A Real-Time Push-Pull Communications Model for Distributed Real-Time and Multimedia Systems

Kanaka Juvva Raj Rajkumar

$January, 1999$

$CMU - CS - 99 - 107$

School Of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890

This research was supported by the Defense Advanced Research Project Agency in part under agreement E30602-97-2-0287 and in part under agreement F30602-96-1-0160. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing policies , either expressed or implied of U.S.Government.

Keywords: Push Communication, Pull Communication, Distributed Systems, Real-Time Systems, QoS, Proxy, End-to-End delay, Multimedia Systems.

Abstract

Real-time and multimedia applications like multi-party collaboration, internet telephony and distributed command control systems require the exchange of information over distributed and heterogeneous nodes. Multiple data types including voice, video, sensor data, real-time intelligence data and text are being transported widely across today's information, control and surveillance networks. All such applications can benefit enormously from middleware, operating system and networking services that can support QoS guarantees, high availability, dynamic reconfigurability and scalability.

In this paper, we propose a middleware layer called the "Real-Time Push-Pull Communications Service" to easily and quickly disseminate information across heterogeneous nodes with flexible communication patterns. Real-time push-pull communications is an extension of the real-time publisher/subscriber model, and represents both "push" (data transfer initiated by a sender) and "pull" (data transfer initiated by a receiver) communications. Nodes with widely differing processing power and networking bandwidth can coordinate and co-exist by the provision of appropriate and automatic support for transformation on data. In particular, unlike the real-time publisher/subscriber model, different information sources and sinks can operate at *different* frequencies and also can choose another (intermediate) node to act as their proxy and and deliver data at the desired frequency. In addition to the synchronous communications of the publisher-subscriber model, information sinks can also choose to obtain data asynchronously. This service has been implemented on RT-Mach, a resource-centric kernel using resource kernel primitives [7]. This paper presents an overview of the design, implementation and a performance evaluation of the model. We also test the applicability and versatility of this service using *RT-Conference*, a multi-party multimedia collaboration application built on top of this model. Finally, we summarize some key lessons learned in this process.

Contents

List of Figures

List of Tables

1 Introduction

The advent of high-performance networks such as ATM and 100 Mbps networks in conjunction with significant advances in hardware technologies are spawning several distributed real-time and multimedia applications. Distributed multimedia systems, in particular, are becoming more prevalent and effective in making widespread information accessible in real-time. Examples include video-conferencing over the internet, multi-party collaborations systems, internet telephony etc. Data communications in these systems take place among geographically distributed participants, whose computing and networking resources can vary considerably. A service infrastructure which supports such distributed communications should be scalable, flexible and cater to different CPU/network bandwidths while providing real-time guarantees. We propose a real-time push-pull communication model, a middleware substrate which provides communications services for different real-time applications executing on top of it, while providing real-time guarantees. The push-pull communications model an extension of the real-time publisher/subscriber [4, 5, 6]. It is a real-time event channel between information sources and sinks. The layering of the Push-Pull Communications as a middleware service in a single node of a real-time system is illustrated in Figure 1.

Services somewhat similar to our real-time push-pull model have been provided in the past and others are currently being developed. For example, our current model builds and extends the real-time publisher-subscriber model of [4]. *Maestro* [1, 2] is a support tool for distributed multimedia and collaborative computing applications. 'Salamander' [3] is a push-based distribution substrate for internet applications. The primary difference between our model and others is that we focus on maintaining timeliness guarantees and predictability, which in turn may necessitate a relatively closed environment. Real-time CORBA services and real-time Object Request Brokers [8, 11, 12] are also attempting to address some aspects related to our model. Our model and middleware service are specifically designed such that they can be utilized within or from CORBA interfaces.

1.1 Motivation

The real-time publisher/subscriber communication model illustrated in Figure 2 can be considered to represent "push communications" where data is "pushed" by information sources to information sinks. As a result, subscribers can obtain information only at the rate at which the data is being pushed. This model is appropriate and efficient for periodic and synchronous updates between sources and sinks which are operating at the same frequencies[1]. Unfortunately, this can be very limiting in many cases where different clients have different processing power and/or widely varying communication bandwidth (because of connectivity to a low bandwidth network such as a telephone modem or an encrypted satellite link). If consumers did not have the same processor power or network bandwidth, a publisher must either falsely assume that they all have the same capability or publish two (or more) streams to satisfy consumers with different capabilities.

It would be very desirable if a client with a relatively low processing power and/or communication bandwidth is able to consume published data at *its* own preferred rate. In other words, the data reaching this client depends on its own needs, and not that of the publishing volume/rate of the publisher. Also, the real-time publisher/subscriber model is completely *synchronous*: subscribers normally block on a "channel" (represented by a *distribution tag*) waiting for data to arrive. Publishers produce data at the rates that they determine, and the published data is immediately sent to the subscribers on that distribution tag.

The Push-Pull Communications model addresses both of the above concerns:

- First, it allows consumers on the same data streams to receive *and* process data at (locally determined) rates, which are independent of those used at the information sources. As a result, clients with high or low processing power and/or high or low network bandwidth can still usefully consume data on a stream. In addition, this can happen without the knowledge of the data producers who do not have to distinguish among the capabilities of the receiving consumers.
- Secondly, in the push-pull communication model, data can be either "pushed" by an information producer or "pulled" by an information consumer. A "pulling" consumer can choose to consume data at a rate lower than the data production rate. In the extreme, a pulling consumer can choose to only consume data *asynchronously*.

In summary, the real-time push-pull communications model we propose not only continues to support efficient synchronous communications among homogeneous nodes, but also supports asynchronous communications among heterogeneous nodes.

[1] A subscriber may choose to operate at a different lower frequency by, for example, skipping every other published datum on a subscribed tag. However, for this to happen, the subscriber must still receive and "consume" the datum albeit in a trivial "drop-it" fashion.

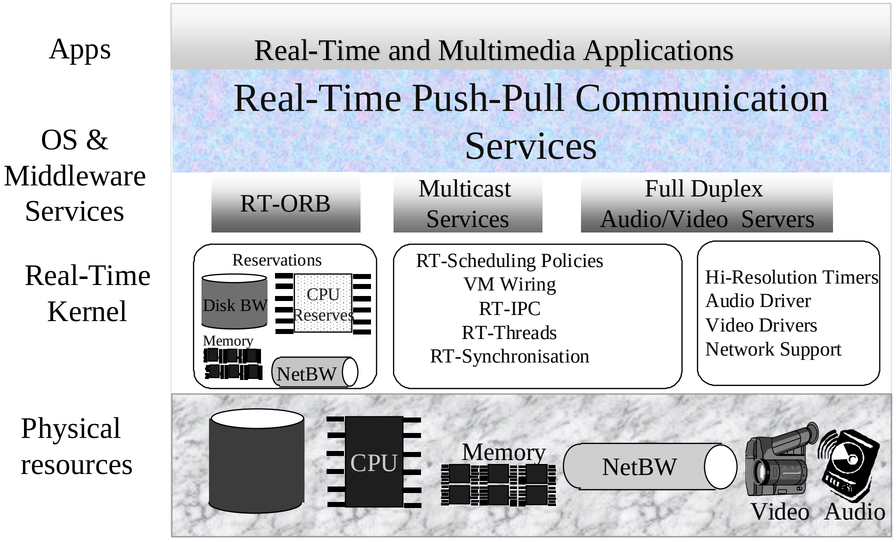

Figure 1: *Real-Time Push/Pull Communications as a Middleware Service.*

1.2 Objectives

The following are the objectives of the Push-Pull model:

Synchronous Communications: Publishers and subscribers operate at the same frequency. This can be called *push communications*. Many hard real-time systems come under this categeory of communications.

