35

# Scalable Distributed Resource Location

by

Scott T. Smith

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998 [June 1998]



Author . .....
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by .....
Andrew Lippman
Associate Director, MIT Media Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

# Scalable Distributed Resource Location

by

Scott T. Smith

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

## Abstract

With more and more computers being constantly connected to the internet, with increasingly larger hard drives, and with network bandwidth not keeping up, caching is quickly becoming a necessity. A protocol is needed to find copies of objects that are near to you, with a minimal impact on the overall CPU usage of all the computers on the network and a minimal impact on the network traffic. This thesis describes an algorithm that attempts to find copies of objects while optimizing resource usage.

Thesis Supervisor: Andrew Lippman
Title: Associate Director, MIT Media Laboratory

# Acknowledgments

First of all, thanks to Andy Lippman, my advisor. Without his support this never would have been possible. And without his insistance, this thesis might never have been written!

Thanks to Roger for his help in understanding HPRTP. If I had not gained the insider knowledge of the system, this project would have been much more difficult.

Thanks to Matt, Yuli, Chris, and Cole for providing (and requiring) relaxing breaks during my "crunch" period. Thanks to them, my stress level never rose much above average.

# Contents

# List of Figures

# Chapter 1

# Introduction

Computers are slowly turning into our universal device for retrieving and viewing information. As Cable companies deliver more and more advanced features from our cable boxes, such as online programming schedule and video on demand, computers seem to be the next logical step in set top boxes. A computer can replace the obsolete cable box with a more advanced and customizable programming guide, replace VCRs by using hard drives as caches, and provide true video on demand. For example, if you arrive home 5 minutes late to watch your favorite TV show, then you should be able to tune in to the remaining 25 minutes while requesting the first 5 minutes be retransmitted. HPRTP[4] provides a mechanism for this exact scenario to be handled gracefully and with a minimal overhead on the network.

It would also be nice to extend the features of this system to allow the retrieval of objects even if they are not currently being broadcast. Thus even if you return home after the show has finished transmission, you could still be able to find a copy of the object in someone else's cache, and retreive the data from their computer instead of having to go back to the original source. This thesis is about designing a protocol to find the nearest copy of an object with a minimal network overhead that works in the framework of HPRTP.

## 1.1 Thesis Overview

There was a fair amount of work to be done to bring HPRTP up to the state where a cached object locator would be useful.

**Chapter 2** explains some fundamentals of HPRTP.

**Chapter 3** explains some of the work done to port HPRTP to the Windows platform. Since the majority of computers use Windows, demonstrating the ability to use this advanced protocol on a common platform shows that this is not a distant dream, but realizable using today's technology.

**Chapter 4** explains the application developed for HPRTP to demonstrate it's effectiveness.

**Chapter 5** discusses the object location protocol which allows currently cached, but not active, objects to be located and reopened.

# Chapter 2

# HPRTP

HPRTP[4] combines a reliable network protocol with a caching subsystem to allow the seamless access of media related information, whether it be located on your computer, on another computer, or if it is currently being broadcast. By incorporating the cache together with the network protocol, the system allows other computers to query the contents of the cache without any intervention from the program that put the contents there in the first place.

As is illustrated in Figure 2-1, HPRTP sits between the application and the disk and network. Any requests for information broadcast or delivery is sent to HPRTP. It then takes care of broadcasting to multiple receivers on the network, handling repairs on lost packets, and if appropriate, caching to the local disk for future access.

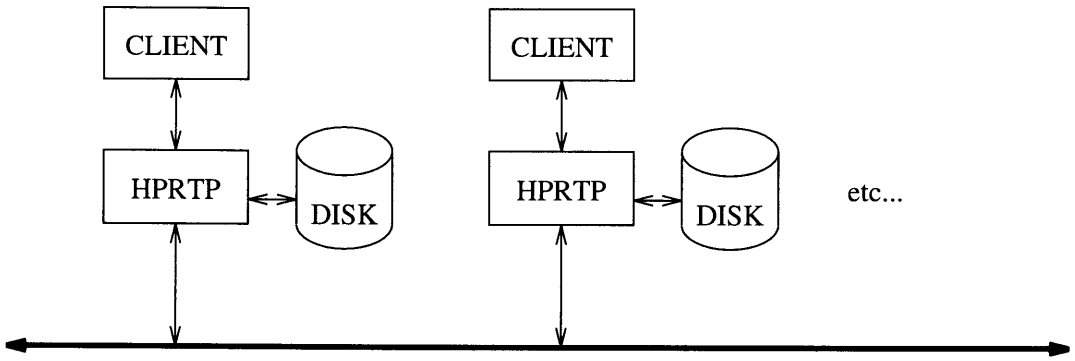



Figure 2-1: HPRTP System Diagram

HPRTP also accepts requests from the network. While the application is receiving a particular data stream, HPRTP can simulatenously serve another part of the

data stream to a different computer. In the event another computer loses a part of the stream, HPRTP can send a repair packet to that computer. The fact that any computer participating in a given data stream can respond to repair requests is a distinct advantage of HPRTP because it reduces the load on the original server.

## 2.1 Reliable Multicast

HPRTP uses multicast IP to deliver the data stream to multiple recipients with a minimal load on the network. Several techniques were added to fix the unreliable nature of multicast.

One is called FEC, or Forward Error Correction. This technique essentially transmits redundant data, under the assumption that if a packet is lost, its information can be inferred from the other packets. This is similar to parity checking used in RAM, RS-232 communications, or RAID.

The other technique is called NACK, or negative acknowledgements. When a client discovers it is missing a piece of data, it will send a negative acknowledgement, which means that it did not receive the data it was supposed to. Another computer will respond to this NACK with the missing data. This is similar to what happens in a conversation when you say "I'm sorry, could you repeat that?" This technique is the opposite of that used in TCP, which uses positive acknowledgements. Positive acknowledgements are sent for each piece of data that is properly received. This is similar to saying "Uh huh" or "Yeah" while another person is talking to you. Both methods work, but negative acknowledgements scale better with the number of receivers.

