A Mechanism for the Administration of Real-Time Channels[1]

Bruce A. Mah
bmah@tenet.Berkeley.EDU
The Tenet Group
University of California at Berkeley
and
International Computer Science Institute

Abstract

Future applications will require network services that offer performance guarantees. We refer to such services as "real-time", and to network connections offering such guarantees as "real-time channels". In this report, we discuss the service description and interfaces for a mechanism for the administration of real-time channels. We also describe the Real-Time Channel Administration Protocol (RCAP), which implements this mechanism in the prototype version of the Tenet Real-Time Protocol Suite.

1.0 Introduction

The Tenet Scheme for real-time communications introduces the concept of "real-time channels". A real-time channel is a simplex, unicast, network connection with associated performance guarantees [FerVer89]. Examples of such guarantees are an upper bound on the maximum end-to-end delay of a packet sent on a particular real-time channel or a lower bound on the throughput provided to a channel.

We assert that many network applications in the future will require such performance guarantees in order to provide acceptable service to their users. This need will be particularly acute in the case of continuous media applications, such as those involving digital video or digital audio services. More details on the requirements of specific applications can be found in [Ferrari90].

Critical to the operation of real-time channels is a mechanism for their administration or management. In particular, a means must be developed to create and destroy channels

1. This research was supported by the National Science Foundation under an NSF Graduate Fellowship, the National Science Foundation and the Defense Advanced Research Projects Agency (DARPA) under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives, AT&T Bell Laboratories, Hitachi Ltd., Hitachi America Ltd., Pacific Bell, the University of California under a MICRO grant, and the International Computer Science Institute. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing official policies, either expressed or implied, of the U.S. Government or any of the sponsoring organizations.

while maintaining the performance required by, and guaranteed to, the clients of the real-time communication service. The mechanism will need to manage the various resources used by the channel, such as link bandwidth, packet scheduling time in gateways, channel identifiers, and so on.

In this report, we present the requirements for a real-time channel administration mechanism and describe the necessary software interfaces to other network protocol entities. We then present the decomposition of the functionality used in the design of the Real-Time Administration Protocol (RCAP), which implements this mechanism, and describe the components, implementation, and operation of the prototype version of the RCAP software.

2.0 Service Description

In this section, we describe the basic functionality for a mechanism for administering real-time channels. The three main services to be performed are channel creation (referred to as “establishment”), channel destruction (“teardown”), and channel status reporting.

As mentioned previously, real-time channels are assumed to be simplex, unicast connections.[2] Therefore, a channel has a fixed source (also known as the “sender”) and a fixed destination (also referred to as the “receiver”). The direction along the channel’s path in which data travels is referred to as “downstream”, while we call the direction opposite the data flow “upstream”.

2.1 Channel Establishment

Channel establishment is the process of creating a real-time channel according to the performance requirements requested by client applications. This process involves several different functions.

One function is that of routing, the process of determining the path of a real-time channel from the source to the destination. Routing mechanisms may range from a simple, static, table-driven routing algorithm to more sophisticated ones that choose routes depending on the resource utilization and availability on each of the links and nodes in a network. Routes can be decided at the channel sources or on a hop-by-hop basis. We assume that routes are fixed at connection establishment time.[3]

The Tenet Scheme calls for admission control at channel establishment time in order to guarantee the required performance of existing channels (as well as that of the channel currently being established). The network can accept channels whose presence will not cause the performance guarantees of existing channels to be violated; others are rejected.

---

2. Ongoing research in the Tenet Group is aimed at extending the real-time channel paradigm to duplex and/or multicast connections.

3. Recent research addresses the issues involved in rerouting real-time channels for load-balancing or fault recovery [PaZhFe92].

Any channel administration mechanism must perform admission control tests at channel establishment time to protect the network from potential overload.

The Tenet Scheme proposes specific tests for the case of a network whose nodes use the Earliest Due Date (EDD) scheduling discipline for their bottleneck resources [FerVer89]. These resources are usually assumed to be associated with the output links on each node, as in the case of a switch with output queueing, but they could also be node-wide resources, such as a CPU in a switch or gateway. Although different tests are required for different scheduling disciplines, [Ferrari92] postulates that these admission control tests can be developed for any member of a wide class of service disciplines.

Resource allocation is the next logical action of the channel establishment process. In some networks, this step may merely consist of adjusting bookkeeping entries in the nodes in order to provide accurate data for the admission control tests. In other cases, resource allocation may be more complex (for example, when it is necessary to create data-link layer virtual circuits).

Resource allocation in the Tenet Scheme consists of an initial phase in which the maximum possible resources are allocated to the channel, followed by a relaxation phase in which excess resources are deallocated and released for other uses. For example, a new channel is initially allocated the lowest delay bound possible in each node. The resulting initial end-to-end delay bound may be much lower than originally requested by the client. During relaxation, the local delay bounds on some or all of the nodes may be adjusted upwards to release the excess “delay bound resource”.

The entities performing data delivery (for example, network layer protocol software) need to be informed of the creation of the new channel. The necessity for this action is particularly apparent in the case that data delivery is performed by entities distinct from those providing the service of channel establishment. This situation exists, for example, in the Tenet Real-Time Protocol Suite.

Finally, the clients of the data delivery service (those requesting communication) need to be notified of the existence of the new channel. They also need to be provided with identifiers (generally referred to as “channel identifiers”) to use when referring to the channel.

2.2 Channel Teardown

Channel teardown is the process of destroying a real-time channel and releasing its resources for re-use. Although the teardown procedure is simpler than that of establishment, it also involves several distinct actions.

The most critical action of channel teardown is the deallocation of the channel’s resources. This action is necessary so that the various network resources, such as link bandwidth, can be reallocated to other channels in response to new channel establishment requests. If some resources cannot somehow be deallocated, they cannot be used by new channels, and new channel establishment requests may be unnecessarily rejected.

Another required action is the notification of the data delivery entities. They must be told that the channel in question no longer exists. In some cases this action is coupled with the deallocation of some resource, such as a channel identifier number or virtual circuit identifier.

Finally, the client applications using the channel need to be notified that the channel has been torn down. Such an action can be made explicit, via some sort of notification mechanism in the operating system, or it can be implicit, perhaps in the form of an error message if an application attempts to send or receive data on the non-existent channel. Presumably the latter method of notification can always be used as a fallback.