Asynchronous Communications: The communication model should cater to different bandwidth and processing capabilities of producers and consumers. Publishers continue to pump information at their own rate and subscribers receive data at their own locally determined rates. On the other hand, publishers and subscribers should be able to operate asynchronously. The frequency transformation and scaling that needs to be done is carried out transparently. Possible transformations that can be supported are

- *History Buffers*: A finite sequence of a real-time activity can be buffered at a location and can be pulled by a subscriber on demand at a later point of time.
- *Flexibility:* For performance and applicability reasons, the location of the "proxy" for transforming data frequency should be flexible. If the proxy is located on the client with lower capability, the benefits of scaling can diminish significantly. On the other hand, if the proxy were sitting at the information source itself, the load on that node can become undesirably high. Thus, the data transformation can take place right at the source or at the sink or at *some* intermediate node. A provision should exist for clients, system administrators, or information sources to choose the actual proxy location at start-time and/or dynamically.
- Scalability: The communication model should be scalable for several nodes and support many applications and multiple clients at the same time.

In our approach, we use a *proxy* to perform the transformation on the data transparent to the data source and the data sink during data communications. These proxy agents can perform the transformations without affecting the functionality of producers and consumerss (but potentially increasing the end-to-end delay between the two). Data scaling might also require that the proxy be aware of the semantics of the data (for example, that it is a raw video stream). A proxy is used only when necessary, and two (or more) clients can be receiving at two (or more) rates from the same data channel. We use *pull communications* to support history buffers and pulling of older messages on demand.

QoS Guarantees : The communications service should be predictable, analyzable and provide timing guarantees. In other words, the components of the communications service should make use of real-time scheduling

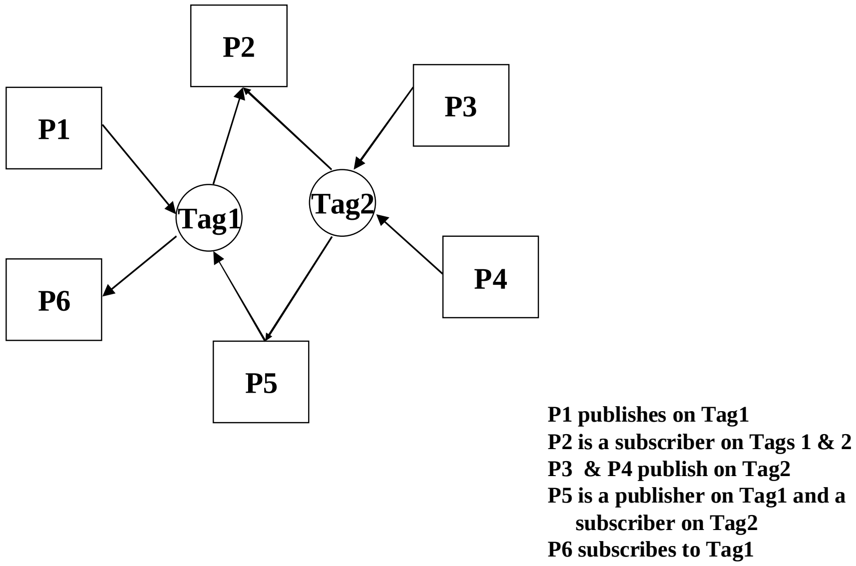

Figure 2: *The Real-Time Publisher/Subscriber Model*

schemes such as priorities and synchronization protocols which bound and minimize priority inversion. For dynamic online control, admission control should be executed at several layers to meet the QoS requirements. Reservation of memory, network bandwidth, CPU bandwidth and disk bandwidth [7,9,10] can be used to provide end-to-end QoS guarantees.

Protection and Enforcement: Since multiple communication channels can be in use simultaneously at different rates, there should be support for spatial and temporal protection among the various channels. As an example, increasing the frame rate of a video chat group should ideally not adversely affect the worst-case end-to-end delays of a surveillance application. The scheduling/dispatching layers should use primitives which support enforcement when possible. For instance, a reservation-based scheduling scheme [7] provides temporal protection and guaranteed timeliness, while priority-based schemes can provide predictable timeliness under worst-case assumptions that do not enforce protection.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we present a high-level overview of the real-time push-pull communication model and describe the primary components of the model. In Section 3, we describe our implementation of the real-time push-pull model, and present an evaluation of our implementation in terms of round-trip delays encountered between communicating parties. In Section 4, we describe in detail a multi-party multimedia collaboration system named *RT-Conference* that we built on top of the real-time push-pull model to evaluate its usability and efficacy. We also summarize some key lessons that we learned during the development of both the push-pull layer and the application. Finally, we present our concluding remarks in Section 5.

1.4 Comparison with Related Work

In this section we discuss the related systems and compare our approach with the others. Our communication model is built on top of resource kernel[7] and uses resource kernel primitives real-time priorities, real-time threads , RT-IPC and basic priority inheritance mechanisms at all leves (client and daemons) and focusses on real-time gurantees like timeliness. The model fits well with in the context of both hard and soft real-time systems and particularly it is very promising to distibuted multimedia applications like videoconferencing. The example application we describe in this paper proves these concepts. The model can be easily extended to use reservations. There is some related work going on this area to support distributed mulitmedia applications. Maestro[1] is a midddeware support tool for distibuted multimedia and collaborative computing applications. Salamander[3] is a push based distribution substrate designed to accomodate the large variation in Internet connectivity and client resources through the use of application specific plug-in modules. However[3] doesnot address temporal protection among different virtual data channels and it does the best fit of resources among applications.

In [21], Fox et. al. propose a general proxy architecture for dynamic distillation of data based on ckient variation. It doesn't address temopral dependencies which impose tight timing constraints on the distillation process, which affects the architecture of the distller. The work in [16] addresses adding group communications support to CORBA. Some work is going on RT-CORBA [8, 11, 12] to provide QoS and minimize end-to-end latencies in CORBA based systems. Work in [15, 17, 19, 22] address group communication protocols and clock-synchronisation.