## 2.2 Dynamic Partitions

In addition to delivering a reliable stream efficiently to multiple receivers, HPRTP servers can create additional late join streams called dynamic partitions. This is useful if one attempts to view a live broadcast after it has started. For example, if you get

home 5 minutes late for your favorite TV show, you can receive the last 25 minutes of it by listening to the current broadcast. Additionally you can request the first five minutes. This is called a late join because you join the current live broadcast late. A neighboring HPRTP server that has the data cached can open a dynamic partition to serve this data to you. HPRTP integrates the two streams together and returns them as one uniform stream to the application that requested the data. Thus the application does not need to worry about the details of caching the live stream while receiving the second stream; HPRTP takes care of all the details.

## 2.3 Application Data Units (ADU's)

HPRTP gives a mechanism to assist applications in segmented their data. It resembles multidimensional arrays: the application specifies the number of "dimensions" and the maximum value for each one. For example, audio data might be segmented into hours, minutes, seconds, and samples, with maximum values of 24, 60, 60, and 44100 for one days worth of sound sampled at 44.1kHz. When the application gives data to HPRTP, it sends along with it the ADU range the data represents. Additionally, when initially describing the object, the expected rate of ADU's is given so that HPRTP knows how quickly to retransmit the data in the event it is required to serve a dynamic partition.

## 2.4 Object Control Channels

HPRTP uses two object control channels, the Logical Data Stream Control Channel and the Intra-Object Control Channel, to communicate information describing the object. NACK's, repairs, a list of available streams for an object, and a description of an object including expected bandwidth are all transmitted among one of the two channels. All HPRTP servers involved with the object with listen to the IOC, and it will listen to the LDS of each stream it is taking part in. By only listening to the channels pertaining to the data the server is handling, HPRTP can somewhat

minimize the excess traffic it must listen to.

## 2.5 Potential Applications

The capabilities of HPRTP make it well suited to be the digital media delivery mechanism for packet switched networks such as the internet. It is capable of serving broadcast television shows, movies, radio, and phone or video conferencing.

# Chapter 3

# Porting HPRTP

In order to make HPRTP usable on a variety of systems, an effort was made to port it from the Digital Unix development environment to Windows NT. During this process, a number of cross platform issues were brought up, both because of the processor and because of the operating system.

Processors can differ in several aspects: the instruction set, which is the set of commands the processor can interpret and run; the word size, which is the range of integers the processor can efficiently deal with; and the byte order (also referred to as endian), which is the order the bytes within a word are ordered when stored in memory. The instruction set differences are masked by the compiler. The Windows NT compiler was also able to emulate the 64-bit word size of the Alpha on the 32-bit Intel processor. Lastly, the byte order of the two machines is the same, although HPRTP is designed to use the universally accepted network byte order (big-endian) when communicating over the network.

Differences in the operating system proved to be the most difficult obstacle to overcome. While the compilers under Windows NT and Digital Unix both support ANSI standards, there are many advanced features of each operating system that work quite differently. These functions include multithreading, mutexes, semaphores, conditional events, multicast network support, and shared memory. To make the design of HPRTP as modular as possible, a common API implemented on both platforms was developed.

## 3.1 Portable Library

There were two main versions of the portable library, which supported the common API. The first was Windows NT, also known as Win32. The second was for Digital Unix and supported major standards such as POSIX and System 5. This library was able to compile under Linux with no changes, and would likely work under other POSIX and System 5 compliant Unices. The functions calls of this library are described in Appendix A.

### 3.1.1 Multithreading

Both operating systems support multithreaded programs. Digital Unix provides this through support for the POSIX standard while Windows NT has a unique API. Given that our needs were simple enough, the only functions that were implemented was one to create a thread and one to terminate the current thread.

### 3.1.2 Semaphores

There is support for global semaphores in both operating systems; Windows provides a reference by name and Unix provides a reference by a randomly generated number. The lowest common denominator chosen was a randomly generated number. A mechanism was needed for Windows to grab a handle to the semaphore given a number. A C++ class was used to encapsulate this abstraction quite easily. However, this C++ class had to reside in shared memory and be used by multiple processes. Since handles are unique to each process, a lookup table was created in private memory that allowed the common class to associate the random number with a local copy of the handles.

### 3.1.3 Mutexes

Mutual exclusion objects, or mutexes, are used in multithreaded programming to ensure only one procedure or thread has access to an object at any given time. Mutexes

differ from semaphores because the thread that locks the mutex actually owns it, and future calls to lock the mutex will succeed as long as the calling thread owns the mutex, even if it is already locked. Mutexes were used in HPRTP, but only locally. While it is possible to use the global semaphore mechanism to implement local mutexes, it can be more efficient to use some other mechanism. Windows supported unnamed mutexes, which were assumed to be local, while the POSIX threads specification described its own mechanism.

### 3.1.4 Conditional Events

Windows supports both local and global conditional events, while Unix only supports local conditional events. Windows supports events where the waiting threads can either be signaled one at a time, or all at once; however the same event cannot be used for both. Unix allows events to be signalled either way depending on the desired result at the time. HPRTP needed local events that could be signalled either way, but since the broadcast signal was only used when shutting down HPRTP, a hack was made which just continuously signalled the event until it was known that all threads received the signal.

### 3.1.5 Multicast Network Support

The multicast support under Windows NT works differently than under Unix.

#### Loopback

Under Windows you cannot disable multicast loopback. This option is designed to prevent a program from receiving packets that it sends out to the multicast group. A filter was set up to automatically discard packets whose originating address was the current computer. Unix can automatically do this filtering.

#### Bind vs Join

Under Windows you cannot bind to a multicast address. It is assumed that after joining the group, all packets sent to the address will be received. Under Unix, you can additionally bind to the multicast address. The benefit of this bind is unclear.

#### Order of Initialization

Under Windows, you must join a multicast group before you can initialize the time to live for the socket. Under Unix, the opposite occurs.

### 3.1.6 Shared Memory

Shared memory was fairly easy to implement on both platforms. While the function calls were slightly different, the supplied functionality was similar enough to provide the same capabilities on both machines.