2.3 Channel Status Reporting

A third function to be performed by a channel administration mechanism is that of reporting the status of a channel. While not necessary for the correct operation of the network, such a function can be invaluable when attempting to design, implement, and debug software that either uses or implements various pieces of a real-time network service. In particular, it can be helpful to know the resource utilization and corresponding performance guarantees at each node along a channel's path.

3.0 Software Interfaces

In this section we describe the software interfaces to various other components of a real-time network service. There are three interfaces that need to be considered, the interface between the channel administration mechanism and its client applications, the interface to the data delivery entities, and the interface between channel administration entities located on different network nodes. The relationship between the different network components, and their interfaces, is shown in Figure 1.

We note that information regarding the state of the resources available at each node is, in general, scattered throughout the network. It is therefore appropriate to carry out channel management as a distributed computation involving all of the nodes along a channel's path. The decomposition of functionality discussed here assumes that this is indeed the case, that channel management is carried out in a distributed manner.

3.1 Interface to Client Applications

The interface to client applications provides the means by which application programs can request channel management services. It is the only application-visible portion of the channel administration mechanism.

We believe that using a procedure call-based interface provides the most natural service in typical UNIX-like programming environments. Other interface paradigms, such as message-passing between the application and the service, seem inappropriate as they tend to complicate the use of the channel administration service without providing any real benefits in functionality. We note that the closest analog to channel administration in BSD-style

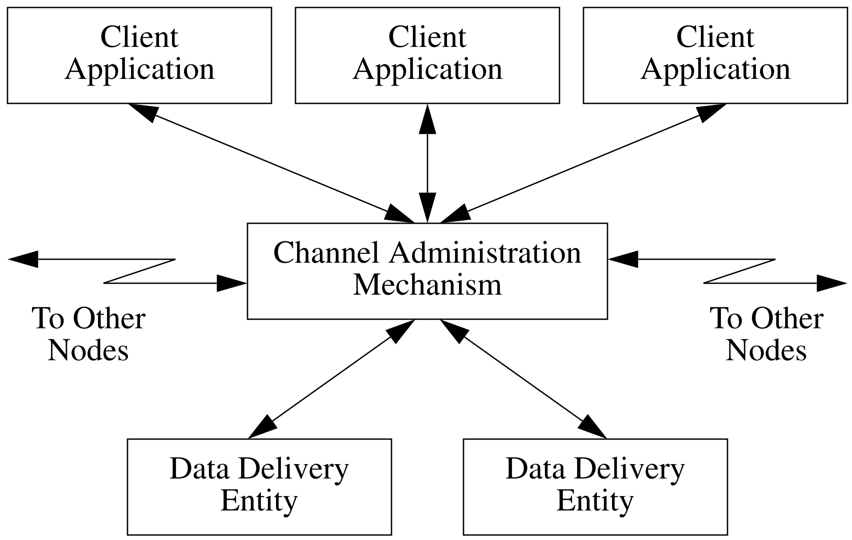

FIGURE 1. Software Interfaces to a Real-Time Channel Administration Mechanism

UNIX systems is the set of socket management system calls, for example `socket()`, `bind()`, and `connect()`. These primitives, while technically not procedure calls, use the same style of software interface [Leffler89].

3.2 Interface to Data Delivery Mechanisms

Part of the process of creating or destroying real-time channels is notifying the data delivery mechanisms on each node. In the case of channel establishment, notification requires the new channel parameters, incoming and outgoing channel identifier and port correspondences, and so on. Channel teardown merely needs the identification of the channel to be destroyed.

The software interface that enables this notification is particularly important in situations where channel administration and data delivery are performed by different protocol entities or modules of an operating system. The particular type of mechanism to be used here is not critical, as the mechanism generally will be constructed only once per implementation. As it is not visible to the client, the choice of paradigm for this interface should not impact applications or users in any way.

3.3 Interface to Other Administration Entities

We assume that channel management is to be carried out in a distributed manner, as the required information on resource availability in the network is also distributed. An interface between the channel management entities on each of the nodes is therefore necessary so that they can communicate during channel management operations.

A procedure call-based mechanism (in this case, some form of RPC) would have the benefit of simplicity; however this paradigm has the disadvantage of requiring synchronous, lock-step actions between the participants. This kind of communication may not be appropriate for a channel administration entity, which may have multiple operations pending simultaneously. Because of this need for asynchrony, a message-passing paradigm is the best suited for this situation.

We note that in an object-oriented distributed system such as Clouds [Dasgupta90], some sort of object migration, where channels are represented by objects, may be appropriate. Not having such an environment available to us, we did not investigate this possibility further.

4.0 Decomposition

We now describe the decomposition of functionality used in the design of the first prototype implementation of the Real-Time Channel Administration Protocol (RCAP). The design was produced by Anindo Banerjea and the author, with the assistance of other Tenet Group members during Spring 1991, and is fully documented in [BanMah91a]. This software performs the channel administration functions in the Tenet Real-Time Protocol Suite.

Our model of the environment is a packet-switching internetwork, possibly heterogeneous. The various subnets making up the internetwork consist of nodes joined by point-to-point links.[4] Dual-homed nodes (those that have interfaces on more than one network) are assumed to be able to route packets between networks. Each node is assumed either to be a general purpose computer (such as a workstation) or a special-purpose network node capable of sending and receiving packets and handling channel administration functions, perhaps with the assistance of another processor. An example of the latter is the XUNET 2 ATM switch, which relies on its attached switch controller to perform virtual circuit and channel administration functions [Fraser92].

4.1 Client Interface

The client interface portion of RCAP is the portion visible to the client entities (the application programs). It exports various primitive operations so that applications can request various channel management services. The operations exported by the current RCAP design are listed in Table 1.

4.2 Resource Manager

A resource manager is responsible for allocating and accounting for the various resources at each node in the network. In accordance with our assumption of distributed channel

4. Appropriate modifications can be made to handle the case of broadcast-style networks.

Primitive	Description
establish_request	The sending client invokes this primitive to request a new real-time channel. It provides its performance requirements, traffic characteristics, and addressing information. RCAP returns a unique channel identifier if the request succeeds.
register	A receiving client uses the *register* primitive to indicate to RCAP that it is ready to receive connections on a given port.
receive_request	This primitive, when invoked by the receiving client, causes the receiving client to wait until an establishment request arrives from a sending client. It then returns the establishment from that request, allowing the receiver to *accept* or *deny* the request.
accept	The receiving client invokes the *accept* primitive to indicate acceptance of an establishment request.
deny	The receiving client uses the *deny* primitive to reject an establishment request.
unregister	This primitive is used by a receiving client to indicate that it will no longer accept any requests on a port.
status	The sender can request information about the state of a channel by invoking the *status* primitive.
close	Either the sending or receiving client can invoke the *close* primitive to close a channel and release its resources.

