SRP: A RESOURCE RESERVATION PROTOCOL FOR GUARANTEED-PERFORMANCE COMMUNICATION IN THE INTERNET

David P. Anderson [1, 2]
Ralf Guido Herrtwich [2]
Carl Schaefer [2]

[1] University of California at Berkeley
Department of Electrical Engineering and Computer Science
Computer Science Division
Berkeley, CA 94720, USA

[2] International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704, USA

ABSTRACT

This report describes the *Session Reservation Protocol* (SRP). SRP is defined in the DARPA Internet family of protocols. It allows communicating peer entities to reserve the resources, such as CPU and network bandwidth, necessary to achieve given performance objectives (delay and throughput). The immediate goal of SRP is to support ''continuous media'' (digital audio and video) in IP-based distributed systems. However, it is applicable to any application that requires guaranteed-performance network communication.

The design goals of SRP include 1) independence from transport protocols (SRP can be used with standard protocols such as TCP or with new real-time protocols); 2) compatibility with IP (data packets are not modified); 3) a host implementing SRP can benefit from its use even when communicating with hosts not supporting SRP.

SRP is based on a workload and scheduling model called the *DASH resource model*. This model defines a parameterization of client workload, an abstract interface for hardware resources, and an end-to-end algorithm for negotiated resource reservation based on cost minimization. SRP implements this end-to-end algorithm, handling those resources related to network communication.

This work was sponsored in part by the California MICRO program, AT&T Bell Laboratories, the Digital Equipment Corporation, Olivetti S.p.A., and the Hitachi Corporation.

1. INTRODUCTION

Audio and video (or *continuous media*) can greatly increase the effectiveness and range of a user interface. It will soon be possible to equip average workstations with the hardware to handle digital continuous media [1] and to connect these workstations by high-speed wide-area networks capable of handling continuous-media data [2]. This hardware base can support distributed continuous-media applications like video conferencing systems and the real-time display of video data stored on a remote file server [3].

Such applications transmit continuous data streams between hosts, and impose performance constraints (delay and throughput) on this transmission. The performance of current hardware, such as CPUs, networks, and disks, is generally sufficient for continuous-media data. However, the scheduling policies used in hosts and gateways are designed primarily for fairness and simplicity. Therefore, especially during periods of heavy system load, the performance of a given connection may fail to meet the application's requirements.

To remedy this situation, a new approach to resource scheduling with the following properties is needed:

- Resources (CPU, network, disk, *etc.*) that can potentially become bottlenecks must be scheduled in a way that allows ''reservations'' (associated with performance guarantees) to be made to individual clients.
- In making a reservation, clients must specify their workload. This is only possible if the workload is known when the reservation is made, *i.e.*, if the software generating the workload operates in a deterministic, time-invariant fashion.
- Since data may traverse resources on several hosts, a protocol for distributed resource reservation is needed. This protocol in turn requires a uniform interface to the various resources involved.

In this report we describe a resource reservation protocol called SRP (*Session Reservation Protocol*) that allows performance guarantees to be made for communication based on IP [4]. SRP can be viewed as a ''network management protocol'' operating at the internetwork (IP) layer as shown in Figure 1. SRP is directly responsible for reserving only network resources. However, it is designed to function as part of a larger framework in which other computer resources (disks, DSP chips, *etc.*) are reserved and scheduled together with network resources. The design goals of SRP also include the following:

- Independence from transport protocols (SRP can be used with standard protocols such as TCP or with new real-time protocols).
- Compatibility with IP. Header fields of IP packets are not added or modified. On hosts that implement SRP, however, IP is modified so that the relative priority of incoming packets can be established.
- A host implementing SRP can benefit from its use even when communicating with hosts not supporting SRP.