### 3.1.7 Clock

The default clock resolution under Windows running on Intel processors is 16 ms (60 Hz), while the Alpha workstations have a resolution of 1 ms. HPRTP is extremely sensitive to the clock, and uses it extensively to determine when to expect data to arrive on a channel, when to schedule requests, when to send data out, and much more. The default resolution under Windows was inadequate for these purposes, so it was increased. HPRTP is also dependant on having all the computers synchronized to a common clock. NTP[5] was used to perform this action.

# Chapter 4

# Development of CD player

HPRTP is a great protocol for distributing media across a wide area network, but it needs an application to prove its effectiveness and demonstrate its superiority over other protocols. Given the proliferation of cheap networkable computers with built in CDROM drives and sound cards, we chose an audio application. Possible real world applications include putting all radio stations online such that one can tune into any radio station from around the world (such as RealAudio[2]) or a conference call system (such as Reflection of Presence[3].) I chose to implement a CD jukebox, where you could put a CD in your computer and play it or a CD from anybody elses computer. The program read the digital data from the CD and broadcasted it using HPRTP. It also included the option to broadcast MPEG compressed music to illustrate HPRTP's ability to handle compressed data streams. However, due to the fact that MPEG is an assymetric codec and cannot be encoded realtime on today's processors, all compression had to be performed offline.

## 4.1 CD Digitization

Most computer CDROM drives support reading audio data directly off the CD. This works well because the data read is lossless. The alternative would be to play the CD and record through the soundcard, but this would involve going through the CD player's DAC and the soundcard's ADC, resulting in a lossy translation through the

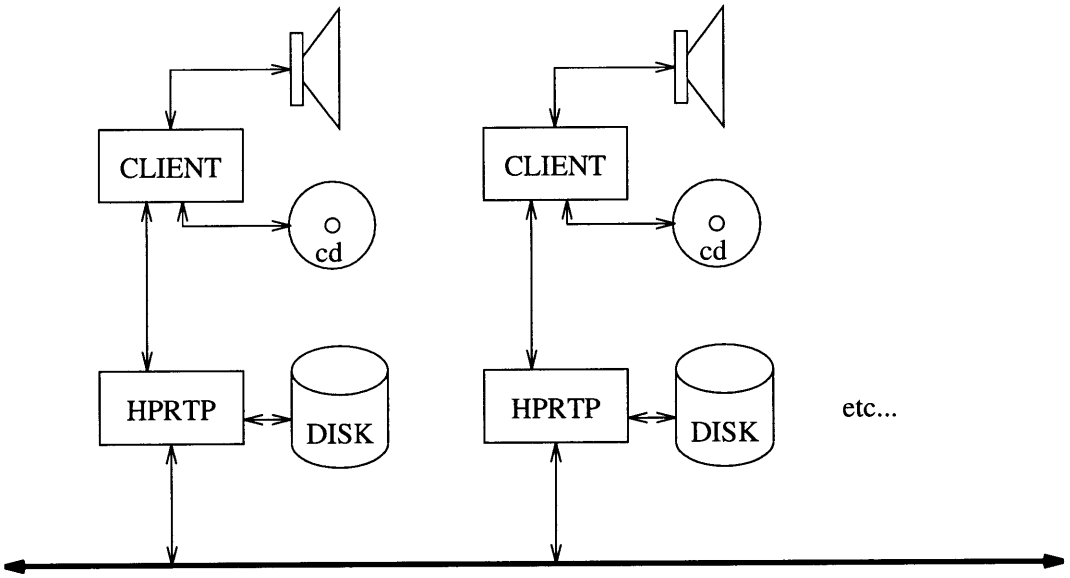



Figure 4-1: CD Player System Overview

analog circuit.

The main difficulty with digitizing CD's is the inaccuracy of the drive. Seeking to a particular location on some lower quality CDROM drives does not always seek to the exact sector, and is sometimes not even aligned on a sector boundary. When reading programs off a CD this is not a problem since each sector of 2048 bytes contains a header that the drive can look for. However, audio data sectors are 2352 bytes and do not have room for a header.

Given that seeks are sometimes inaccurate, each consecutive read must overlap by a small amount. Since the end of the first read overlaps with the beginning of the second read, this can be used to match the two reads together (see Figure 4-2). This wastes some time performing the extra read and memory compares, but given drives today running at 24 times the normal CD speed, it is not a problem. Even a 2x drive has no problem keeping up.

## 4.2 Audio library

To ensure portability, a separate interface was written to the audio devices. There is a portable library for Unix called AudioFile; a library providing my interface us-

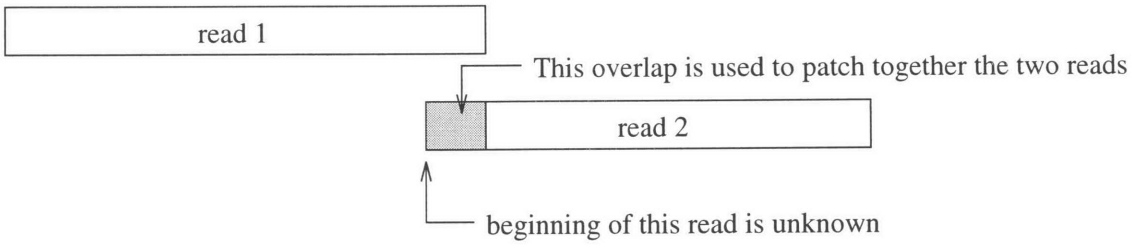



Figure 4-2: Two consecutive reads from CD

ing AudioFile was easy to write and allowed it to work on Digital Unix along with many others, in cast HPRTP is ever ported to other platforms. Writing a library for Windows was also very simple.

## 4.3 MPEG support

Transmitting the full CD bitrate across the network is fine for the ATM network we have in lab, but it isn't practical for the cable modems or DSL some people have or for the Internet backbone that must carry this traffic. To demonstrate that more reasonable data rates are possible, MPEG layer 2 and 3 audio decompression was added. Layer 3 allows a data rate of 256 kilobits per second to be used to achieve CD quality audio[1], thus giving a compression rate of 5.5:1. Soon to be released MPEG AAC compression will allow compression rates of 11:1 while still acheiving CD quality audio.

