. Valdez, Z. Chen, J. Wong, A Component-Based Architecture for Scalable Distributed Multimedia, in: Proceedings of the 14th International Conference on Advanced Science and Technology (ICAST'98), Lucent Technologies, Naperville, 1998, pp. 121–135.

[6] R. Steinmetz, K. Nahrstedt, Multimedia: Computing, Communications, and Applications, Prentice Hall, Inc., 1995.

[7] OMG, CORBA v2.2 Specification, Object Management Group, Framingham, MA, OMG Document 98-07-01 (February 1998).

[8] OMG, CORBAservices: Common Object Services Specification, Object Management Group, Framingham, MA, OMG Document 98-12-09 (1998).

[9] S. Deering, Host Extensions for IP Multicasting, RFC 1112 (August 1989).

[10] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RTP: A Transport Protocol for Real-Time Applications, revision of RFC 1889 (January 2000).

[11] P. Sinha, N. Venkitaraman, R. Sivakumar, V. Bharghavan, WTCP: A Reliable Transport Protocol for Wireless Wide-Area Networks, in: Proceedings of ACM Mobicom, Seattle, WA, 1999.

[12] M. C. Chan, A. A. Lazar, Designing a CORBA-based High Performance Open Programmable Signalling System for ATM Switching Platforms, IEEE Journal on Selected Areas in Communications 17 (9).

[13] M. P. Golombek, et al. Overview of the Mars Pathfinder Mission and Assessment of Landing Site Predictions, Science 278 (5344) (1997) 1734–42.

[14] C. K. Hess, D. Raila, R. H. Campbell, Design and Performance of MPEG Streaming to Palmtop Computers, in: Multimedia Computing and Networking 2000 (MMCN00), San Jose, CA, 2000.

[15] J. Purtilo, R. Cole, R. Schlichting (Eds.), Fourth International Conference on Configurable Distributed Systems, IEEE, 1998.

[16] F. Kon, R. H. Campbell, Supporting Automatic Configuration of Component-Based Distributed Systems, in: Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, San Diego, CA, 1999, pp. 175–187.

[17] F. Kon, Automatic Configuration of Component-Based Distributed Systems, Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois at U-C (May 2000).

[18] S. Frølund, J. Koistinen, Quality of Service Aware Distributed Object Systems, in: Proceedings of the 5th USENIX Conference on Object-Oriented Technology and Systems (COOTS'99), San Diego, 1999, pp. 69–83.

[19] J. Nieh, M. S. Lam, The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications, in: Proceedings of the Sixteenth Symposium on Operating Systems Principles, ACM, Saint Malo, France, 1997.

[20] K. Nahrstedt, H. hua Chu, S. Narayan, QoS-aware Resource Management for Distributed Multimedia Applications, Journal of High-Speed Networking, Special Issue on Multimedia Networking 7 (1998) 227–255.

[21] D. Kotz, R. S. Gray, Mobile Agents and the Future of the Internet, ACM Operating Systems Review 33 (3) (1999) 7–13.

[22] J. Bredin, D. Kotz, D. Rus, Market-based Resource Control for Mobile Agents, in: Proc. of the 2nd Int. Conference on Autonomous Agents, 1998, pp. 197–204.

[23] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, A Secure Environment for Untrusted Helper Applications: Confining the Wiley Hacker, in: Proceedings of the USENIX Security Symposium, 1996.

[24] F. Kon, B. Gill, M. Anand, R. H. Campbell, M. D. Mickunas, Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents, in: Proc. IEEE Symp. on Agent Systems and Applications / Mobile Agents, Zurich, 2000.

[25] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, R. H. Campbell, Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB, in: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), no. 1795 in LNCS. New York, 2000, pp. 121–143.

[26] E. Amir, S. McCanne, R. Katz, An Active Service Framework and its Application to Real-time Multimedia Transcoding, in: Proceedings of the ACM SIGCOMM Conference, Vancouver, BC, 1998.

[27] VTel, TurboCast Architecture, `http://www.turbocast.vtel.com` (2000).

[28] White Pine, Deploying H.323 Conferencing on Your IP Network, white paper available at `http://www.wpine.com/meetingpoint` (1999).

[29] M. Baldi, G. P. Picco, F. Risso, Designing a Videoconference System for Active Networks, in: Proceedings of the Second International Workshop on Mobile Agents (MA'98), 1998, pp. 273–284.

[30] R. Braden, et al. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification, IETF Proposed Standard, RFC 2205 (September 1997).