The DARPA Internet framework provides both advantages and disadvantages for real-time communication. The datagram abstraction of the IP layer is compatible with continuous-media applications, which often do not require reliability and could not, in fact, tolerate the delays caused by retransmissions in reliable link-layer protocols. IP also has the advantage of widespread use. However, the dynamic routing generally used by IP implementations poses problems for real-

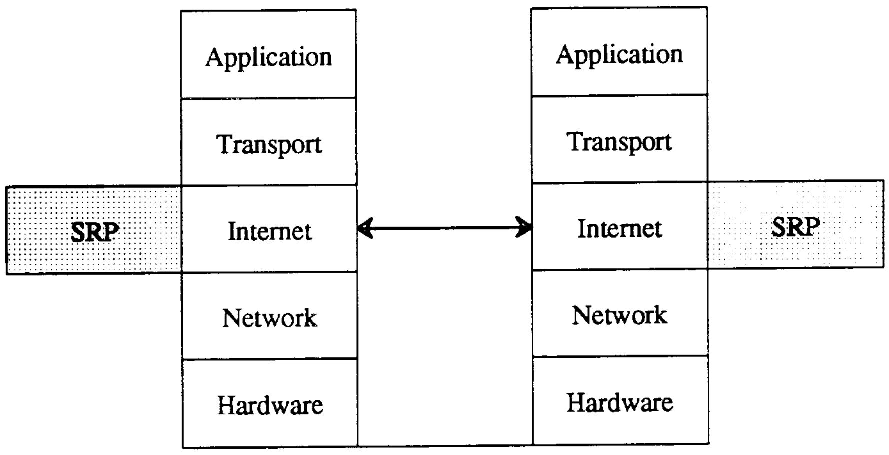

Figure 1: SRP as a management protocol in the Internet protocol hierarchy.

time communication.

TCP [5], the dominant stream transport protocol in IP networks, is not especially well-suited to continuous media applications. For continuous-media data, regular delivery is at least as important as reliable delivery, and sometimes more important. It is possible to use TCP for continuous-media data; however, its main features (flow control and error recovery) are not useful in this context. They may actually interfere with timing, and increase workload, in ways that are detrimental to real-time performance. We do not concern ourselves here with the design of transport protocols to be used in conjunction with SRP.

The remainder of this paper is organized as follows. Section 2 deals with the workload and scheduling model on which SRP is based. A generic interface to resources is presented in Section 3. Section 4 specifies the actual SRP protocol. The interaction between IP and SRP is described in Section 5. Section 6 describes an extension of SRP that accommodates hosts in the communication path that do not implement SRP.

2. THE DASH RESOURCE MODEL

To formalize the reservation of resource capacity, a model for expressing workload and processing is needed. The model used in SRP is called the *DASH resource model* [6]. In this model, the set of system components that handle continuous-media data is decomposed into a set of *resources*. In general, a resource corresponds to a schedulable hardware device and its accompanying software driver. For example, a CPU and its scheduler might comprise a resource. Resources may also be more complex: a local area network (which includes multiple interface devices, concurrent operation, and multiple scheduling mechanisms) might be treated as a single resource.

The DASH resource model assumes that work is assigned to resources in discrete units called *messages*, typically representing a segment of continuous-media data. Each message has a well-defined *arrival time* at which it is available for handling by a resource and *completion time* at which the handling is finished.

The flow of continuous-media data is considered to consist of linear simplex streams of messages that pass through one or more resources. Data is generated by a *source resource* (a disk, digitizer, or compression unit), is then processed by a sequence of *handler resources* (networks, CPUs, etc.) and finally is consumed by a *sink resource* (disk, decompression unit, etc.). A message's completion time in one resource is its arrival time at the next resource. Many of these simplex data streams may exist concurrently, even within a single application. Therefore this scheme encompasses many continuous-media applications: playback of continuous media from disk, storage on disk, live conversations between human users, and others.

2.1. Linear Bounded Arrival Processes

Each data stream flowing across an interface defines an *arrival process* into the downstream resource. To describe the message arrival, the DASH resource model uses *linear bounded arrival processes* (LBAPs), an abstraction introduced by Cruz [7]. An LBAP has the following parameters:

maximum message size	$S_{\max}$	(bytes)
maximum message rate	$R_{\max}$	(messages/second)
maximum burst size	$B_{\max}$	(messages)

In any time interval of length t, the number of messages arriving at the interface may not exceed

$$B_{\max} + t\,R_{\max}$$

The long-term data rate of the LBAP is

$$S_{\max} R_{\max}$$

bytes per second. The burst parameter $B_{\max}$ allows short-term violations of this rate constraint, modeling programs and devices that generate "bursts" of messages that would otherwise exceed the rate constraint.