# Chapter 5

# Cached Object Locater

The mechanism to locate objects that was built into HPRTP used SAP and SDP, which are currently used on the multicast backbone. They allow the distribution of currently broadcasted streams, but an additional protocol is needed to locate objects that may be stored in caches but is not currently being broadcast.

I developed a protocol using multicast communication channels to find the nearest copy of an object. To support the use on a world wide network, a protocol involving servers as intermediaries is described.

## 5.1 Peer Multicast Protocol

When looking for the nearest copy of an object, it makes sense to first ask those computers near you first. This reduces the amount of extraneous requests computers must listen to, since they will only hear a request if there are no computers closer who have the object. The peer multicast subset of the cached object locator is based on the idea of asking neighboring computers to reply in a cooperative fashion first, gradually broadening the search criteria until the object is found.

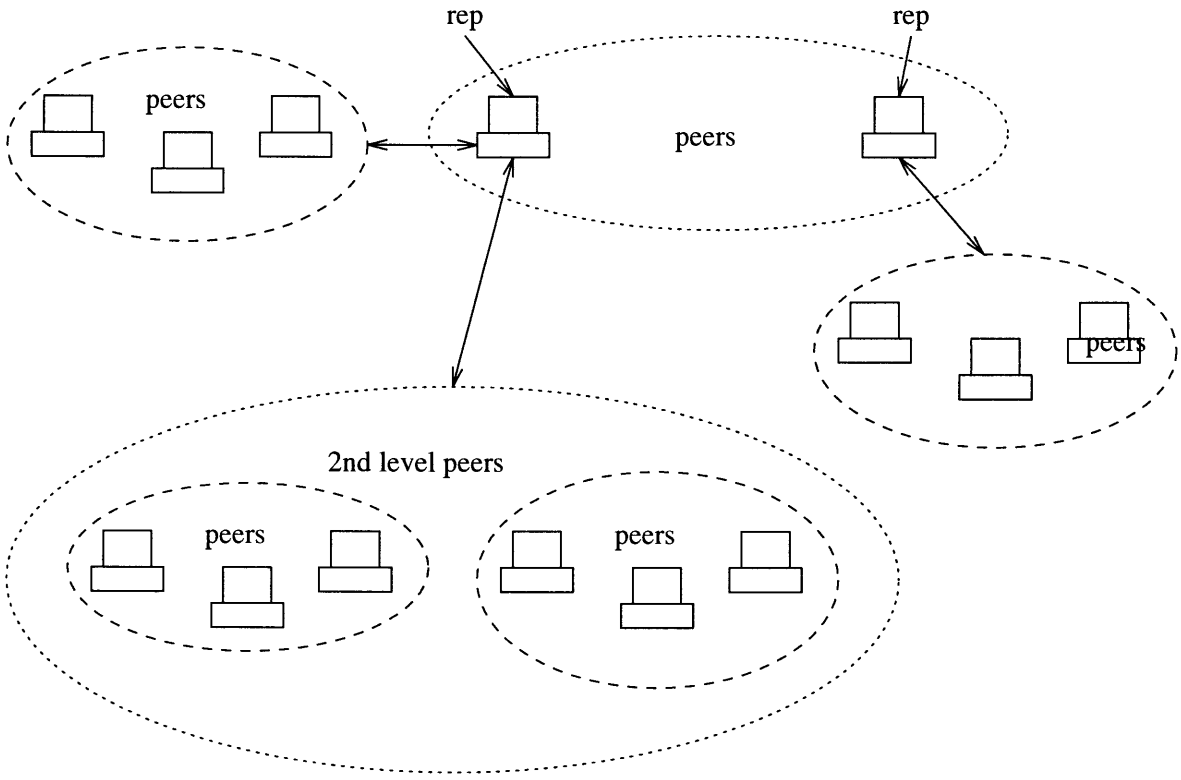



Figure 5-1: Example of a heirarchy with clients and representatives

### 5.1.1 Protocol description

When a program requests an object, HPRTP first uses the current mechanisms to see if it knows about the object. If it does, it can get the data without making a special request. Otherwise, a request for the object is sent out to a local group of peers. If no response is received, subsequently larger and larger groups are asked until either the object is found or all resources are exhausted.

#### Heirarchy setup

The protocol starts by asking small groups of peers first, and then moves on to larger and larger groups until it finds the object. This cuts down on network traffic if a nearby computer has the object, since only the local network will hear the request. It also assists in finding a local copy, since local computers are queried first.

Currently, a global configuration file, shared by all computers, is used to describe the hierarchy queries should progress through (see Figure 5-2). HPRTP parses this

file at startup and decides which groups it should listen in and reply to, and the order in which requests should be made. This simple mechanism allows a network administrator to centrally manage the heirarchy and distribute the information via NFS. Alternatively, the protocol could be expanded to allow it to communicate with a central server (denoted by a DNS entry) and fetch the configuration information from that server.

```
# (peer mcastaddress port scope ttl host1 or (peer ...) host2...)
(peer 239.255.255.254 9876 5 7
      (peer 239.255.255.253 9876 5 7 rocky-horror repo-man liquid-sky)
      (peer 239.255.255.252 9876 5 7 alphaville bad-taste barbarella))
```

Figure 5-2: Sample Heirarchy Configuration File

### Request

A request for the object is made when an application attempts to open an object that it does not have cached or a session announcement for. The server requests that a local computer that knows about the object join the IOC channel (see Section2.4). If the client makes a call to initialize the object for receiving, then the local server will query for information about the object on the IOC channel, such as the data rate.

When a request is sent to a group, the sender will wait for a reply. The amount of time it waits should be proportionate to the maximum roundtrip time between it and any other computer in the group. Empirically this works out to roughly 5ms within MIT campus, 30ms within the East Coast, and 130ms across the US. The actual timeout should be around five times larger to allow for the variable latency reply (see below). Larger groups should have longer delays to minimize the number of responders.