TABLE 1. RCAP Service Primitives Available to Client Programs

administration, resource management is also distributed, with each node's resources being managed exclusively by a resource manager running on that node.

The state information contained within the resource manager (such as the allocation and availability of resources) should be protected from tampering by the client applications on each node. This can easily be solved in a UNIX-like environment by running the resource manager in a separate process from any of the client applications. This approach is, in fact, taken in the prototype RCAP implementation. Similar steps could be taken in other operating system environments.

5.0 The Tenet Real-Time Protocol Suite

Before presenting the details of the Real-Time Channel Administration Protocol (RCAP), it is first necessary to describe the Tenet Real-Time Protocol Suite, which contains and uses RCAP.

The Tenet Real-Time Protocol Suite is a set of communication protocols designed to provide real-time communication (with performance guarantees) in a packet-switching internetwork [Lowery91]. The components of the Suite (and their relationships) are pictured in Figure 2. Of particular interest is the separation of the functions of data delivery and control. The portion of the protocol stack on the left of the figure reflects the traditional layering of network protocols, while the arrows going to and from RCAP on the right of the diagram show the flow of channel administration control.

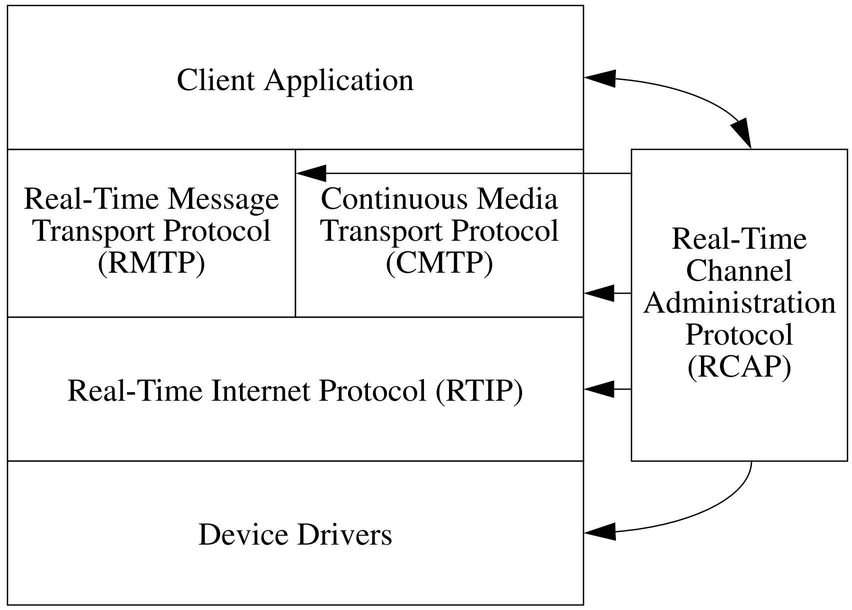

FIGURE 2. The Tenet Real-Time Protocol Suite

The Real-Time Channel Administration Protocol (RCAP) handles connection-level channel management for the Protocol Suite in response to requests from the application programs. It performs the three channel management functions of establishment, teardown, and status reporting described earlier. RCAP communicates with the network-layer and data link-layer protocol entities at each node along the channel's path, as well as with the transport-layer protocol entities at the channel's endpoints. These control paths are shown by the arrows in the protocol stack.

The Real-Time Internet Protocol (RTIP) provides for connection-oriented, performance-guaranteed, unreliable delivery of packets [VerZha91]. It occupies a place analogous to IP in the Internet Protocol suite. In fact, an earlier design of RTIP, described in [Lowery91], was based on IP and supported real-time communication using the IP options fields (thus the origins of this protocol's name).

The services of RTIP are used by two different transport-layer protocols. The Real-Time Message Transport Protocol (RMTP) provides the service of connection-oriented, performance-guaranteed, unreliable delivery of messages [VerZha91]. This transport layer is quite lightweight. Two features frequently associated with transport layers, connection management and reliable delivery through retransmission, are absent from this protocol (the former is provided by RCAP and the latter is incompatible with the notion of guarantees on message delay). Thus, the main functions of this transport layer are flow control (accomplished by rate control) and the fragmentation and reassembly of messages.

The Continuous Media Transport Protocol (CMTP) is designed to support the transport of periodic network traffic with performance guarantees. By using the knowledge that the data to be transmitted is periodic in nature, CMTP can gain more effective use of the network. The prototype implementation also uses the knowledge of periodicity to implement implicit send and receive operations. Data is passed to and from the client application using shared memory; explicit send and receive system calls are unnecessary because both the client and the operating system know when data needs to be sent or received [WolMor91].

The possibility of network failures requires a reactive network control mechanism, in addition to the proactive mechanism provided by RCAP. The Real-Time Control Message Protocol (RTCMP) is intended to address this need. One of its main functions is to facilitate the rerouting of channels whose paths have been interrupted due to a failure. The exact functionality and interfaces for RTCMP have not been well-defined yet, so it has not been included in Figure 2. RTCMP is planned for inclusion in the next version of the Tenet Real-Time Protocol Suite.

6.0 RCAP Implementation

We now describe the first version of RCAP, designed and implemented by Anindo Banerjea and the author during 1991 and 1992. The software is divided into two distinct sections of code, a library that is to be linked into each client application and a daemon process that runs on each node, independent of the applications. The relationship between the library, daemon, and associated application programs is shown in Figure 3.

6.1 The RCAP Library

The RCAP library implements the client interface portion of the channel administration mechanism. It exports a set of procedures that can be called from within a user program to request various channel management functions. The procedures correspond roughly to the function primitives enumerated earlier; these procedures are shown in Table 2.