We define a function *b(m)* representing the *logical backlog* of the arrival process. This is the number of messages by which the arrival process is "ahead of schedule" (relative to its long-term rate) when message m arrives. The logical backlog is not necessarily the number of queued messages since a resource may process messages upon their actual arrival if it is fast enough. *b(m)* is defined by

$$b(m_0) = 0$$
$$b(m_i) = max(0,\ b(m_{i-1}) - (t_i - t_{i-1}) R_{\max} + 1)$$

where t_i is the arrival time of message m_i.

Using *b(m)*, we define the *logical arrival time*, *l(m)*, of a message m as

$$l(m_i) = t_i + \frac{b(m_i)}{R_{\max}}$$

Intuitively, *l(m)* is the time m would have arrived if the LBAP strictly obeyed its maximum message rate.

2.2. Sessions

The use of a resource by a particular data stream is called a *session.* A session represents a reservation of part of the capacity of the resource. Clients must request sessions with all of the resources they need (using a scheme defined below) prior to sending messages. As part of the reservation, the client must specify its workload; in return, the resource provides a bound on the delay it will impose. The client can then decide if this delay is sufficient for its purpose.

Each session has associated sets of LBAP parameters for its input and/or output interfaces. A handler resource accepts LBAPs, producing output LBAPs. The client of the resource must enforce the input LBAP parameters; the scheduler of the resource must enforce the output parameters. We assume that handlers do not modify the message stream, *i.e.*, that they do not lose messages, change the size of messages, speed up or slow down the message stream. Therefore, the incoming and outgoing LBAP for handler resources must have the same values for $S_{\max}$ and $R_{\max}$. On the other hand, incoming and outgoing LBAP may have different burst sizes, depending on the overall workload and scheduling policy of the resource.

In addition to their LBAP specifications, handler sessions also have the following parameters:

maximum logical delay	$L_{\max}$
minimum actual delay	$A_{\min}$
maximum buffered delay	$M_{\max}$

The *actual delay* of a message m in a handler resource is the time interval between its arrival at the input interface and its arrival at the output interface. The *logical delay* of m is the interval between the m's logical arrival time and its logical arrival time at the output interface. Logical delay, rather than actual delay, determines end-to-end delay bounds. The *buffered delay* is the portion of actual delay during which the message is not stored in host memory. In resources such as wide-area networks, $M_{\max}$ will be smaller than $L_{\max}$ due to message propagation time. $A_{\min}$ and $M_{\max}$ are used to calculate buffer space needs.

Two *classes* of sessions are distinguished, *guaranteed* and *best-effort.* For guaranteed sessions, a resource reservation is made, and the delay parameters hold unless a failure occurs. For best-

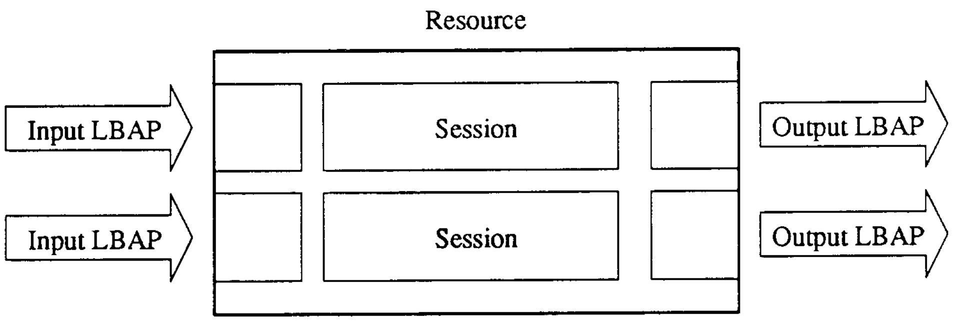

Figure 2: A handler resource with two sessions.

effort sessions, no reservation is made. The workload parameters are "hints" to the resource, the delay parameters are hints to the client. If the resource becomes loaded messages may be dropped and delays may be exceeded.

When data traverses a sequence of resources, the *basic sessions* within the resources are said to form an *end-to-end session.* An end-to-end session represents a unidirectional point-to-point communication path that traverses several resources, either within a single host or across a network. The output interface of each resource in an end-to-end session is the input interface of the next resource. Each resource must be prepared to handle the burst size generated by the previous resource. The *end-to-end logical delay* of a message is the interval between its logical arrival time at the source output and its logical arrival time at the sink input. The maximum logical delay of an end-to-end session is the sum of the maximum logical delays of its handler resources.