### Reply

When a request is received, HPRTP checks to see if it knows about the object. If it does and it is willing to serve the object, it schedules a response with a small random

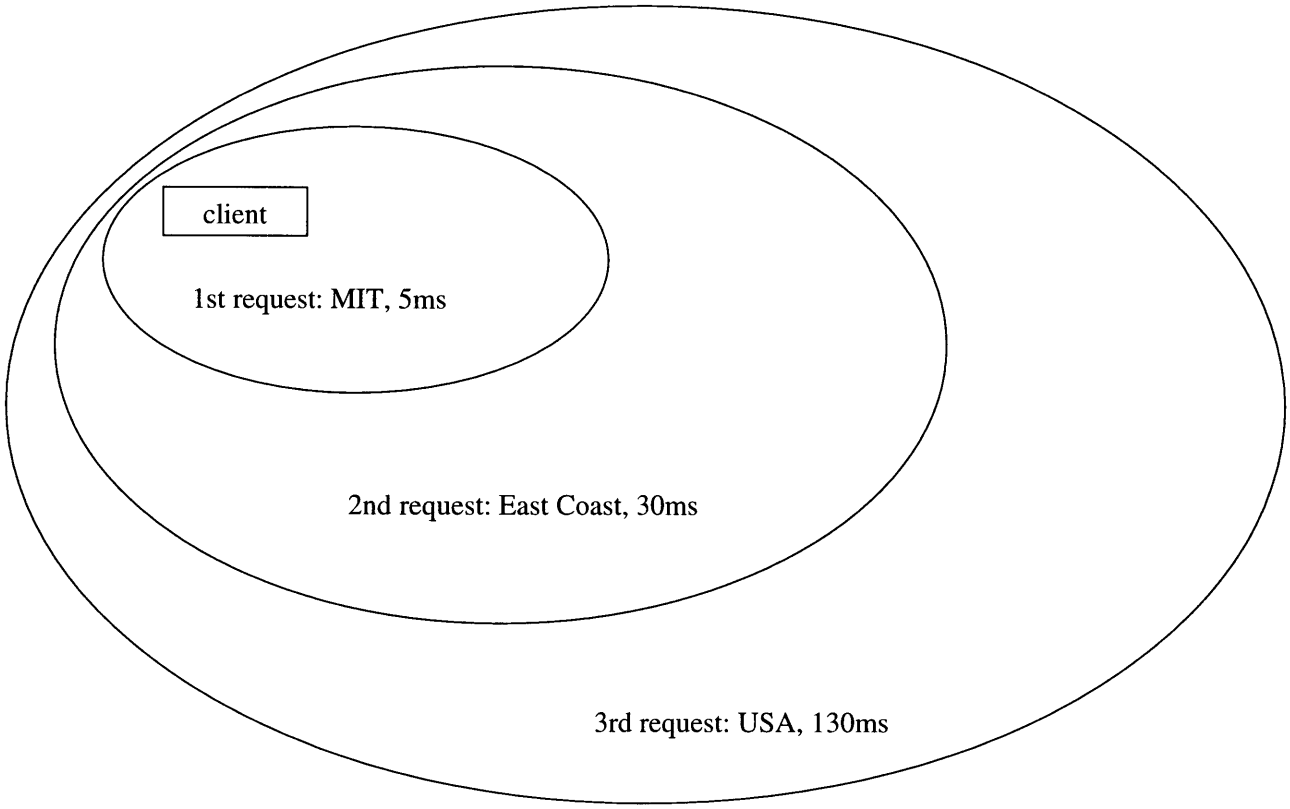



Figure 5-3: Example of a heirarchy with timings

delay. If during this delay it hears another computer responding, then it unschedules the response and does not join the IOC channel. This tecchnique is used in other areas of HPRTP[4]. This suppression technique reduces the number of responses and reduces the number of servers joining the IOC channel, thus reducing the impact of a request on the surrounding computers.

### 5.1.2 Analysis of traffic / scalability issues

Assuming a tree like structure where each successive group is $b$ times larger than the previous:

$o$=number of objects

$n$=number of objects in a local cache

$c$=number of computers

$\frac{c*n}{o}$=number of computers an object is cached on

$\frac{n}{o}$=probability an object is cached on this computer

$b$=number of branches on the tree

$\frac{\ln c}{\ln b}$=height of tree

$b^l$=number of peers on level $l$ (range 1..*height*)

$p = 1 - (1 - \frac{n}{o})^{b^l}$=probability object is in level $l$

$(b^l) * (\ln(1 - \frac{n}{o})) = \ln 1 - p$

$b^l = \frac{\ln 1-p}{\ln(1-\frac{n}{o})}$

$b^l = \frac{\ln 1-p}{-\frac{n}{o}} = \frac{-o*\ln 1-p}{n}$ approximation

$l * \ln b = \ln \frac{-o*\ln 1-p}{n}$

$l = \frac{\ln \frac{-o*\ln 1-p}{n}}{\ln b}$ l is which level is needed to achieve a probability $p$ that the object can be found.

This shows that in general, $\frac{-o*\ln 1-p}{n}$ people must be asked before finding a copy of the object. However this model for viewing does not take into account normal viewing practices, where a small number of objects are viewed by the majority, and a large number of objects are rarely accessed.

## 5.2 Representative Protocol

The multicast protocol works well for small groups of computers, but when the size gets much larger, then the amount of traffic on the channel can become quite large. To alleviate this, a representative protocol was designed. In this configuration, one or more dedicated servers handle all requests. This relieves the load from the clients.

### 5.2.1 Client Protocol Description

The representative acts as a multicast channel to which requests can be made. The clients will first ask all the peer groups it knows about, and then lastly ask the representative. At that point the representative will take over handling the request. It first checks all the objects it knows about; if it finds a match, it returns the location; otherwise it forwards the request up the heirarchy much like a client would.

Each representative stores information about all the objects stored under it in the heirarchy. Thus is serves as a proxy and a sort of barrier. Any requests coming from

outside the subtree governed by the representative are handled by the representative, rather than flooding all the computers with the request.