2 The Real-Time Push-Pull Communication Model

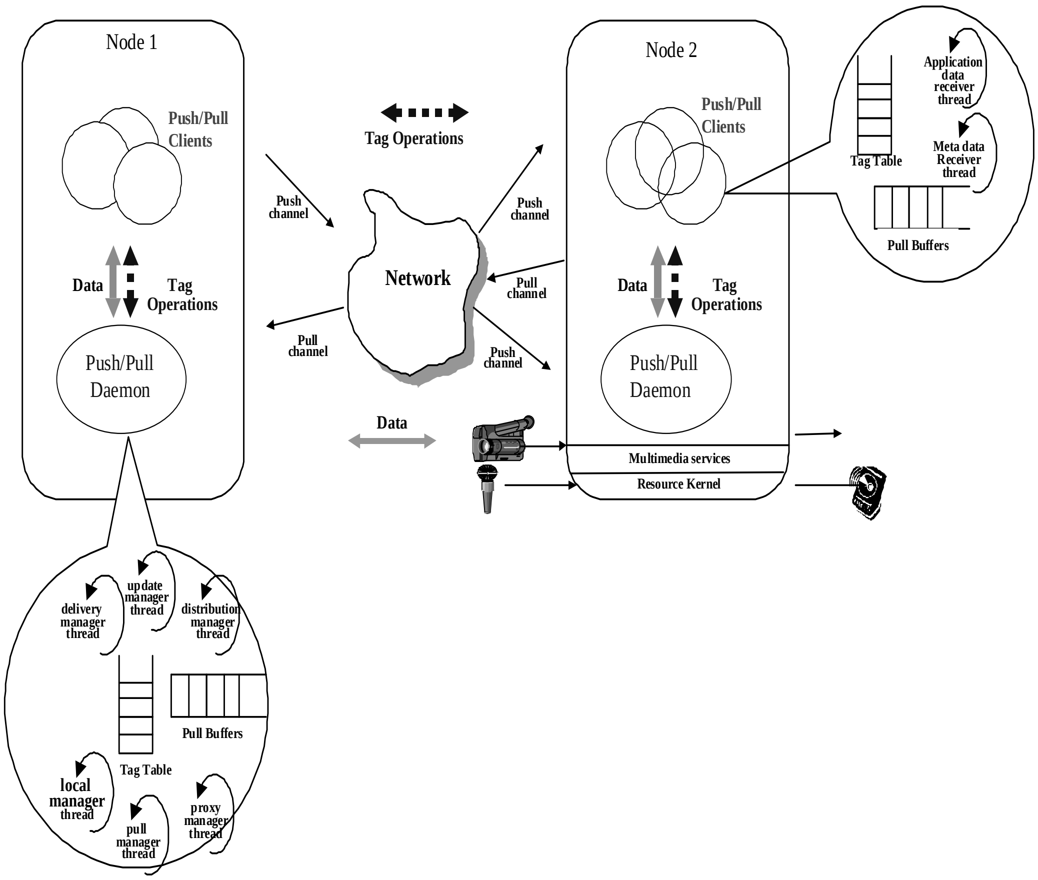

Figure 3: *The Architectural Components of the Real-Time Push-Pull Communications Model.*

The push-pull communication model is an extension of the real-time publisher/subscriber model. In this section, we summarize the real-time publisher/subscriber model and describes the push-pull extensions to the model.

The architecture of the real-time push-pull communications model is illustrated in Figure 3. Nodes in the system are assumed to be interconnected using a real-time network fabric. Every node runs a daemon comprising of multiple threads. A client wishing to communicate using the real-time push-pull layer interfaces to a library in its own address space. The library maintains local information from the real-time push-pull substrate and its own local buffers. This information maintained in the client is structured such that any damage to this data will affect only this client. The threads within the daemon and the client are discussed in more detail in Section 3. The individual components of the push/pull communication model are given below.

- *Distribution Tag:* Distribution tag is the logical handle for communication. Tag Table entry in each daemon is a repository of all the distribution tags.
- *Push/Pull Daemon:* The Push/Pull daemon executes on every node involved in the push/pull communications. The major function of the daemon is to establishment communication between clients.
- *Push Client:* Publishers(information source) and Subscribers(information sink) are push clients. Publishers publish data on a specified distribution tag and subscribers subscribe to the tag for data.
- *Pull Client:* Pull subscribers are pull clients.
- *Push Channel:* Push channel is the logical communication channel to which the publisher pushes the data.
- *Pull Channel:* Pull channel is the logical communication channel on which the pull subscriber obtains the data.
- *Pull Buffers:* Pull buffers are the repository of the data samples, which can be pulled by the pull subscribers.

The key aspects supported by the real-time push-pull model are described next.

2.1 The Real-Time Publisher/Subscriber Model

The push-pull communication model is based on and provides complete support for the Real-Time Publisher/Subscriber model [4] for communication between information producers (publishers) and information consumers (subscribers). In the real-time publisher/subscriber model illustrated in Figure 2, a consumer can consume information from multiple sources, while a producer can produce information for multiple consumers. A producer can also be a consumer and vice-versa. A data channel represented by "a distribution tag" represents each category of data that is available in the system. For example in a multiparty-multimedia collaboration application, which is described in the later sections of the paper audio, video, text and whiteboard data are represented by different tags *audio, video, text and whiteboard* respectively. A publisher publishes on a distribution tag, and subscribers to that tag can consume the information published on that tag. However, a publisher need not know who the subscribers to its published information are, and a subscriber need not know who the publishers of its consumed information are. In other words, the publisher-subscriber communication model allows a general many-to-many communications model.

In the real-time publisher-subscriber communication model, timing delays for communications between an information producer and an information consumer are predictable, analyzable *and* efficient. As may be expected, this analyzability is based on the assumption that the communication demands are known *a priori* or will be explicitly specified at run-time using admission control schemes.

The implementation of the real-time publisher-subscriber communication model allows the information about publishers and subscribers to be stored as close to the publisher as possible (namely in its own process address space on its own node as a library). This proximity to the end-user yields both performance and predictability benefits. A distributed fault-tolerant name-service hidden from the application interface allows the physical communications among publishers and subscribers to occur [6]. This name-service allows processor nodes to fail and/or to (re)join the system. When failures happen or when nodes (re)join, the naming service continues to function and real-time publication/subscription on distribution tags continue to function normally.

2.2 Proxy

A consumer upon subscription to a tag (either as a push-client or a pull-client) can specify frequency scaling that must be done by the middleware service on the data stream it is subscribing to. A proxy will be created to accomplish this frequency transformation, and once established, is transparent to the data source as well as the data sink. The existence (or non-existence) of the proxy is retained in the attributes of the distribution tags (channels) maintained by the daemons and client libraries, and used appropriately when data is published or pulled. In other words, the proxy sits as a transparent and useful entity between a source and a sink. This is illustrated in Figure **??**.

A push-pull daemon is run on every node of the system using our middleware service, and the proxy agent when needed resides typically within the daemon of the node where it is located. It can reside in a client library if the proxy is at either the publisher or the subscriber node. Data published to a subscriber who has requested a proxy is frequency-transformed as it is being transmitted (by dropping as necessary). The critical aspect is that multiple subscribers on the same channel may want to have their proxy in distinct modes. For example, *Subscriber-1* may want to have its proxy in `Proxy-Mode-1`, *Subscriber-2* in `Proxy-Mode-2` and *Subscriber-3* in `Proxy-Mode-3`.

2.3 Pull Communications

"Pull" communications is used by a subscriber to "pull" specific data samples asynchronously on demand. For instance, a subscriber may want to pull dynamically a particular recent sequence of an ongoing video conference. An aircraft control system may want to pull weather information about a particular region in the past two hours. As a result, it is desirable that the middleware service support a **history buffer** which stores recent versions of data samples published on a channel. A customer can then request on demand the most recent copy of a data sample, the n^{th}-most recent version or a specific absolute sample from this history.[2] The pull sequence as viewed by a pull client is illustrated later in Figure 7.

A subscriber issues pull requests to pull the messages. Time-stamping and/or versioning of data is required to indicate a specific message. In our current implementation, all the messages carry a sequence number and the sequence number is used to pull a specific message. End-to-End delays for *pull* communications can be predicted in a way similar to *push* communications.

2.4 Why is this model "Real-Time"?

A logical question to ask of the real-time push-pull model is what makes it deserve the qualifier "real-time"? As argued in [4], the real-time publisher-subscriber model (which represents the "push" portion of the real-time push-pull model) exhibits two desirable properties:

1. If the parameters of all tasks in the system are known (either *a priori* or because of explicit specification at admission control time), the worst-case end-to-end delays can be computed using traditional real-time techniques such as Rate-Monotonic Analysis. This is made possible by the use of real-time scheduling algorithms and real-time synchronization primitives that bound priority inversion.
2. The model clearly distinguishes between non-real-time actions (like `Create_Tag`) and real-time actions with deadlines (like `Publish_Data`). The former is always handled at lower priority than the latter. In addition, the design enables the real-time actions to communicate using a very direct path between senders and receivers. This in turn reduces the end-to-end delays for real-time applications.