The RCAP library calls have rough analogs in the BSD 4.2 UNIX networking system calls. `RcapEstablishRequest()` is similar to the BSD `connect()` system call, although the former, by necessity, requires much more information on the connection to be established. `RcapRegister()` is almost identical in function to BSD's `listen()` call. The combination of `RcapReceiveRequest()` and `RcapEstablishReturn()` is roughly analogous to the BSD `accept()` system call (however, applications using the BSD system calls normally do not have the opportunity to reject a connection request once they have received it). There are no counterparts for the `RcapUnregister()` and `RcapStatusRequest()` library calls. The `RcapCloseRequest()` call is much like the BSD `close()` system call, although the latter is capable of operating on many more objects than network connections (files and pipes, for example).

In current implementations, the RCAP library is a standard library that is linked into client applications at load-time. A header file provides function prototypes and data structure definitions needed by the clients.

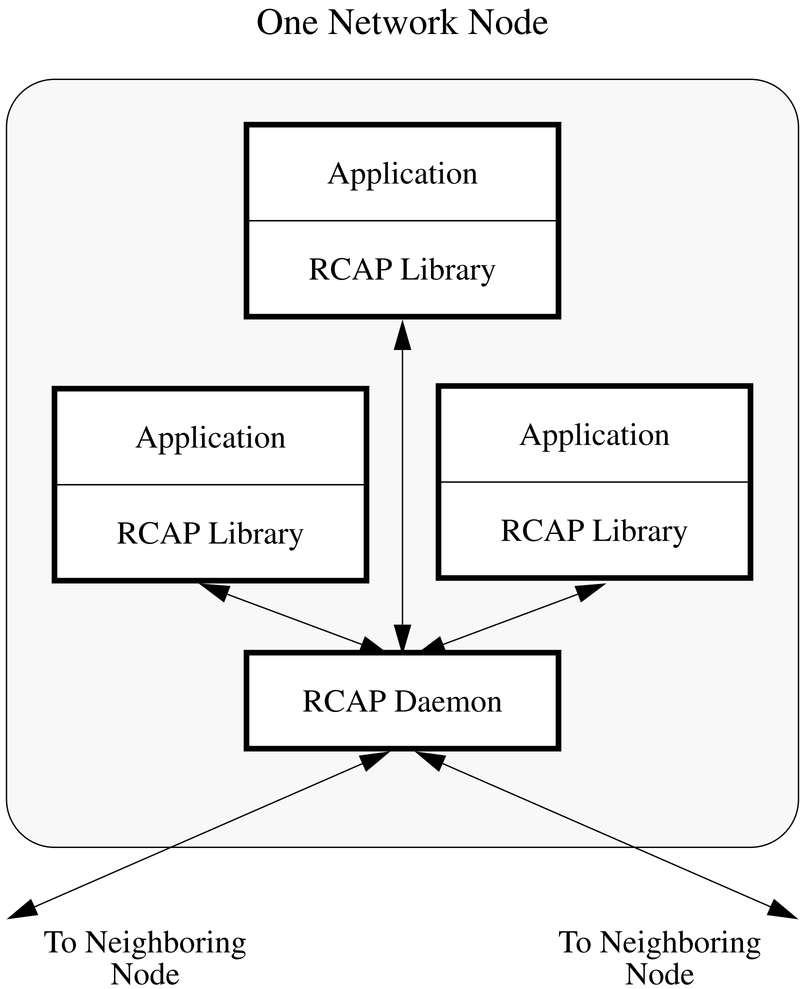

FIGURE 3. The RCAP Library and Daemon

The RCAP library communicates with the RCAP daemon process running on the local node via BSD-style TCP sockets. The library and daemon communicate using messages transmitted over the sockets; the messages themselves will be described in Section 6.2. We opted to use TCP sockets as the method of message transport primarily because they provide reliable delivery of the RCAP control messages. Another consideration was that the socket mechanism was supported and implemented by all the operating systems we could foresee using in the near future. The RCAP software is, however, easily modifiable to use a different form of message transport.

The RCAP library contains very little state information, and none at all related to the state of the real-time channels. As this library will be running in the same address space as the application programs, each application program can, at least potentially, change information associated with the library. Therefore, we chose not to trust the library with any information that could affect the operation of the network. It is convenient to think of the RCAP library as simply a specialized set of Remote Procedure Call (RPC) client stubs.

Library Function	Description
`RcapEstablishRequest()`	Request to establish a new channel. Inputs are the traffic characteristics, performance requirements, addressing information, and optional control data for higher-level control functions. Output is a channel identifier if establishment was successful or an error code if establishment failed.
`RcapRegister()`	The receiver makes this procedure call to indicate its willingness to accept new connections on a given port. Sending client applications will use an ordered pair consisting of a network address and a port number to specify the destination of a real-time channel.
`RcapReceiveRequest()`	The receiver uses this procedure to actually obtain an incoming establishment request. Any such request has passed all admission control tests within the network, but the receiving client has the ultimate decision as to whether or not the channel establishment will succeed or fail.
`RcapEstablishReturn()`	This function combines the *accept* and *deny* primitives. The receiver uses this function to indicate final acceptance or denial of the channel (the receiver has the power to "veto" a channel establishment made to it, even if the network would otherwise permit the channel).
`RcapUnregister()`	The receiving client uses this function to indicate that it is no longer willing to accept any new connections on a given port. Existing real-time channels are unaffected.
`RcapStatusRequest()`	This function provides a means for the sending client to obtain the status of an already established channel. It returns a buffer with network-dependent data structures giving the resource usage and parameters of the channel at each of the nodes along its path.
`RcapCloseRequest()`	Either the sending or receiving clients can use this function to initiate a channel teardown.

TABLE 2. RCAP Exported Functions

6.2 The RCAP Daemon

There is one RCAP daemon process running on each node. It manages the resources (such as link bandwidth) associated with the local node, in response to requests for channel establishment and channel teardown.

Each RCAP daemon communicates using control messages sent over the TCP control connections. Each message has an implicit direction of travel associated with it, either "upstream" (towards the sending application) or "downstream" (towards the receiving application). Table 3 shows the messages used between the RCAP daemons and between the daemons and the instances of the RCAP library.