### 5.2.2 Object Registration Protocol

In order for the representative to know about all the objects underneath it, the clients must send messages to the representative describing which objects it has. These messages should be infrequent so that the network does not become flooded with messages about objects. Additionally, a message should be sent when the object is deleted from the cache to prevent stale data.

### 5.2.3 Multiple Representative Cooperation Protocol

To avoid a single point of failure, multiple representatives can work together to serve a subtree. When working together, the representatives can shuffle the load of requests around and check the availability of the other representatives. In the event a representative goes down, they can reconfigure themselves to rebalance the load.

In order for the representatives to be able to communicate with each other, a multicast address is used. This channel is used to communicate each representatives presence so a count can be kept, a request for a load balance, and to maintain consistancy in the object directory.

#### Existance Message

Once a second, an existance message is sent to the multicast address. This allows other representatives to keep a list of active representatives. This message also includes information on the number of requests it received from outside the subtree. The address range that this representative is assigned to is divided into four subranges, and separate tallies are kept for each group. By keeping these statistics on hand and readily available, reorganizations are easy to compute. The message also includes the number of objects currently known, so that consistancy among the object caches can be maintained.

### Consistancy Check

When a representative notices that another has a different number of objects, then a request for a consistancy check is sent out. When each representative receives a request, it sends an acknowledgement. At this point the representative goes into consistancy check mode (it will still handle other requests). If another consistancy request comes in, an acknowledgement that it is in that mode already is sent out.

Now each computer broadcasts what objects it has. Rather than broadcast every single object, checksums of the objects whose name fits within a range are sent out. When discrepancies are noticed, that range is further subdivided until the differences are worked out.

Specifically, the full name range is divided until 16 groups based on the upper 4 bits of the first character. The MD5 checksum of all the object names for each range is computed and broadcasted. For each range where a discrepancy is noticed, a request to investigate is sent out. At that point the subrange is further subdivided into 16 groups, and the process repeats. When the total number of objects in the range is less than 4 then all the objects are transmitted instead of the subgroups.

### Object Registration

Representatives receive object announcements from clients. If the message is about a new object or about deleting an object, it is broadcasted to the other representatives immediately to maintain the best cache conherence. If it is an update announcement, the request is queued for several seconds to reduce representative to representative communication.

### Load Rebalancing

The representatives predetermine a division of the IP address range to serve. It is possible that the division of labor will cause a major load imbalance. This can be detected from the existance message. If a computer notices that such an imbalance exists, he will propose a new distribution of addresses. The other representatives

will respond in agreement, and a switch will occur on a signal. When the signal is received, the representative will serve both address ranges for five seconds, to allow for propogation delays in the network. Since the protocol allows for multiple responses to a request, this will not significantly affect the load.

### 5.2.4 Analysis of scalability

The representative protocol is used to ensure scalability for large networks. As the size of the governed subnetwork becomes larger and larger, then the amount of server power necessary also goes up. However, given today's environment with the current array of TV series and movies, this should fit within a reasonable amount of RAM.

It is envisioned that there will be a representative at each corporations or at each ISP's link to the backbone. It would be possible for several small ISP's to collaborate and share one representative to reduce costs. It is possible that at an even higher level there might be a representative, such as at the country level, or at each coast.

# Chapter 6

# Conclusion

This document describes a method for setting up a heirarchy for querying object location. There is yet to be a wide distribution and implementation of multicast applications and technologies, so much of this is still speculation. Certainly as the technology becomes widespread and popular, the protocols will need to be revised.

Future work will need to consider security issues. In an environment where anyone can cache an object and serve it, then one must take steps to validate the authenticity of the object and to pay royalties to the originator.

The field promises to be an exciting one as conventional TV and radio broadcasts move permanently to the digital domain. Hopefully the ideas presented here will have some lasting impact in this area.

# Appendix A

# Portable Library API

## A.1 Multithreading

### A.1.1 pl_thread_create

```
int pl_thread_create(void (*func)(void *),void *param);
```

Creates a new thread of execution which calls `func` with the parameter `param`.

### A.1.2 pl_thread_exit

```
void pl_thread_exit();
```

Causes the current thread to terminate.

## A.2 Semaphores

```
class pl_shared_sem {
public:
        pl_shared_sem(int nsems,int initval);
        pl_shared_sem(int semid);
        void remove();
        int getid() const;
        bool islocked(int n);
```

```
        int getcount(int n);
        int lock(int n);
        int unlock(int n,int count=1);
};
```

### A.2.1 pl_shared_sem

Constructs a local reference to a global group of semaphores.

`pl_shared_sem(int nsems,int initval);` constructs a new group of `nsems` semaphores with an initial value of `initval`.

`pl_shared_sem(int semid);` constructs a local reference to an already existing group of semaphores. `semid` should be set to the return value of `getid()` from the group you wish to reference.

### A.2.2 remove

```
void remove();
```

Deletes the semaphores from the operating system.

### A.2.3 getid

```
int getid() const;
```

Returns the identifier of the semaphore group. This value can be used as the parameter to the constructor to make a duplicate reference of the group, even in another process.

### A.2.4 islocked

```
bool islocked(int n);
```

Returns whether a given semaphore is locked.

### A.2.5 getcount

```
int getcount(int n);
```

Returns the count of semaphore `n`.

### A.2.6 lock

```
int lock(int n);
```

Locks semaphore `n`. Blocks until it can receive the lock.

### A.2.7 unlock

```
int unlock(int n,int count=1);
```

Unlocks a semaphore. `count` is set to the number of times to unlock.

## A.3 Mutexes

```
class pl_unshared_mutex {
public:
        pl_unshared_mutex();
        ~pl_unshared_mutex();
        bool islocked();
        int lock();
        int unlock();
};
```

### A.3.1 pl_unshared_mutex

```
pl_unshared_mutex();
```

Constructs a local mutex (one that will not be shared among processes).

### A.3.2 islocked

```
bool islocked();
```

Returns whether the mutex is locked or not.

### A.3.3 lock

```
int lock();
```

Locks the mutex. This call will succeed if the mutex is unlocked or if it is locked by the current thread.

### A.3.4 unlock

```
int unlock();
```

Unlocks the mutex.