Similarly, the "pull" model is also designed such that assertions about predictable end-to-end delays can be made and proved. Consider a "pull client". Its end-to-end delay to the "pull buffer" can be predicted analogous to the "push" side. In addition, if a publisher is assumed to update the "pull buffer" at a certain rate, the timeliness of its updates as a publisher can also be guaranteed. Combining these two aspects, it is possible to guarantee that the version (or timestamp) of a data sample obtained by a "pull" client is no older than the most recent version by a known, fixed offset.

3 Implementation of the Real-Time Push-Pull Model

Our real-time push-pull model has been implemented on RT-Mach, Resource Kernel version RK97 [7] using the resource kernel primitives of real-time scheduling and reservations which also bound priority inversion. RK97a is the recent release of the resource kernel and supports extended CPU reservations, disk reservations, real-time threads, real-time IPC, real-time synchronization primitives, very high resolution timers, full duplex audio support, real-time video capture, a real-time window manager, fault-tolerant timeline scheduling, and wireless roaming support.

Each daemon in the communications infrastructure consists of six real-time threads, each with a different real-time priority. The client library used with each application consists of two real-time threads. UDP is used for communication between clients and daemons. RT-IPC is used for sending application data from the daemon to local clients.

Daemon:

1. **Local Manager:** It receives the client's requests (tag operations) and updates the local tag table.
2. **Distribution Manager:** It sends the updated tag table to both remote daemons and local clients.

[2] Our current implementation supports a circular buffer which stores a fixed number of the most recent samples published.

Thread	Main function	Relative priority
Local Manager	Receives requests as to communication data and updates the local tag table	12
Update Manager	Updates the local tag table when it receives updated tag information from a remote daemon	10
Distribution Manager	Multicasts updated tag information to both remote daemons and local clients	11
Delivery Manager	Multicasts application data to local clients that are subscribing on this data	2
Proxy Manger	Receives data from publisher, scales frequency of data and transmits to subscriber.	1

Figure 4: *The daemon threads, their functions and their priorities. Lower numeric values represent higher priorities.*

3. **Update Manager:** It updates its local tag table when it receives the updated tag information from a remote daemon.
4. **Delivery Manager:** It multicasts the application data to local clients that subscribed to the tag.
5. **Proxy Manager:** It receives proxy data and does the transformation. Transformed data is multicast to the daemon of the proxy subscribers. This applies only for `Proxy-Mode-3`. For `Proxy-Mode-1` and `Proxy-Mode-2` transformation is done by the subscriber and publisher respectively.
6. **Pull Manager:** It receives the pull data and sends it to the pull subscriber.

The priorities of the threads in the daemon are listed in Figure 4.

Client Library:
There are two real-time threads in the application library linked with each application using the real-time push-pull service:

1. **Application Receiver:** It receives the published data from the daemon using RT-IPC. Data is stored into a buffer protected by a condition variable.
2. **Network Receiver:** It receives all updates to distribution tags it locally needs from its daemon.

The priorities of the client threads are discussed in more detail in the next section.

3.1 Application Programming Interface

The real-time push-pull communications model has programming interfaces to create/destroy a distribution tag, obtain publication/subscription rights on a distribution tag, push/receive data synchronously on a distribution tag, and pull data asynchronously from a distribution tag. In addition, the specification of the proxy and the setting of QoS attributes is accomplished by setting attributes on a distribution tag. This API is presented in Figure 5.

3.2 Client's Usage of the Real-Time Push-Pull Model

The sequence of actions that a client must perform in order to use the real-time push-pull model is outlined below. Each node which wants to communicate using the real-time push-pull model must have a daemon running. A client calls `ClientAPI_Init()` where communication ports are initialized. It creates a distribution tag and obtains a tag handle is obtained for its communications. A publisher obtains a publication rights on the tag, and a subscriber obtains subscription rights on the tag. A proxy subscriber specifies the location of its proxy by setting necessary proxy attributes on the particular tag. A pull subscriber specifies the pull attributes by setting the pull attributes on the tag.

A publisher publishes messages on the tag, and corresponding subscribers receives these messages. A proxy subscriber receives its messages from the proxy rather than from the publisher directly at appropriately transformed rates.

The sequence of events triggered in the real-time push-pull infrastructure by some invocations made by the client are illustrated in Figure 6. They can be summarized as below.

When a non-steady-state path request(tag creation/deletion request, publish/subscribe rigth request, set_push_attribute, set_pull_attribute) is issued,

```
/* initialization and closing functions */
ClientAPI_Init();
int UnRegister();

/* create and destroy tags */
tagID_t Create_DistTag(tagID_str_t tag);
int Destroy_DistTag(tagID_t tag);

/* set QoS attributes of tags */
int Set_Tag_Attributes(tagID_str_t tag, int flavor, int param1, int param2,char *hostname);
int Set_Pull_Attributes(tagID_str_t tag, User pub, int param1, int param2,char *hostname);

/* get publication/subscription rights */
tagID_t Get_Send_Access(tagID_str_t tag)
int Release_Send_Access(tagID_t tag)
tagID_t Subscribe(tagID_str_t tag)
int Unsubscribe(tagID_t tag)

/* publish, receive or pull messages */
Send_Message(tagID_t tag, void *msg, int msglen);
int Recv_Message(tagID_t tag, void *msg, int *msglen);
int Pull_Message(tagID_t tag, void *msg, int *msglen);
int Recv_Message_From(tagID_t tag, void *msg, int *msglen, inaddr_t *source, u_int *pid, struct timeval *tp,
int *seqno);

/* Get tag information */
int Get_Tag_List(int *count, char *** tag_names);
int Get_Local_Tag_List(int *cnt, char ***tag_list);
User* Get_Publisher_List(tagID_t tagid, int* cnt);
tagID_t Get_TagID(tagID_str_t tagname);
```

Figure 5: The Application Programming Interface for the Real-Time Push-Pull Model.

1. The client's local request service sends a request to the local manager of of the Push/Pull Daemon
2. The local manager checks to see if the tag status change can be made. If so, it sends an update status to the update manager of the remote Push/Pull daemons. Then, it updates its local tag table.
3. The local manager then sends a response (SUCCESS/ERROR) back to client.

When the steady-state path request "publish" message on a tag is issued,

1. The calling thread automatically checks the local tag information for valid information.
2. If valid, it sends copies to all receivers on local node(if any) and atmost one message to each remote delivery manager/proxy manager whose node has atleast one receiver/proxy for that tag.

When the steady-state path request "receive message on a tag" is issued,

1. The calling thread atomically checks the local tag information for valid information.
2. If valid the calling thread issues in RT_IPC call to receive the message. If a message is already pending , a message is returned to the client.
3. If no message is pending , the client waits on IPC port for a message

When the steady-state path request "pull message on a tag" is issued,

1. The client's local request service sends a request to the local manager of the local Push/Pull daemon and waits at an IPC port for message
2. The local manager checks for pull mode and in mode-1 sends the pull message to client. In mode-2 and mode-3 it sends the pull request to the correpnding Push/Pull daemon and waits for a message.
3. After receiving the pull message from remote daemon, the message is sent to the client.

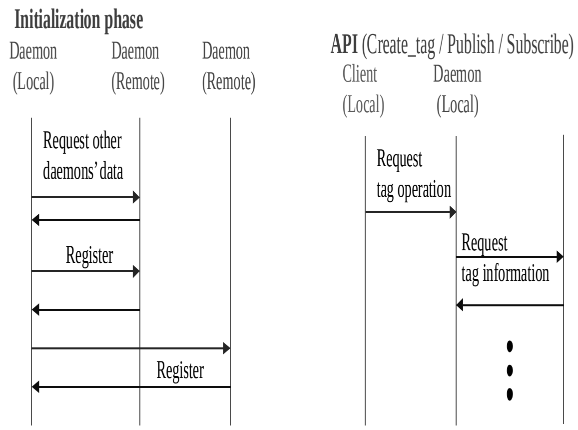

Figure 6: *The Sequence of Actions on Daemon Arrival and Tag Creation.*

The view of events seen by a pull subscriber is presented in Figure 7. Two primary data structures exported to a client include a `Tag` structure representing the distribution tag on which messages are sent/received, and a `User` structure which represents an identity of a process. These two data structures are presented in the Appendix.