2.3. An Economic Approach to Delay Allocation

It may be possible for a resource to guarantee a maximum delay anywhere within a certain range. The shorter the delay is, the more costly it is because the resource has less freedom to schedule other service requests – and the fewer additional sessions it can support. To divide the delay between resources in an end-to-end session the DASH resource model takes an approach based on economics. This approach has also been used for problems such as routing and load-balancing [8].

When a client reserves a session with a resource, the resource makes reservations for the smallest possible maximum delay. In addition, the resource provides a *cost function* indicating, for each larger maximum delay, the associated cost to the client. The client may relax the maximum resource reservation to minimize cost.

Cost can either be real money, to be later billed to the client, or some metric reflecting the resource's current load. The cost function may be a function of the workload of a resource or of parameters like the time of day or the identity of the user (*e.g.*, so that frequent users pay less).

For tractability, the DASH resource model requires that every cost function be 1) continuous and piecewise linear; 2) strictly monotonic decreasing, and 3) convex. In other words, for each vertex (d_i, c_i) and its surrounding elements we have

$$d_{i-1} < d_i < d_{i+1}, \quad c_{i-1} > c_i > c_{i+1}, \quad \frac{c_{i-1} - c_i}{d_i - d_{i-1}} \geq \frac{c_i - c_{i+1}}{d_{i+1} - d_i}$$

The first value for which a cost is given is the smallest achievable maximum delay. We assume that at some point more delay will not lead to lower costs because the cost of buffering messages over the delay period will exceed the cost saved by the larger delay.

The cost of an end-to-end-session is the sum of the costs of its component sessions. Since we have defined cost functions to be convex piecewise linear functions, they can be combined by the following procedure: The segments of the functions are sorted in order of decreasing (more negative) slope. They are placed end-to-end, starting at the point which is the sum of the initial endpoints of the functions. This procedure is illustrated in Figure 3.

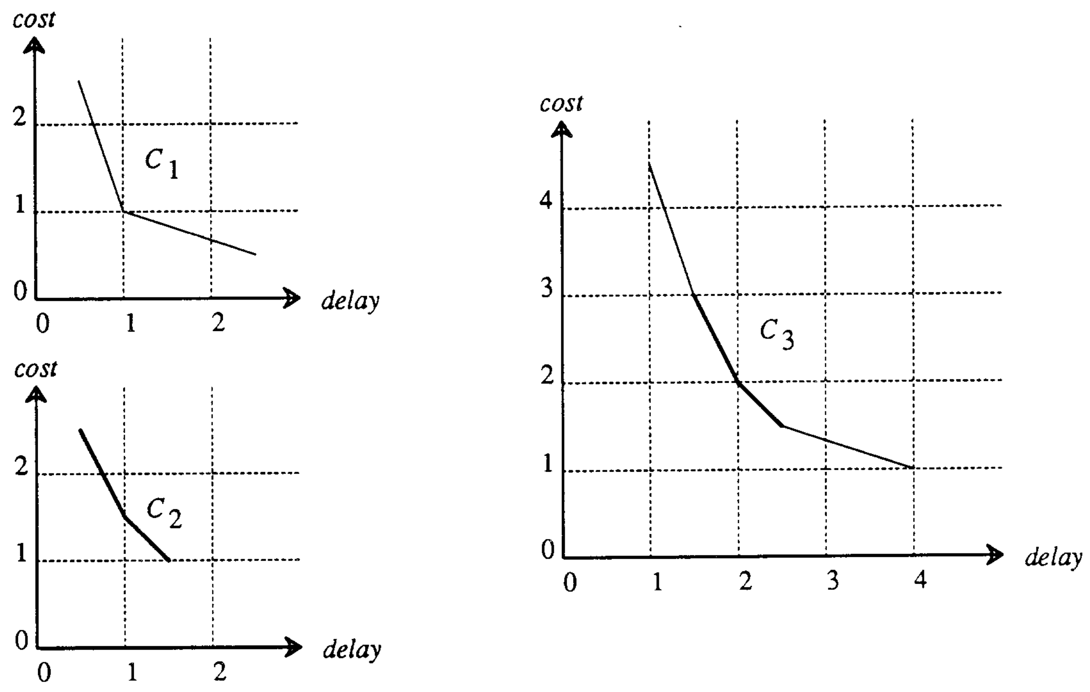

Figure 3: Combining cost functions; C_3 is the combination of C_1 and C_2.