## A.4 Conditional Events

```
class pl_unshared_cond {
public:
        pl_unshared_cond();
        ~pl_unshared_cond();
        int wait(pl_unshared_mutex &pum);
        int timedwait(pl_unshared_mutex &pum,const timespec &ts);
        int signal();
};
```

### A.4.1 pl_unshared_cond

```
pl_unshared_cond();
```

Creates a local conditional object.

### A.4.2 wait

```
int wait(pl_unshared_mutex &pum);
```

Waits for the conditional to be signalled. Before waiting, the mutex is unlocked, and after being signaled, the mutex is relocked.

### A.4.3 timedwait

```
int timedwait(pl_unshared_mutex &pum,const timespec &ts);
```

Similar to `wait`, but includes a timeout of `ts.ts_sec` seconds and `ts.ts_nsec` nanoseconds.

### A.4.4 signal

```
int signal();
```

Signals one thread waiting on this conditional.

## A.5 Shared Memory

### A.5.1 pl_shmcreate

```
int pl_shmcreate(size_t size);
```

This creates a shared memory segment of a given size and returns it global identifier (can be used in any process). The segment is not actually usable until it is attached (see below).

### A.5.2 pl_shmremove

```
int pl_shmremove(int shmid);
```

This deletes a shared memory segment from the operating system, thus freeing up resourced.

### A.5.3 pl_shmattach

```
void *pl_shmattach(int shmid,pl_flags flags);
```

This attaches a given shared memory segment. Currently the only flag supported is `pl_RDONLY`, which attaches the segment as read only memory.

### A.5.4 pl_shmdetach

```
int pl_shmdetach(void *a);
```

This detaches a shared memory segment by supplying a pointer to the base of the memory segment.

# Appendix B

# Packet Descriptions

## B.1 Client Request

| command | adulen | objnamelen | unused |
|---|---|---|---|
| uniqueid | | | |
| ioc address | | | |
| ioc port | | unused | |
| originator address | | | |
| start adus | | | |
| stop adus | | | |
| object name | | | |

**command** Unique identifier for this type of packet, set to 0.

**adulen** The number of 32-bit integers used to represent each ADU. Can be set to 0 if information about the object is being requested.

**objnamelen** The length of the object name, not including the NULL character.

**uniqueid** A unique identifier for this request. This can be used by a representative (who may have several outstanding requests at once) to differentiate the replies.

**ioc address** The IOC address for the object that the replying computer should join.

**ioc port** The IOC port for the object that the replying computer should join.

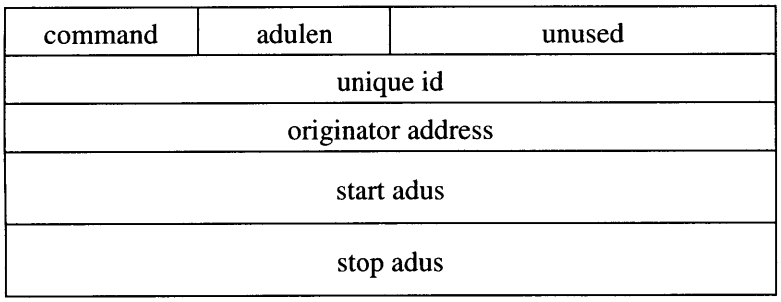
**originator address** The originator of the request (only different from the sending computer if a representative is sending the request). Used to determine whether a computer will reply.

**start adus** The start of the data range the client is requesting. Stored as an array of 32-bit integers.

**stop adus** The end of the data range the client is requesting. Stored as an array of 32-bit integers.

**object name** The name of the object, without the null terminator. Stored as an array of characters.

## B.2 Client Reply With Data




**command** Unique identifier for this type of packet, set to 1.

**adulen** The number of 32-bit integers used to represent each ADU.

**uniqueid** A unique identifier matching the request.

**originator address** The originating clients IP address.

**start adus** The beginning of the range of data that this computer can supply. In the event the computer cannot supply all the data, this range will be a subset of the requested range. Stored as an array of 32-bit integers.

**stop adus** The end of the range of data that this computer can supply. Stored as an array of 32-bit integers.

## B.3 Multiple Representative Packets

These are the packet types used for representatives to communicate with one another.

### B.3.1 Existance

| command | unused |
|---|---|
| numrequest-1 | |
| numrequest-2 | |
| numrequest-3 | |
| numrequest-4 | |
| numknown | |

**command** Unique identifier for this packet, set to 2.

**numrequest 1/4** The number of requests this computer has seen from the first quarter of its assigned address range.

**numrequest 2/4** Same for the second quarter.

**numrequest 3/4** Same for the third quarter.

**numrequest 4/4** Same for the fourth quarter.

**numknown** The total number of objects known by this computer.

### B.3.2 Reqest for Consistancy

| command | unused |
|---|---|

**command** Unique identifier for this packet, set to 3.

### B.3.3 Reply to Consistancy

| command | prev state | unused |
|---|---|---|

**command** Unique identifier for this packet, set to 4.

**consistancy state** A 1 if the computer was already in the consistancy check mode or a 0 if not.

### B.3.4 Object Count

| command | unused | range length |
|---|---|---|
| md5[0] | | |
| md5[1] | | |
| md5[2] | | |
| md5[3] | | |
| md5[4] | | |
| md5[5] | | |
| md5[6] | | |
| md5[7] | | |
| md5[8] | | |
| md5[9] | | |
| md5[10] | | |
| md5[11] | | |
| md5[12] | | |
| md5[13] | | |
| md5[14] | | |
| md5[15] | | |
| search prefix, packed in bytes | | |

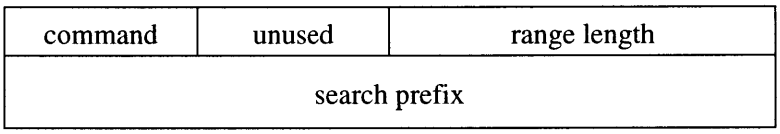
**command** Unique identifier for this packet, set to 5.