3.3 Performance Evaluation

The real-time push-pull model has been successfully designed and implemented. This section describes a set of measurements obtained on a network of three Pentium-120 MHz PCs with 32MB RAM running RT-Mach version RK97a. The network was a dedicated 10Mbps ethernet, and a Subscriber, a Publisher and a Proxy were run on three separate nodes.

3.3.1 The Performance Impact of a Proxy

The experiment we conducted to measure the performance impact of a proxy is as follows. A publisher transmits a 64-byte message which is received by a subscriber, who in turn re-transmits that message by publishing on a separate tag. The original publisher receives this message and the time taken for this sequence to complete at the first publisher node corresponds to a *Round-trip Delay.* We calculated the average of this round-trip delay after 100 messages. We also calculated the standard deviation of the round trip delay to evaluate the perturbations in the model. These measurements based on an unoptimized implementation[3] are summarized in Table 1.

We repeated the experiment in 2 configurations: without a proxy, and with a proxy inbetween the first publisher/subscriber pair. In addition, the proxy if used could be located on the publisher site, the subscriber site or an intermediate site. The measurements of round-trip delays are listed in Table 1.

As can be seen, the presence of a proxy at a subscriber node or a publisher node adds very little overhead compared to the case of having no proxy at all. In this case, the proxy was scaling the data stream by a factor of one, i.e. passing the data straight through. When the proxy is on a remote node, a latency of about $3ms$ is added to the round-trip path.

[3]The system measured uses an ISA bus 8-bit Ethernet card, and we expect significantly better absolute performance numbers on a 32-bit PCI card.

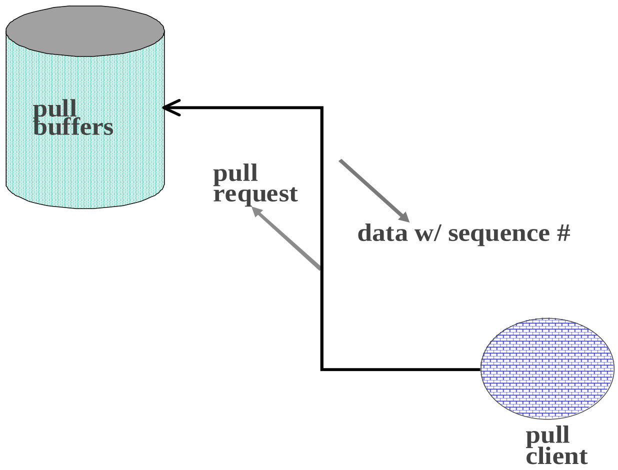

Figure 7: *The Pull View.*

Mode	*Round Trip delays in ms*	*std deviation in ms*
No Proxy	6.012933	0.0118728
Proxy At Subscriber	6.01278	0.021036
Proxy At Publisher	6.0176868	0.01262178
Proxy At Intermediate Node	9.349978	0.220098

Table 1: Round-Trip delays for Different Proxy Locations

3.3.2 The Impact of Message Lengths and Multiple Subscribers

Number of Subscribers	*Message Size(Bytes)*					
	64	256	512	1024	1536	1792
1	5.914	6.018	7.369	6.463	9.014	9.020
4	5.975	7.818	8.972	8.981	8.992	9.013
8	8.973	9.513	10.483	10.899	10.141	12.160

Table 2: The Round-Trip Delays (in ms) encountered with *no* proxies.

We repeated the previous experiment but varied the length of each message. One would expect that the round-trip delays would increase with message size. The round-trip delays as message length is varied and in the absence of any proxy are listed in Table 2. As expected, the round-trip delay increases (almost) linearly with the increase in the size of the message. We then tested the case where there is more than 1 subscriber subscribing to the first tag on the same machine and only the last subscriber responds to the publisher. The round-trip delays farther increase because of additional preemption on the receiving node.

The round-trip delays as message length varies and in the presence of a proxy on an intermediate node are listed in Table 3. Again, the round-trip delay increases (almost) linearly with message length. Furthermore, the addition of the proxy adds at least $3ms$ to the round-trip delay, which, as one might expect, increases as the message length increases.

Message Size (bytes)	Round Trip delays (in ms)
1	8.964
64	9.361
1024	11.760
1536	12.195
1792	14.986

Table 3: The Round-Trip Delays with a *proxy on an intermediate node.* The number of publishers is 1.

4 RT-Conference: A Multi-Party Multimedia Collaboration System

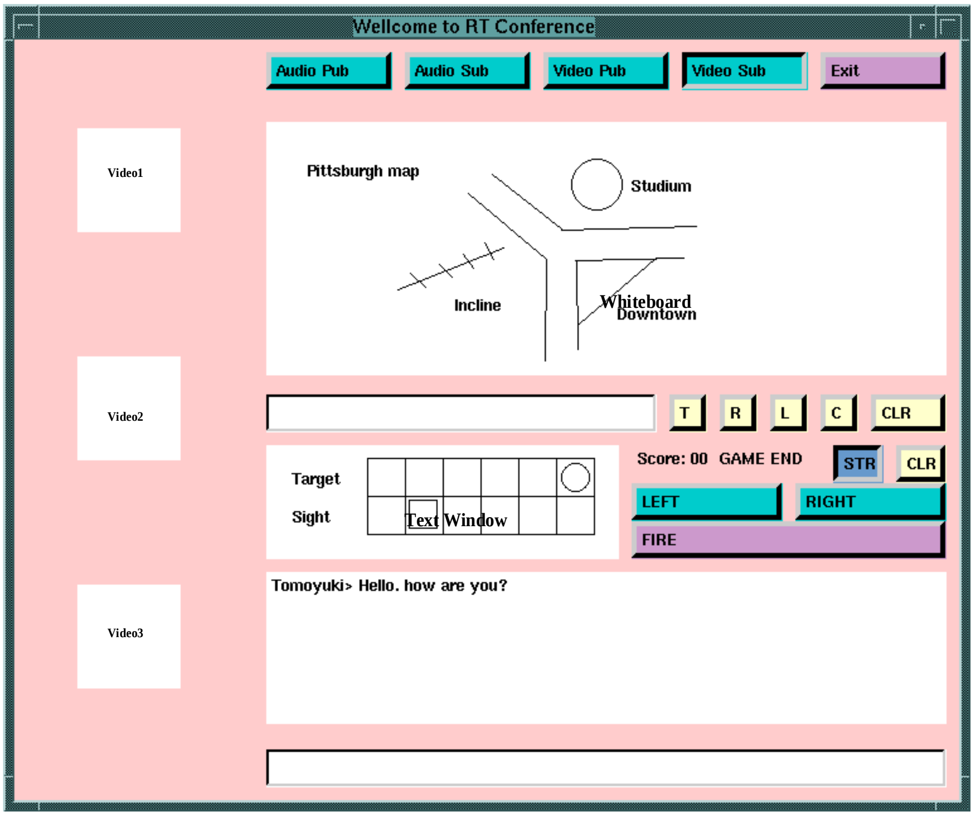

Figure 8: *RT-Conference: A Multi-party Multimedia Collaborative System on Real-Time Mach*

The implementation of the real-time push-pull service described in the previous two sections is relatively complex due to the various possible configurations that must be supported. However, the elegance of the model lies in its power of many-to-many communications and the simplicity that it provides in terms of transparency of underlying nodes, networks and communication protocols. We therefore wanted to exercise this abstraction to validate its usability and ease of programming a real application. From that perspective, we built a multi-party multimedia collaborative system named *RT-Conference* [23] on top of this model.