The RCAP library and daemon use an additional set of messages to communicate within a single node. These messages implement a simple cross-domain procedure call facility for RCAP operations contained entirely inside a single network node; each of these messages

Message Name	Direction	Description
establish_request	Downstream	A control message sent from a source client requesting establishment of a new channel. Causes admission control tests to be run at each node along the channel's path and resources to be tentatively allocated for the channel.
establish_accept	Upstream	Indicates acceptance of a new channel. Causes relaxation of resources at each node along the channel's path and communicates establishment of the channel to the data delivery entities.
establish_denied	Upstream	Indicates that a channel establishment request was rejected, either inside the network or by the destination application. Causes the deallocation of all resources previously allocated to the channel.
status_request	Downstream	A message sent by the source application requesting the status of the channel at each node. Information on the local channel parameters is appended to the message as it travels down the channel's path towards the destination application.
status_report	Upstream	Contains local channel parameters gathered by a status_request message. This message is merely used to convey these parameters back to the sending application.
close_request_forward	Downstream	A request from the sending application to close an existing channel.
close_request_reverse	Upstream	A request from the receiving application to close an existing channel.

TABLE 3. RCAP Control Messages

also has an implicit direction of travel to an RCAP library or to the local RCAP daemon. They are described briefly in Table 4.

6.3 Implementation Notes

The prototype version of RCAP was implemented by Anindo Banerjea and the author during 1991 and 1992. It is written in C, and initially ran on DECstation 5000 series computers using the Ultrix operating system (a BSD UNIX derivative). The prototype RCAP daemon consists of eleven thousand lines of C source code, while the RCAP library is composed of two thousand lines of C.

As can be inferred, the majority of the work involved the RCAP daemon. In an attempt to balance the workload of the implementors, we decided to split the RCAP daemon in two parts for implementation purposes. Roughly, one part consisted of the real-time admission control tests and resource manager, while the other contained the interprocess communication code necessary to communicate with other nodes and with various instances of the

Message Name	Direction	Description
`local_register`	To Daemon	"Registers" an application as a destination willing to receive channel establishment requests on a given port.
`local_unregister`	To Daemon	"Unregisters" a receiving application, as it is no longer willing to handle incoming channel establishment requests.
`local_receive_request`	To Daemon	Used by a receiving application to get a channel establishment request from the local RCAP daemon.
`local_return_parms`	To Library	Returns the channel parameters of a pending channel establishment. Sent from the daemon to the library at a channel's destination so that the receiving application can decide whether or not it wants to accept the channel.
`local_establish_return`	To Daemon	Message indicating a final accept/reject decision by a channel destination.
`local_return`	To Library	A message used to communicate return values from various operations back to an instance of the RCAP library and its attached application.

TABLE 4. "Local" RCAP Control Messages

library. One of the implementors (Banerjea) implemented the former side of the daemon, while the other (the author) wrote the IPC code and the RCAP library. The two "sides" of the RCAP daemon communicate across a send/receive interface, by which the resource manager sends or receives RCAP control messages.

The above decomposition was intended to make the RCAP software easy to implement. The resource manager and admission control tests were developed and largely tested in isolation, while the IPC module and library were written and tested without the need for the other components of the RCAP daemon.

However, we discovered that the splitting of the RCAP daemon is somewhat less than ideal. In particular, the simple send/receive interface does not allow for any sharing of information about the various real-time channels. The common state is necessary to support various IPC functions that rely on the state of channels (to ensure, for example, that control messages are sent correctly to the next upstream or downstream node along the path of a channel). The result is that there is quite a bit of duplication of information (and effort) as the two "sides" of the RCAP daemon both attempt to maintain an accurate picture of the channels being managed. As an extreme example it is possible for an incoming RCAP control message to be converted from the network representation to the host-dependent representation as many as three times as it is being processed by the daemon.

The RCAP software does have the advantage of being quite portable, as it has few dependencies on the remainder of the protocol suite or on the host operating system. The initial RCAP implementation was ported to run on SunOS by Sun Microsystems, HP-UX by Hewlett-Packard, and IRIX by Silicon Graphics, on their respective hardware platforms, with few or no changes necessary. A port to Sprite, a UNIX-like operating system developed at the University of California at Berkeley for the DECstation and SparcStation platforms, is under way with promising results. Most changes were only needed to accommodate operating system and compiler idiosyncracies.

RCAP first became operational in the summer of 1992. Shortly afterwards we were able to use it to establish and manage RMTP/RTIP channels between DECstations on a small FDDI testbed network.

Currently, Anindo Banerjea is continuing the refinement and debugging of the real-time tests and resource management, as well as trying to tune the current RCAP implementation to the FDDI testbed (the real-time tests rely on a characterization of each machine and link in the network, which must be generated from measurements and analysis). The author is preparing to adapt RCAP to establish real-time channels between the hosts on a high-performance HIPPI network recently constructed at Berkeley.

7.0 Operations

In this section, we describe, in an informal manner, how the different RCAP components perform the task of channel management. As mentioned previously, we assume an environment of general-purpose computers connected by a packet-switching internetwork; the internetwork is composed of various subnets, possibly heterogeneous, connected by gateways or routers.

7.1 Channel Establishment (Receiver)

Before any communication can take place on a channel, an application must be present to act as a receiver on that channel. Applications indicate their willingness to do so by “registering” themselves with the RCAP service. The flow of control in registration (and corresponding unregistration) is shown in Figure 4.

The potential receiver executes an `RcapRegister()` library call, indicating the port number for which it will receive establishment requests and act as a destination. Note that an application may be a receiver for channels on more than one port, while simultaneously establishing and sending data on channels to other applications.

The RCAP library linked to the application constructs a `local_register` control message, which it then sends to the local RCAP daemon. The library (and hence the application process) then blocks, waiting for a response from the daemon.

The daemon adjusts its data structures to reflect the application’s status as a potential channel destination, and sends a `local_return` control message back to the library to acknowledge the request. The library call then returns.

Once registered, the application then has the opportunity to process channel establishment requests directed towards its port. The process is illustrated in Figure 5.