2.4. Buffer Reservation

The DASH resource model is designed to prevent message loss due to buffer overflow. For a given resource, this requires reserving enough buffer space to accommodate the input burst size plus messages being processed in the host. In bytes, this amount of buffer space is given by the expression

$$S_{\max}(B_{\max} + R_{\max} M_{\max})$$

When several sessions on a given host are part of an end-to-end session, a formula similar to the above gives a tighter bound on the number of buffers needed for the entire chain of sessions:

$$S_{\max}(B_{\max} + R_{\max} \overline{M}_{\max})$$

The input burst size $B_{\max}$ is that of the first session in the chain, and the maximum buffered delay $\overline{M}_{\max}$ is summed over the basic sessions.

A second need for buffer space arises when the receiving end of an application must deliver messages at a constant rate to the output device (*e.g.*, audio or video converters). Suppose the first message of a stream arrives with minimum delay. If the application outputs the first message immediately, and the second arrives with maximum delay, there will be an unacceptable pause in the output between the two. The application must therefore buffer messages to ensure that there is no "jitter" in the output.

Assuming that the source resource generates messages fast enough to maintain a nonzero backlog, jitter can be avoided as follows. The receiver waits until

$$R_{\max}(\bar{L}_{\max} - \bar{A}_{\min})$$

messages have been received (where $\bar{L}_{\max}$ and $\bar{A}_{\min}$ are the sums of $L_{\max}$ and $A_{\min}$ over all sessions in the end-to-end session), and then waits until the logical arrival time of the last of these messages. If $O_{\max}$ is the output burst size of the last session, the number of bytes needed for buffering is

$$S_{max}(O_{\max} + R_{\max}(\bar{L}_{\max} - \bar{A}_{\min}))$$

2.5. End-to-End Session Establishment

The DASH resource model defines an establishment protocol for end-to-end sessions. Using this protocol, the application's allowable end-to-end delay is divided between the resources, and burst sizes are established. The establishment protocol is carried out by *host resource managers* (HRMs).

Initially, the HRM at the source host is given a client request that specifies the resources involved, the message size and rate, and the end-to-end delay requirements. These requirements are given by a *target* and *maximum* value, denoted E_{target} and $E_{\max}$. We do not specify how clients learn about their delay requirements. They could negotiate them immediately prior to establishing the session. If we assume that the decision factor for the suitability of a session is the delay rather than the costs associated with it, it is possible to cancel the session establishment even before contacting the receiving client, should it become clear that not even $E_{\max}$ can be achieved. E_{target} makes a "fast" session establishment possible: the receiving client does not need to be contacted before a session is established, yet the sending client can specify its preferences using E_{target}.

The protocol has two phases:

(1) The first phase traverses the hosts from the source to the sink. A *request* message is exchanged between HRMs. The request message contains the data message size and rate, the client delays E_{target} and $E_{\max}$, the burst size from the previous host, the cumulative sums of $L_{\max}$ and $A_{\min}$, and the cumulative cost functions. Maximum reservations are made for each resource, and corresponding buffer space is reserved.

(2) The second phase proceeds in the reverse direction. The receiving client evaluates the end-to-end session parameters and decides on a delay for the session. A *reply* message containing the remaining excess delay (see below) and the burst size into the next host is passed back towards the source. For each resource, the session parameters are relaxed appropriately. The delay may be increased and additional buffers may be reserved both for this purpose and to accommodate larger input bursts. Delays are relaxed only up to the amount of buffer space available.

The protocol can fail for a number of reasons: The communication between the HRMs may not be successful, a resource reservation may fail, the delay of the end-to-end session may be unacceptable for the receiving client or there may not be enough buffer space available. In these cases, a failure message is propagated back to the sending client and reservations which have already been made are cleared. Note, that the relaxing of a delay in the second phase may make more buffers necessary – and these buffers may not be available. Instead of considering the session establishment to have failed (and sending a failure message to both clients), we assume that the delays are relaxed only up to the amount of buffer space available. The resource then provides a better service at no extra cost.