The graphical user interface of *RT-Conference* which is used to process the requests and displaying the responses and video is shown in Figure 4. This system supports the following features:

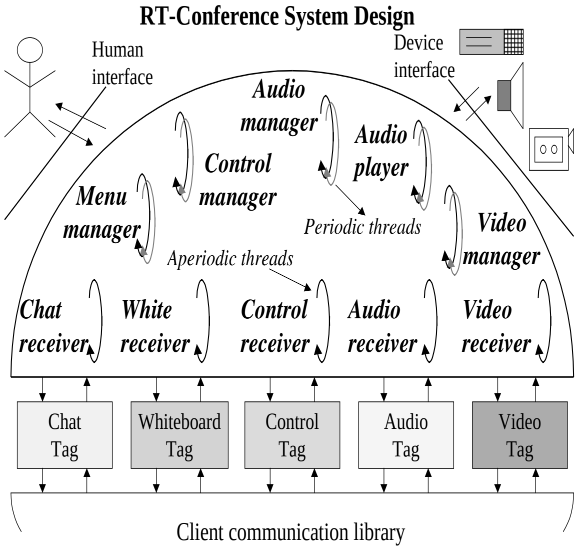

Figure 9: *The Architecture of the Multi-Party Collaborative System.*

4.1 The Architecture and Components of *RT-Conference*

The internal architecture of the conferencing system is shown in Figure 9. There are a total of five tags in the system, with a tag being assigned to each media type. This has two advantages. First, clients can deal with different multimedia data types in different ways and with different timing requirements. Secondly, a user can choose to publish on and subscribe to any media subset within the conference. For example, low bandwidth clients might opt only for audio and not publish or subscribe to any video data. High bandwidth clients can choose all available streams. Some passive "pull clients" might want to pull the white-board on demand to sporadically watch the progress of a design, a chess game or a discussion. The various threads in the client, along with their functions and their priorities, are listed in Figure 10.

The components of RT-Conference are explained below.

- Audio Publish: Clients who wants to publish audio initiate the request by pressing the push button. An *audio_tag* is created if it does not exist. A full-duplex Vibra-16 (Sound Blaster) audio driver captures audio and plays back in real-time. It schedules DMA operations in "auto-init" mode to achieve full duplex operation. The details of the driver design [24] are not relevant to the paper. A full duplex audio server handles the record and play for multiple audio clients. The multi-party conferencing system interacts with the audio server to capture audio and playback.
- Audio Subscribe: Client receives audio data by subscribing to *audio_tag*, which inturn initiated by clicking the push-button.
- Video Publish:Clients who wants to publish video initiate the request by pressing the push button. An *video_tag* is created if it does not exist.
- Video Subscribe:Client receives video data by subscribing to *video_tag*.
- White-board: A shared "white-board" is provided for for drawing text and graphics. White board data is published on *whiteboard_tag* and is tranmitted all clients.
- Text Chat:Text chat data is published on *chat_tag* and is tranmitted to all clients.

Twelve real-time including two library threads provide the communication services. These threads have different real-time priorities. A `network receiver` receives the tag operations. An `application receiver` receives the application data. A `whiteboard receiver`, a `chat receiver`, a `control receiver`, a `video receiver` and an `audio receiver` thread receive the published data on their respective distribution tags.

We now explain audio mixing and other related features of the audio clients:

Name	Main function	Relative Priority
Network receiver	Receives all communication data from local daemon and updates the tag table.	10
Application receiver	Receives all application data from local daemon.	2
Chat receiver	Receives text data for chatting and display it to the window.	9
Whiteboard receiver	Receives data for whiteboard and draw figures to the window	7
Control receiver	Receives data for real-time data application and changes the status on the screen.	2
Audio receiver	Receives audio data and buffers them until the period to replay comes.	3
Video receiver	Receives video data, assembles a video frame, and displays it on the screen.	5
Menu manager	Manages the event from a user on the screen, and sends data if necessary.	2
Audio manager	Receives audio data from microphone periodically and sends it to subscribers.	3
Video manager	Receives video data from video camera periodically and sends it to subscribers.	5
Control manager	Changes the status of application and sends this information to all members.	2
Audio player	Mixes audio packets into one packet and passes it to the speaker periodically.	3

Figure 10: *The client threads, their functions and their priorities.*

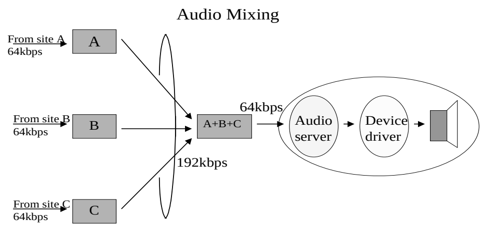

Figure 11: *The Mixing of Multiple Audio Input Streams to Speaker.*

- **Audio Mixing:** The system uses an audio sampling rate of 8 KHz and 8 bits per sample. A block of length 256 bytes is used for record and play. Audio mixing is achieved by simply adding the PCM samples. Details of the mixing algorithm can be found in [23] and are illustrated in Figure 11.
- **Amplitude Control:** One practical issue that must be dealt with in the context of multi-party audio communications is the issue of audio amplitude control during audio mixing. If the number of audio sources is large and they must be mixed together, the system must decide how much "weight" is given to each audio stream. If all streams are treated as is and summed up, it is possible that the resolution of the audio on the mixer side is exceeded, overflow occurs, and the resulting audio will be a complete distortion of its inputs. *RT-Conference* adopts the simple scheme of scaling each audio stream down by the number of current publishers on the audio tag. This summing based on the number of receivers (R) is illustrated in the top of Figure 12. We have also experimented with tracking in real-time the number of "speakers" who are talking at any given time. That is, even though 3 people may be participating in the conference, only one may be talking. It may be desirable to ensure that a single person's voice is heard distinctly and their audio volume is not turned down. A state transition scheme tracks the number of speakers in real-time, and is based on a check of overflows happening within an audio block. This scheme is illustrated in the bottom half of Figure 12.
- **Silence Detection:** Dealing with delayed audio packets is an important feature of the system. Since audio packets are sent from different publishers through the network, they do not arrive at the subscriber site

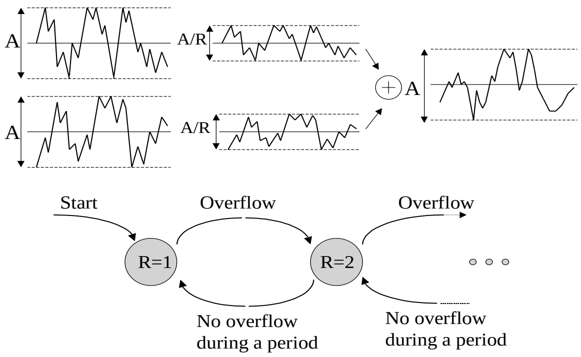

Figure 12: *Amplitude Control to Avoid Signal Overflow. The top half depicts the amplitude scaling for R received streams. The bottom half illustrates the real-time tracking of R.*

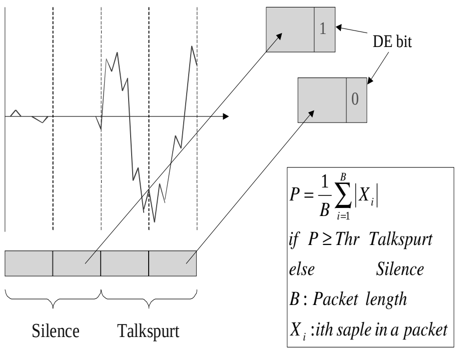

Figure 13: *Silence Detection Mechanism.*

synchronously. In order to keep audio quality above a certain level, the audio from different publishers should be played back with small delay and delay jitter. To maintain audio quality and to reduce a demand on system resources with tight timing requirements, a silence detection mechanism has been implemented as shown in Figure 13.