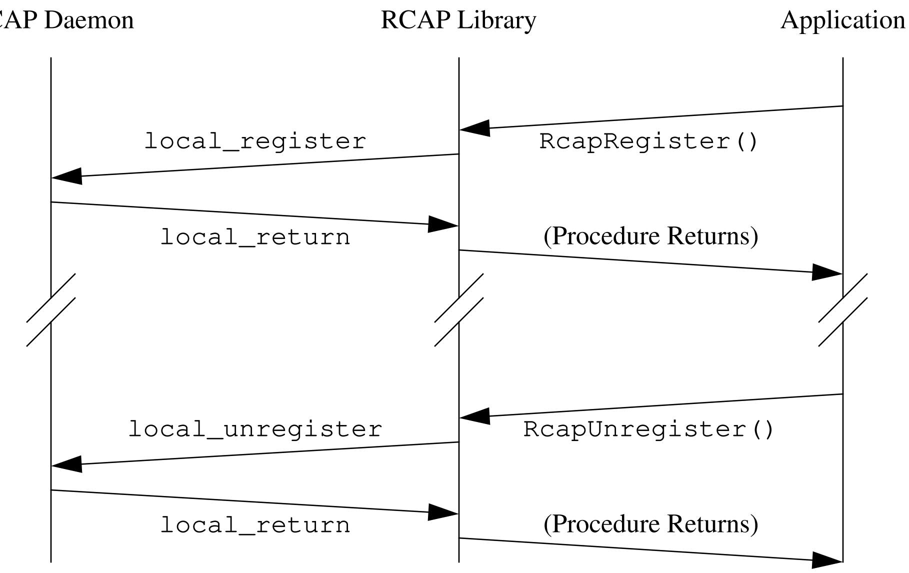

FIGURE 4. Registering and Unregistering a Receiving Application

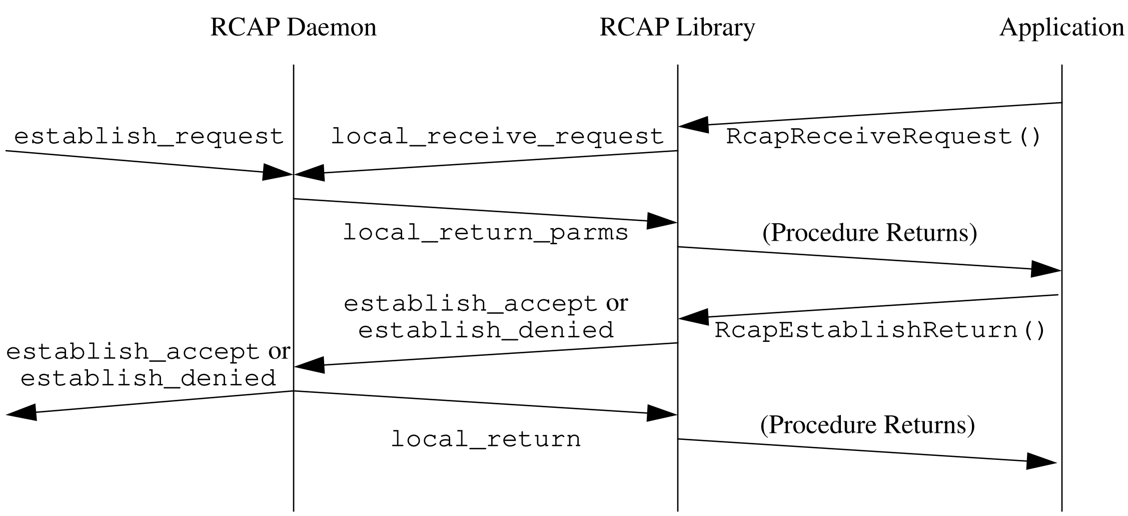

FIGURE 5. Channel Establishment (Receiving End)

To get a channel establishment request, the application executes an `RcapReceiveRequest()` library call. This message indicates the port number for which the request should be received; of course, the application must already be registered for that port.

The RCAP library code then builds a `local_receive_request` message, which it sends to the local daemon. The daemon looks for outstanding `establish_request` messages (indicating pending channel establishments) for the port in question. If such a message is found, it returns a `local_return_parms` message to the calling library containing the parameters for a pending channel establishment request. The `RcapRe-`

ceiveRequest() library call then returns to the application. If no such message is found, the RCAP library stores the local_receive_request for a time when a channel establishment request arrives. The application is blocked until it receives the local_return_parms message.

The receiving application has the ultimate say on whether or not the pending real-time channel is to be established; it indicates its decision via an RcapEstablishReturn() library call. The library transmits either an establish_accept or establish_denied control message to the local daemon. The daemon forwards the message upstream and returns a local_return message to the library.

Assuming the destination application accepted its end of the connection, it can now read data from its end of the real-time channel, invoking whatever interface is defined for the data delivery protocols being used. For example, when using the prototype implementation of RMTP and RTIP, a special type of socket is provided so that the receiving application can use the standard read() series of system calls.

The process of unregistering an application from a port is simply the opposite of registering, and the control messages exchanged are analogous. This process is illustrated in the lower part of Figure 4.

7.2 Channel Establishment (Sender)

According to the current Tenet Scheme, channel establishment is initiated by the sender, as illustrated in Figure 6.

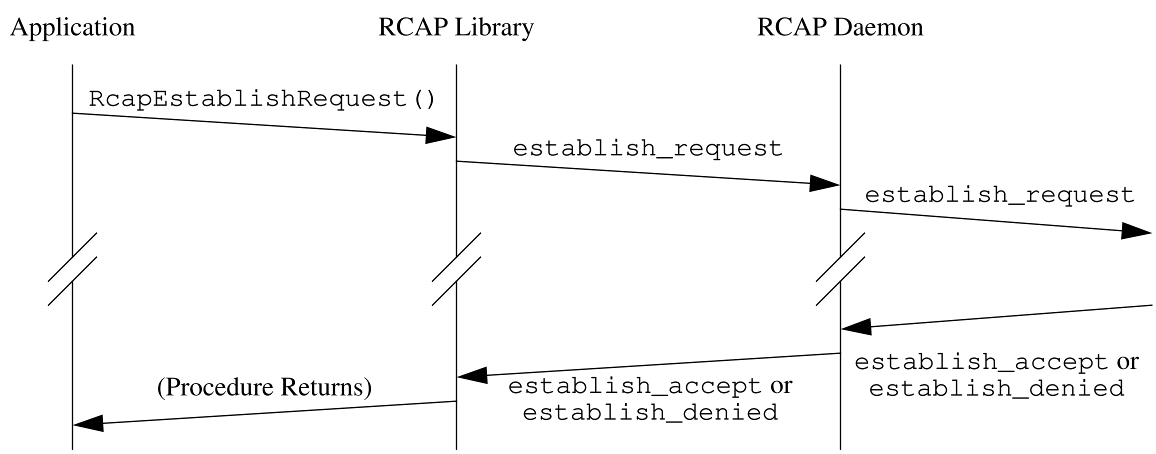

FIGURE 6. Channel Establishment (Sending End)

A potential sender wanting to establish a real-time channel must first obtain the network address and port number of the receiving application. The mechanisms for doing so are beyond the scope of this work; we envision that, ultimately, some sort of nameserver will provide this kind of location information. For the moment, we assume “well-known” ports and network addresses.