Let E_{actual} be the actual end-to-end logical delay obtained in the first phase of the session establishment. If this delay is less than E_{target}, some *excess delay* E_{excess} defined as

$$E_{\text{excess}} = E_{\text{target}} - E_{\text{actual}}$$

can be distributed among the resources. This should be done as economically as possible, *i.e.*, in a way which saves the largest amount of money.

In the second phase of the protocol, each HRM hands the remaining excess delay to the previous one. Each HRM knows the outgoing accumulated cost function and the cost functions of all local resources. It shifts the outgoing cost function to the left, so that the first cost value is given for 0. From this cost function the segments from 0 to E_{excess} are examined. If any of these correspond to segments of local cost functions, the corresponding resources are relaxed by the time-extent of the segment, provided there is enough buffer space available. If E_{excess} lies in the middle of a segment, the amount of the relaxation is the part of the segment that lies to the left of E_{excess}. Any excess delay that is not returned to local resources is passed back to the previous host.

3. RESOURCE MANAGEMENT

In order to establish end-to-end sessions, basic sessions with resources have to be established first. For this purpose, every resource needs to have a *session manager* through which sessions can be established and deleted. In this section we present a model interface for session management in terms of C++ function prototypes and briefly consider how decisions about the establishment of new sessions are made.

3.1. Session Manager Interface

The session manager interface provides three functions: `reserve()`, `relax()`, and `free()`. `reserve()` is used to establish sessions, delivering the best achievable performance guarantees and a cost function as described in Section 2. This reservation can be adjusted using `relax()`. Finally, `free()` deletes a session.

The `reserve()` function delivers the smallest possible minimum and maximum delays for a specified workload. These delay guarantees are based on the workload specification, on the already existing workload of the resource and on its scheduling algorithm. It also delivers an output burst size which is calculated by the session manager from the input burst size and the scheduling algorithm. A cost function is provided by the session manager to inform the reserving entity about cost savings if it relaxes its reservation to allow a larger delay.

```
int Session_manager::reserve (
   // INPUT
   Size         size,            // maximum message size
   Rate         rate,            // rate
   Burst        in_burst,        // input burst limit
   Class        class,           // guaranteed or best-effort
   ...
   // OUTPUT
   B_session*   bsid,            // basic session ID
   Time*        max_log_delay,   // maximum logical delay
   Time*        min_act_delay,   // minimum actual delay
   Time*        max_buf_delay,   // maximum buffered delay
   Burst*       out_burst,       // output burst limit
   Delaycost*   delaycost        // cost function
);
```

The ellipsis indicates that particular resources may have additional arguments to their `reserve()` functions. These parameters are needed to determine the delay values. For example, the time for which a single work item of a session occupies a resource has to be given explicitly for a CPU resource because the process execution time cannot be deduced from any other parameter. For a network resource, on the other hand, it can be deduced from the message size. In case of a network resource the destination address of the connection needs to be known to determine the propagation time of a message.

To any resource, a basic session is identified by a *session ID* that is unique (for the life of the session) to that particular resource.

```
typedef int B_session;           // locally unique session IDs

enum Session_class
   { BEST_EFFORT, GUARANTEED };

typedef int   Size;              // bytes = octets
typedef float Time;              // seconds
typedef float Rate;              // messages/second
typedef int   Burst;             // messages
```

The piecewise linear cost function is defined by the endpoints of its linear segments. If the function has only one element, only one delay is possible.

```
typedef int Cost;                // cost units
struct Point {                   // point of cost function
   Time         max_delay;       // maximum delay
   Cost         cost;            // cost of this delay
};
struct Delaycost {
   Point        point;           // this point
   Delaycost*   next;            // next point
};
```

`reserve()` returns one of the following values:

SUCCESS	0	reservation ok
FAIL_USAGE	-1	bad parameters
FAIL_PERM	-2	permanent rejection
FAIL_TEMP	-3	temporary rejection

A reserving entity may increase the maximum delay for a resource and the output burst size by means of `relax()`. Depending on the scheduling algorithm, a larger output burst may allow a larger input burst. It is always possible to relax a delay. The relaxing entity must ensure beforehand that sufficient buffer capacity is available. Since the scheduling algorithm may not generate output bursts as large as specified in the function parameters, `relax()` returns the actual maximum output burst.

```
int Session_manager::relax (
    // INPUT
    B_session     bsid,              // basic session ID
    Time          new_max_log_delay, // requested logical delay
    Burst         new_out_burst,     // requested output burst limit
    // OUTPUT
    Burst*        out_burst,         // actual output burst limit
    Burst*        in_burst           // new input burst limit
);
```

The function returns `FAIL_USAGE` instead of `SUCCESS` if the session has not been established before, or if the specified maximum delay or output burst limits are smaller than those previously returned by `reserve()`.