Since all participants do not always speak at the same time, each person's speech is divided into two parts, "`silence`" and "`talkspurt`". If the average of the absolute values of the audio samples in a packet, P, is greater than a given threshold, Thr, the packet is considered to be in the `talkspurt` and if P is less than Thr, then the packet is considered to be in the `silence`. Because packets in the `silence` are less important than packets in the `talkspurt`, a `discard-eligible bit` (DEbit) is set in the header of `silence` packets, which can be discarded at the receiver site if necessary. We experimentally found that $Thr = 1$ is enough to distinguish between `talkspurt` and `silence`.[4] At the receiver site, each user is assigned a set of memory buffers. An *Audio Player* thread periodically picks up a packet stored by an *Audio Receiver* thread from each buffer, mixes them into one packet and passes it to the audio server. However, if the number of packets in this buffer exceeds the maximum number of `silence` packets `Max_S`, the *Audio Receiver* starts discarding `silence` packets. If the number of packets in the buffer exceeds the maximum number of `talkspurt` packets, `Max_T`, the *Audio Receiver* starts discarding the `talkspurt` packets.

[4] In a deployed context, however, the threshold may be higher in order to mask out some background noise.

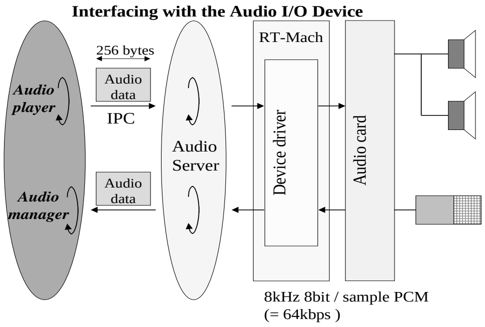

Figure 14: *The Interface to Audio Device Input/Output.*

- **Audio Device Interface**: As mentioned earlier, full-duplex audio is supported in RT-Mach by a user-level server which can provide audio services to multiple local clients simultaneously. This is illustrated in Figure 14.

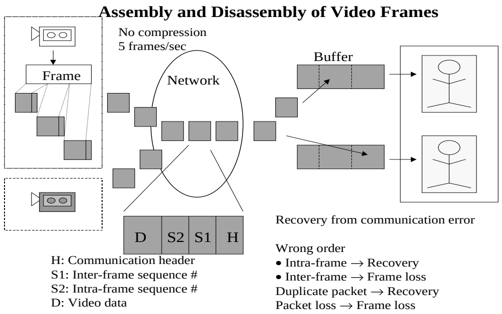

Figure 15: *The Assembly and Dis-assembly of Video Frames.*

We now explain the communication scheme for the relatively large video packets and other related features of the video and game clients:

- **Communication of Video Frames:** Since *RT-Conference* publishes raw video at this point, the size of each video frame (at least 80x80) is larger than the size of a single packet. Each frame is therefore fragmented into multiple packets and then published. The assembling and disassembling of packets is done using intra-frame sequence number and inter-frame sequence numbers, and is illustrated in Figure 15. Any communication errors (such as dropped packets, delayed or out-of-order deliveries) are also taken care of by this subsystem.
- **Video Device Interface**: The video input/output interface used by *RT-Conference* is illustrated in Figure 16. Contrary to audio, the video output path communicates with the X-server for display on the monitor.
- **Real-Time Video Game**: The real-time networked video game that the *RT-Conference* participants can play is illustrated in Figure 17. The objective of the game is to shoot a randomly moving target in collaboration with one another. The person who starts the game gets to move a gun-sight towards the moving target, and the others get to fire at the target. A finite amount of shots can be taken and the game expires after a fixed amount of time. The number of times the target was hit is the score gained by the participants.

4.2 Lessons Learned

We learnt several lessons during the design and implementation of the real-time push-pull layer and the *RT-Conference* system. We summarize some of them below:

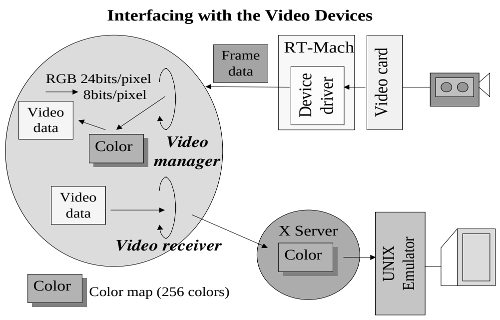

Figure 16: *The Interface to Video Input/Output.*

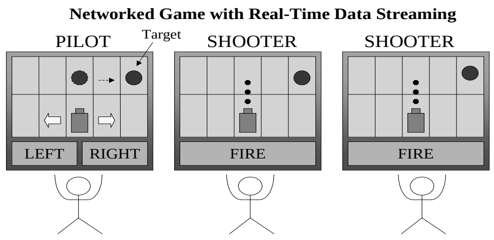

Figure 17: *The Networked Video Shooting Game for Real-Time Data Streaming.*

- *Push-Pull*: The push-pull communications service made the programming of the distributed portions of the system rather easy enabling seamless communication of the various streams. Actually, the (correct) use of UDP/IP in the underlying communication layer of the push-pull model even had an unexpected side benefit. Recently, during a demonstration where *RT-Conference* was run over a real-time network which offered bandwidth guarantees, an operator error brought down the network. When the network bandwidth got re-established, all video and audio streams recovered without any intervention and *RT-Conference* resumed its functioning .
- *Flexibility:* The model is very flexible and can support both hard real-time and soft real-time applications. Several applications with different QoS requirements can coexist in the model at the same time. Model clearly distinguishes non real-time actions from real-time actions.
- *Priority and reservation management*: In a system such as *RT-Conference*, the priorities (or the choice of reservation periods which in turn dictates the priorities) of the various threads play a critical role. Each data type in this system has different semantics to the user, and different timing characteristics. Audio is very sensitive to jitter and is significant for interactive communications. It is therefore relatively easy to assign audio the highest priority in the system. The real-time data stream of the video game was assigned the next highest priority. The video thread was assigned the next highest priority followed by the white-board and the chat window. However, other combinations of priorities may also work depending upon available system resources and the expected frequency of usage of some data types. Hence, these parameters may actually need to be offered as options to the end-user(s).

  In addition to the threads within the client application, the underlying communication service threads must also cooperate with the application threads with appropriate priorities. It is useful to note here that daemon threads and client threads must coordinate their priority levels, else one or the other would suffer. Such situations are the logical candidates for abstractions like processor reserve, disk and network reserves [7], which provide timeliness guarantees *and* temporal protection from misbehaving threads.

- *System bottlenecks*: While video clearly consumes a good portion of the system's bandwidth, our experience is that the network stack can also play a significant role. In particular, if audio data is being processed in very small blocks (for obtaining good end-to-end latencies), the costs of the communication protocol stack and context switching overheads can also contribute to significant overhead. We used a block size of 256 bytes for the audio stream. This imposes an acceptable overhead on Pentium 150MHz workstations but our past experience is that the corresponding overhead can be rather high on i486 66 MHz machines.
- *Silence detection*: We also found that for human conversations, about 90% of the audio packets can actually be filtered away by silence detection. This not only acts as an excellent context-sensitive compression scheme, but also greatly ameliorates the problem stated in the previous item.
- *Thread safety and X-11/Unix:* The biggest problem that we faced was one of thread safety. X-11 library calls and Unix system calls are both being used to support the *RT-Conference* GUI, but neither is thread-safe in the context of RT-Mach. This posed our biggest problem and was eventually solved only by the judicious introduction of `mutex` locks and `select()` statements to ensure that at most one of the thread-unsafe events was happening at any given time.