The sending application makes an `RcapEstablishRequest()` library call. This procedure call has as its arguments the network address and port number of the destination application, the traffic characteristics of the sender’s data, and the performance requirements necessary to the application. We assume that the latter two parameter vectors are either known to the application or can be obtained via some network service.

The attached RCAP library transmits an `establish_request` message to the local RCAP daemon. The message travels along the future channel’s path towards the receiver, transmitted between the RCAP daemons along their control connections. Currently, routing is done by a static routing table, loaded at daemon start-up time. The table is similar in format to those found in BSD UNIX kernels.

At each node, the necessary admission control tests are run. If the tests succeed, resources are tentatively reserved for the real-time channel and a record containing the local channel parameters is appended to the `establish_request` message.[5] The new `establish_request` message is then forwarded to the next node. The flow of control messages through each RCAP daemon during channel establishment is shown in Figure 7.

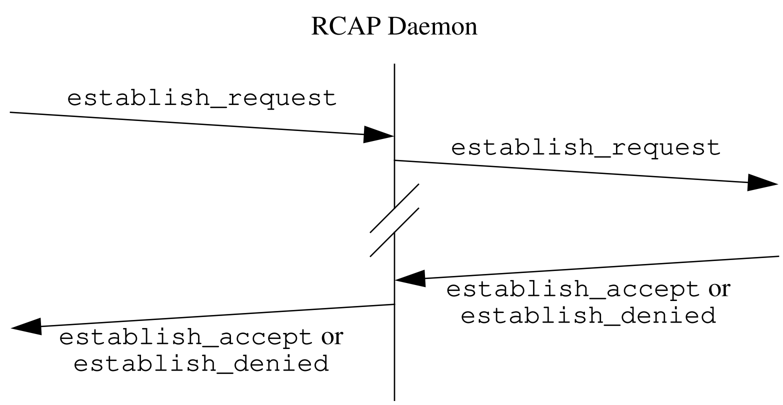

FIGURE 7. Channel Establishment (In RCAP Daemon)

If at any point along the path, an RCAP daemon decides that the channel cannot be supported by the network, an `establish_denied` message is returned to the sender. It causes the resources allocated to the channel in progress to be released. When the message reaches the source node, its daemon passes the `establish_denied` message to the sending application’s library, which then returns an error code to the application. The application may then retry the establishment if it so desires.

5. In the case of a heterogeneous internetwork, a hierarchical abstraction is used to summarize the network-dependent parameters for use by the end-to-end channel establishment processing. This abstraction, however, does not substantially change the description of the channel administration mechanism. Details of this process may be found in [BanMah91b].

When the receiving application receives the establish_request message, the daemon on that node has the opportunity to make the final establishment decision, as previously described in Section 7.1. If the destination is willing to receive data along this channel, it begins sending an establish_accept message, hop-by-hop via each RCAP daemon, backwards along the channel's path towards the source. At each hop along the reverse path, resources for the new channel may be relaxed, allocation confirmed, and the data delivery entities notified of the new channel setup.

When the establish_accept message reaches the channel's source node, the local RCAP daemon then passes the message back to the sending application, unblocking the process. The application then obtains the channel identifier for the channel from the returned values from the library call. The source application can then begin sending data along the new channel.

We note that although RCAP was designed for Tenet-style real-time channels, which currently must be established beginning with the sender, it would be simple to adapt the channel administration mechanism (and the RCAP implementation) to perform receiver-initiated channel establishment or to operate on duplex channels.

7.3 Channel Status Request

RCAP provides a means for the sending side of an existing channel to determine various channel parameters at each of the nodes along its path. The control messages used during a status request are shown in Figure 8.

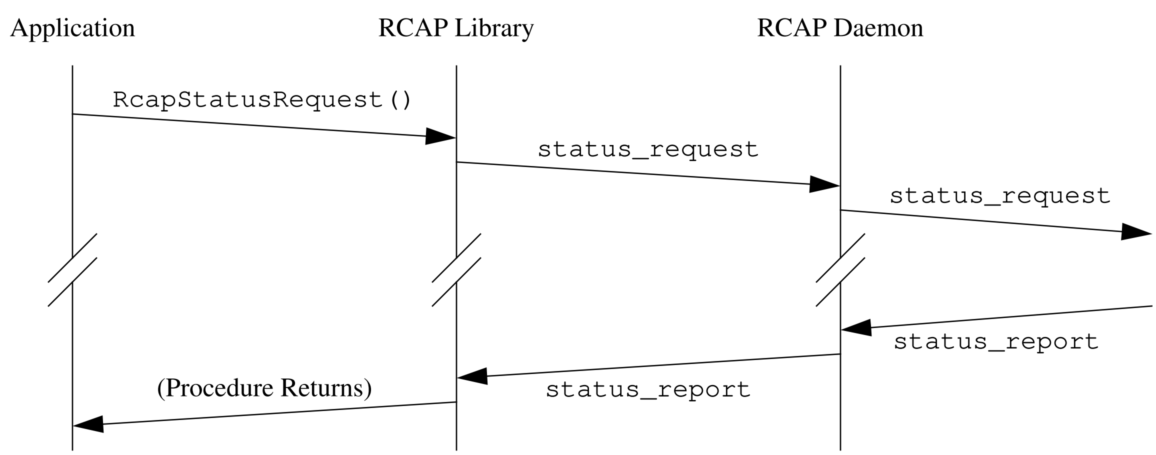

FIGURE 8. Channel Status Reporting

The sending application initiates this process using the library call RcapStatusRequest(). The RCAP library generates a status_request message, which it then passes to the local RCAP daemon. At each node in the network, the local channel parameters (such as buffer allocation and local deadline) are marshalled into a record, which is then appended to the message. The status_request message is then forwarded to the next downstream node.