**range length** The length of the search prefix in nibbles (4 bits).

**search prefix** The search prefix is the first *range length* nibbles of the object name which this search represents.

**md5[16]** The MD5 checksum of all the objects represented by the search prefix plus the index in this array.

### B.3.5 Request to investigate

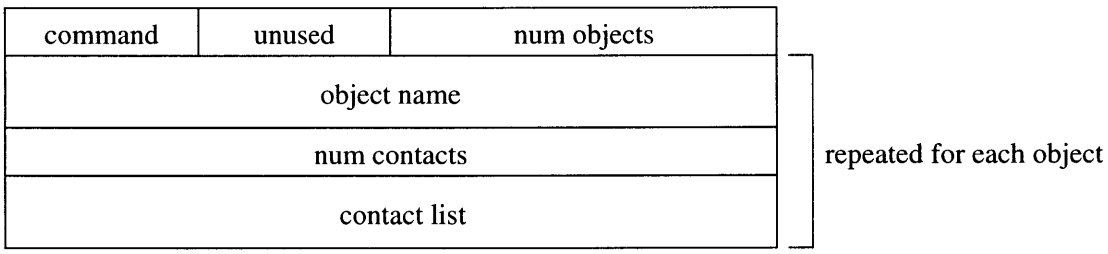



**command** Unique identifier for this packet, set to 6.

**range length** The length of the search prefix in nibbles (4 bits).

**search prefix** The search prefix, as described by the Object Count packet.

### B.3.6 Object List




**command** Unique identifier for this packet, set to 7.

**number of objects** Number of objects listed in this packet.

**object name[]** The name of each object, as an array of null terminated strings.

**contact list[]** The contact list for each object. Listed as the number of contacts followed by the each contact's IP address and ADU range.

### B.3.7 New Object

| command | namelen | unused |
|---|---|---|
| timeout, as # seconds since Jan 1 1970 | | |
| object name | | |
| contact info | | |
| ADU range | | |

**command** Unique identifier for this packet, set to 8.

**name length** Length of the object name.

**object name** Object name.

**contact info** The new contact for this object.

**ADU range** The ADU range this client has cached.

**timeout** When this object will timeout from the contact's cache.

### B.3.8 Delete Object

| command | namelen | unused |
|---|---|---|
| IP address of contact | | |
| object name | | |

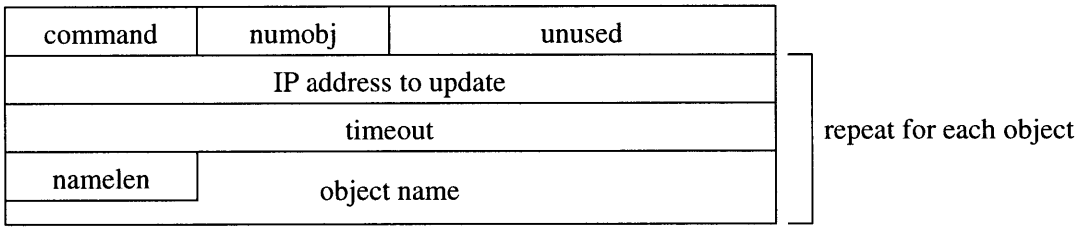
**command** Unique identifier for this packet, set to 9.

**name length** The length of the object name.

**object name** The object name.

**contact info** The IP address to delete from the contact list on this object.

### B.3.9 Update Object




**command** Unique identifier for this packet, set to 10.

**numobj** The number of objects in this update list.

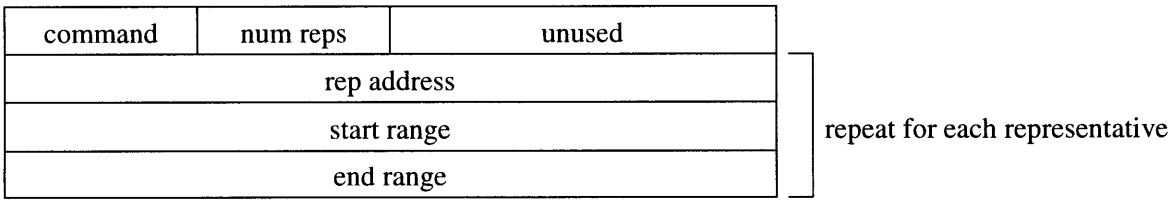
**name length**[] For each object, the length of the name.

**object name**[] The object name.

**contact info**[] The contact for this object to update.

**new timeout**[] The new timeout for this contact on this object.

### B.3.10 Load Balance Proposal




**command** Unique identifier for this packet, set to 11.

**num reps** The number of representatives known and listed.

**rep address**[] The address of a rep.

**rep start range**[] The first address the representative should serve.

**rep end range**[] The last address the representative should serve.

### B.3.11 Load Balance Acceptance

| command | unused |
|---|---|
| proposer's IP address | |

**command** Unique identifier for this packet, set to 12.

**proposer's IP** Since multiple proposals may be out, this identifies which proposal is being accepted.

### B.3.12 Load Balance Execute

| command | unused |
|---|---|

**command** Unique identifier for this packet, set to 13.

# Appendix C

# Implementation

The peer protocol is implemented inside HPRTP. The files `deadbit_manager.?xx` contain the code that parse the heirarchy configuration file and create the necessary listening threads. There is a call from `hprtp_server.cxx` to `deadbit_manager::find_object` in the event that a call to open the object fails. Thus when the object cannot be located, an attempt will be made to find computers with copies of the object, and get those computers to join the IOC channel.

# Bibliography

[1] Mp3 benchmark web site. http://www.mp3bench.com.

[2] Real audio web site. http://www.realaudio.com.

[3] Stefan Agamanolis, Alex Westner, and V. Michael Bove Jr. Reflection of presence: Toward more natural and responsive telecollaboration. *Proc. SPIE Multimedia Networks, 3228A*, 1997.

[4] Roger George Kermode. *Smart Network Caches: Localized Content and Application Negotiated Recovery Mechanisms for Multicast Media Distribution.* PhD dissertation, Massachusetts Institute of Technology, June 1998.

[5] David L. Mills. Network time protocol (version 3): Specification, implementation and analysis. RFC 1305, March 1992.