5 Conclusions

The real-time push-pull communications model extends the synchronous real-time publisher/subscriber model in two significant ways. One, it allows information sinks to access recently published data samples asynchronously on demand. Two, it supports a heterogeneous environment by allowing low-bandwidth, low-power computing nodes to participate in ongoing communications. This distributed communications service has been successfully implemented as a middleware layer on top of RT-Mach, a resource-centric kernel which provides QoS support. These services can be also invoked from CORBA [18]. We have presented some end-to-end measurements. We have studied performance of the model in different configurations. An interesting multi-party collaboration test-bed has been built on top of this model. Providing more flexible QoS guarantees for each data channel, intelligent and load-balancing positioning of proxies, embedding proxies in network elements for custom data scaling, support for high availability, etc. can be interesting future directions.

References

[1] Ken Birman, Roy Friedman, Mark Hayden and Injong Rhee. Middleware Support for Distributed Multimedia and Collaborative Computing. SPIE International conference on Multimedia computing and Networking,'98, San Jose, USA.

[2] Robert van Renesse, Ken Birman, Thorsten von Eicken and Keith Marzullo New Applications for Group Computing In Theory and Practice of Distributed Systems, Lecture Notes in Computer Science, Vol.938.

[3] G. Robert Malan, Farnam Jahanian, and Sushila Subramanian Salamander: A Push-based Distribution Substrate for Internet Applications. Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997; Monterey, California.

[4] Raj Rajkumar, Mike Gagliardi and Lui Sha The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation In Proceedings of the IEEE Real-time Technology and Applications Symposium, June 1995.

[5] Mike Gagliardi, Raj Rajkumar and Lui Sha Designing forcEvolvability: Building Blocks for Evolvable Real-Time Systems. In Proceedings of the IEEE Real-time Technology and Applications Symposium, June 1996.

[6] Raj Rajkumar and Mike Gagliardi. High Availability in The Real-Time Publisher/Subscriber Inter-Process Communication Model. In Proceedings of the IEEE Real-Time Systems Symposium, December 1996.

[7] Raj Rajkumar, Kanaka Juvva, Anastasio Molano and Shui Oikawa. Resource Kernels: A Resource Centric Approach to Real-Time Systems. In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.

[8] Douglas C. Schmidt, Daid L. Levine, and Sumedh Mungee. The Design of the TAO Real-Time Object Request Broker, In Computer Communications Journal,Summer 1997.

[9] Anastasio Molano, Kanaka Juvva and Raj Rajkumar. Real-Time Filesystems:Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In Proceedings of the IEEE Real-Time Systems Symposium, December, 1997.

[10] Anastasio Molano, Raj Rajkumar, Kanaka Juvva. Dynamic Disk Bandwidth Management and Metadata Pre-fetching in a Reserved Real-Time Filesystem. In Proceedings of 10th Euromicro Real-Time Workshop.

[11] V. Fay Wolfe, L.C.DiPippo, R.Ginis, M.Squadrito, S. Wohlever, I. Zykh, and R. Johnston Real-Time CORBA. In Proceedings of the Third IEEE Real-TIme Technology and Applications Symposium, (Montreal Canada), June 1997.

[12] J.A.Zinky, D.E.Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objetcs Theory and Practice of Object Systems,vol. 3, No. 1, 1997.

[13] Talley, T.M., Jeffay, K. Two-Dimensional Scaling Techniques for Adaptive, Rate-Based Transmission Control of Live Audio and Video Streams. Proc. Second ACM Intl. Conference on Multimedia, San Francisco, CA, October 1994, pp. 247-254.

[14] Peter Nee, Kevin Jeffay, Gunner Danneels. The Performance of Two-Dimensional Media Scaling for Internet Videoconferencing. In Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video, St. Louis, MO,May 1997.

[15] van Renesse, R., Birman, K.P., and Maffeis,S. Horus: A Flexible Group Communication System. Commun ACM 39, 4 (April 1996).

[16] Maeffis, S. Adding group communication and fault-tolerance to CORBA. In Proceedings of the 1995 USENIX Conference on Object- Oriented Technologies (Monterey,calif.,June 1995).

[17] L.E.Moser,P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia and C.A. Lingeley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. Commun ACM 39, 4 (April 1996).

[18] Amir, E., McCanne, S., and Katz, R. Receiver-driven Bandwidth Adaptation for Light-Weight Sessions. Usenix-97.

[19] Flaviu Cristian, Frank Schmuck. Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems. UCSD Technical Report CSE95-428, 1995.

[20] M. Clegg and K. Marzullo. A Low-Cost Group Membership Protocol for a Hard Real-Time Distributed System. In Proceedings of Real-Time Systems Symposium, San Francisco, California,December 1997,pp. 90-99.

[21] A. Fox, S.D.Gribble,Y. Chawathe,E. Brewer, P. Gauthier. Cluster-Based Scalable Network Services. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles(SOSP-16), Saint-Malo, France, October 1997.

[22] Alan Fekete, Nancy Lynch, Alex Shvartsman. Specifying and Using a Partitionable Group Communication Service. in 1997 PODC97, Santa Barbara, CA, pp. 53-62.

[23] Tomoyuki Ueno. The Design and Implementation of a Collaborative Conferencing System on Real-Time Mach. Master's Thesis Report, 1997-03, Information Networking Institute, Carnegie Mellon University.

[24] SoundBlaster-16 Programmer's Guide, Creative Labs, 1996.

[25] Flaviu Cristian. Synchronous and Asynchronous Group Communications. in IEEE Workshop on Fault-tolerant and Parallel Distibuted Systems, Honolulu.

[26] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A QoS-based Resource Allocation Model In *Proceedings of the IEEE Real-Time Systems Symposium.* December, 1998.

[27] Lehoczky, J. P., Sha, L. and Ding, Y. The Rate Monotonic Scheduling Algorithm — Exact Characterization and Average-Case Behavior. Real-Time Systems Symposium, Dec, 1989.

[28] Joseph, M. and Pandya. Finding Response Times in a Real-Time System. The Computer Journal (British Computing Society,(29) 5:390-395, October, 1986.

[29] Tindell, K. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Technical Report YCS189, Department of Computer Science, University of York, December, 1992.

—**oOo**—

Appendix A

The two key data structures used by our real-time push model are listed below. The comments on each field are self-explanatory to some extent.

```
typedef struct{
tagID_t id;              /* Numerical Identifier */
tagID_str_t id_str;      /* String Identifier */
u_int primary;           /* IP Address of Primary Node for this tag */
u_int backup;            /* IP Address of Backup Node for this tag */
UserList pList;          /* List of Publishers on this tag */
UserList sList;          /* List of Subscribers to this tag */
u_int seq_no;            /* Sequence# for a publisher */
u_int active;            /* Flag indicating activity on a tag */
} Tag;

typedef struct {
u_int pid;               /* Process ID */
u_int addr;              /* IP Address */
u_short port;            /* Port listened on */
Ipc rt_port;             /* Port for ipc */
u_int type ;             /* Push Subscriber/Pull Subscriber */
u_int freq;              /* Frequency for proxy */
u_int proxy_location;    /* Proxy/Pull location for the subscriber */
u_int proxy_ipaddr;      /* IP address push/pull applies only to mode 3 */
u_int noMessages ;       /* No of pull messages */
u_int pub_pid ;          /* pull publisher id */
u_int pub_addr ;
} User;
```

Figure 18: Key Data Structures Used.