The RCAP daemon at the channel's destination node "turns around" the message, changing it into a `status_report` message, which is then passed unchanged, via the RCAP daemons, back upstream towards the channel's source. The daemon at the source node then passes the message to the generating RCAP library; this act allows the application to unblock and continue execution.

The status information is returned to the sending application in the form of records in a buffer. The application is responsible for interpreting the contents of the records. The structure of the records is network-dependent. In the case of a heterogeneous internetwork, the application may need to interpret several different kinds of status records. To provide for the possibility that an application may not be able to interpret all types of records, they are structured so that the application can ignore those it cannot understand.

The usefulness of this status information may appear limited, since the application needs to be able to interpret the records for each and every network type. However, we note that the status information is generally so specialized with respect to each kind of network (for example, the local scheduling parameters at each node) that any kind of network-independent summary would be meaningless. As this facility is only intended to aid in network and protocol testing, we accepted this minor shortcoming.

7.4 Channel Teardown

Channel teardowns may be initiated either by the sender or by the receiver. The only operational difference is the control messages exchanged (due to the implicit message direction of each message type). The actions involved in a channel teardown are diagramed in Figure 9.

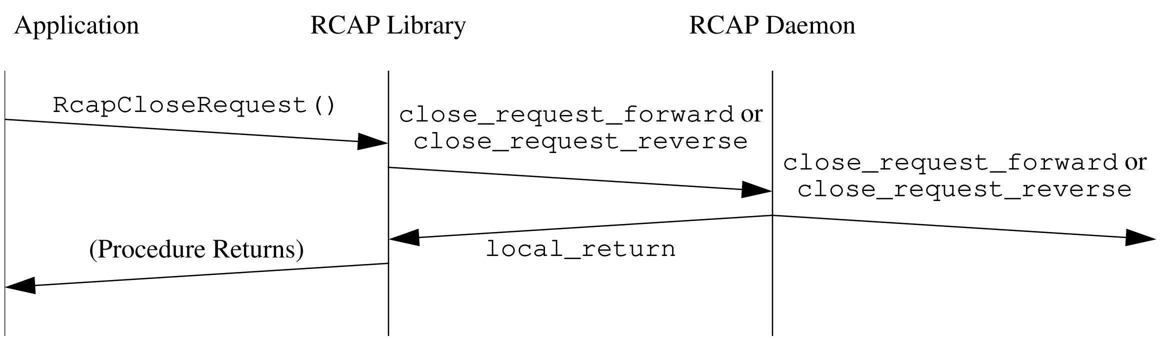

FIGURE 9. Channel Teardown

The application initiating the teardown makes an `RcapCloseRequest()` library call, passing the appropriate channel identifier. The RCAP library then sends either a `close_request_forward` message (if the sender initiated the teardown) or a `close_request_reverse` message (if the receiver initiated the teardown) to the local RCAP daemon.

The daemon then releases the resources it had allocated to the channel and forwards the message in the appropriate direction. The process repeats until the teardown message has reached the opposite end of the path. No confirmation message is sent or required.

8.0 Conclusion

In order to properly manage network resources and connections, a communication service offering performance guarantees requires a mechanism to manage its real-time channels. In this paper we described the functionality and interfaces to a channel administration mechanism for the Tenet Scheme for real-time communications. We then described the decomposition, structure, and operation of the prototype implementation of the Real-Time Channel Administration Protocol, which provides the channel administration functions in the Tenet Real-Time Protocol Suite.

9.0 Acknowledgments

The author gratefully acknowledges the assistance of Professors Domenico Ferrari and Randy Katz for their helpful comments and suggestions, also the support and ideas of the past and present members of the Tenet Group. Special thanks to Anindo Banerjea, co-designer and co-implementor of RCAP, for his insights and efforts.

10.0 References

[BanMah91a] A. Banerjea and B. Mah. “The Design of a Real-Time Channel Administration Protocol”, unpublished report, University of California at Berkeley and International Computer Science Institute, Berkeley, California, May 1991.

[BanMah91b] A. Banerjea and B. Mah. “The Real-Time Channel Administration Protocol”, *Proc. Second Int’l. Workshop on Network and Operating System Support for Digital Audio and Video,* Heidelberg, Germany, November 1991.

[Dasgupta90] P. Dasgupta, R. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc, W. Appelbe, J. Bernabéu-Aubán, P. Hutto, M. Khalidi, and C. Wilkenloh. “The Design and Implementation of the Clouds Distributed Operating System”, *USENIX Computing Systems,* Vol. 3, No. 1, Winter 1990.

[Ferrari90] D. Ferrari. “Client Requirements for Real-Time Communication Services”, *Request For Comments 1193,* November 1990.

[Ferrari92] D. Ferrari. “Real-Time Communication in an Internetwork”, Report TR-92-001, International Computer Science Institute, Berkeley, California, January 1992.

[FerVer89] D. Ferrari and D. Verma. “A Scheme for Real-Time Channel Establishment in Wide-Area Networks”, *IEEE Journal of Selected Areas on Communications SAC-8,* April 1990.

[Fraser92] A. Fraser, C. Kalmanek, A. Kaplan, W. Marshall, and R. Restrick. “Xunet 2: A Nationwide Testbed in High-Speed Networking”, *Proc. INFOCOM ’92,* Firenze, Italy, May 1992.

[Leffler89] S. Leffler, M. McKusick, M. Karels, J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

[Lowery91] C. Lowery. “Protocols for Providing Performance Guarantees in a Packet-Switching Internet”, Report TR-91-002, International Computer Science Institute, Berkeley, California, January 1991.

[PaZhFe92] C. Parris, H. Zhang, and D. Ferrari. “A Mechanism for Dynamic Re-routing of Real-time Channels”, Report TR-92-053, International Computer Science Institute, Berkeley, California, August 1992.

[VerZha91] D. Verma and H. Zhang. “Design Documents for RTIP/RMTP”, unpublished report, University of California at Berkeley and International Computer Science Institute, Berkeley, California, May 1991.

[WolMor91] B. Wolfinger and M. Moran. “A Continuous Media Data Transport Service and Protocol for Real-Time Communication in High Speed Networks”, *Proc. Second Int’l. Workshop on Network and Operating System Support for Digital Audio and Video,* Heidelberg, Germany, November 1991.