`free()` deletes a session.

```
int Session_manager::free (
    // INPUT
    B_session     bsid               // basic session ID
);
```

The function returns `FAIL_USAGE` if the session does not exist, `SUCCESS` otherwise.

3.2. Session Establishment Decision Algorithms

When establishing a new session, the session manager has to ensure that the workload of this session does not lead to a violation of previously given guarantees. How this "non-interference" can be determined is highly resource-specific, but any policy for which an upper bound on delay can be derived from given input LBAP parameters can be used. For example, *round-robin*, *FIFO*, *rate-monotonic* and *earliest-deadline-first* scheduling all have this bounded-delay property and the resource interface can be successfully implemented on top of them.

Many existing results in real-time scheduling can be applied [9], especially if we restrict our attention to resources that consist of a single hardware device (*e.g.*, a CPU). For example, if guaranteed delay can always equal the interarrival time of messages on a session, a simple test for preemptive *rate-monotonic* scheduling of a singular resource would be

$$\sum_{s \in E} R_{\max}(s)\, T_{\max}(s) \le |E|\,(2^{\frac{1}{|E|}} - 1)$$

where E is the set of all established sessions (including the new one) and $T_{\max}$ is the *maximum service time* for each message. Under the same conditions,

$$\sum_{s \in E} R_{\max}(s)\, T_{\max}(s) \le 1$$

is a sufficient non-interference condition for preemptive *earliest-deadline-first* scheduling [10]. Worst-case simulation provides a more general decision procedure [11].

For resources that encapsulate more than a single device, the management procedures are more complicated. For example, the sources of delay in an FDDI network (viewed as a single resource) include queuing, media access, and propagation, and many scheduling policies are possible. Establishing a session in such a resource may involve using network management protocols to reserve network bandwidth in ''synchronous'' or ''isochronous'' channels. The question of how to support sessions in such a resource is a subject of ongoing investigation.

4. THE SESSION RESERVATION PROTOCOL

SRP is a mechanism to achieve performance guarantees for communication in the Internet. It is used in the process of establishing an end-to-end session with resources involved in an IP-based communication between a sending and a receiving client. This end-to-end session is associated with a connection of a particular IP-based protocol (for example, a TCP connection). The performance guarantees of the end-to-end session apply to the data traffic from the sending to the receiving client on the associated connection.

SRP is responsible for reserving (and relaxing) the following resources :

- On the sending host, SRP reserves the network resource through which messages leave the host.
- On a gateway, SRP reserves the CPU resource needed for protocol operation and the network resource through which messages leave the gateway.
- On the receiving host, SRP makes no reservations, and simply conveys the session request to the receiving client. (The incoming network resource was reserved by the previous host in the path.)

The clients of SRP are responsible for reserving all other resources that handle the stream of continuous-media messages (for example, the reservation of an input or output device). Thus, on the sending and receiving hosts the HRM function is divided between the sending client and SRP. We assume that the set of resources involved in a connection is always fixed, *i.e.*, that the connection is based on a static route through the network. How this can be achieved for IP-based communication will be described in Section 5.

Every resource reservation by SRP includes a corresponding reservation of buffer space. SRP reserves no buffer space to avoid jitter in the output, but it provides the receiving client with information about the delay so that the client can implement jitter avoidance.

4.1. Operation of SRP

A typical scenario for establishing and end-to-end session using SRP is the following (illustrated in Figure 4):

(1) The sending and receiving clients set up a transport-level connection. The performance goals for the connection are determined, and a globally unique end-to-end session ID is obtained.

(2) The sending client reserves all resources related to the data source (disk, camera, VCR, *etc.*) and the CPU resource. The service times of the CPU reservation must include the requirements of all software modules used by the application, *i.e.*, not only the application itself, but also the file service, the network protocols, *etc.* The client can learn about

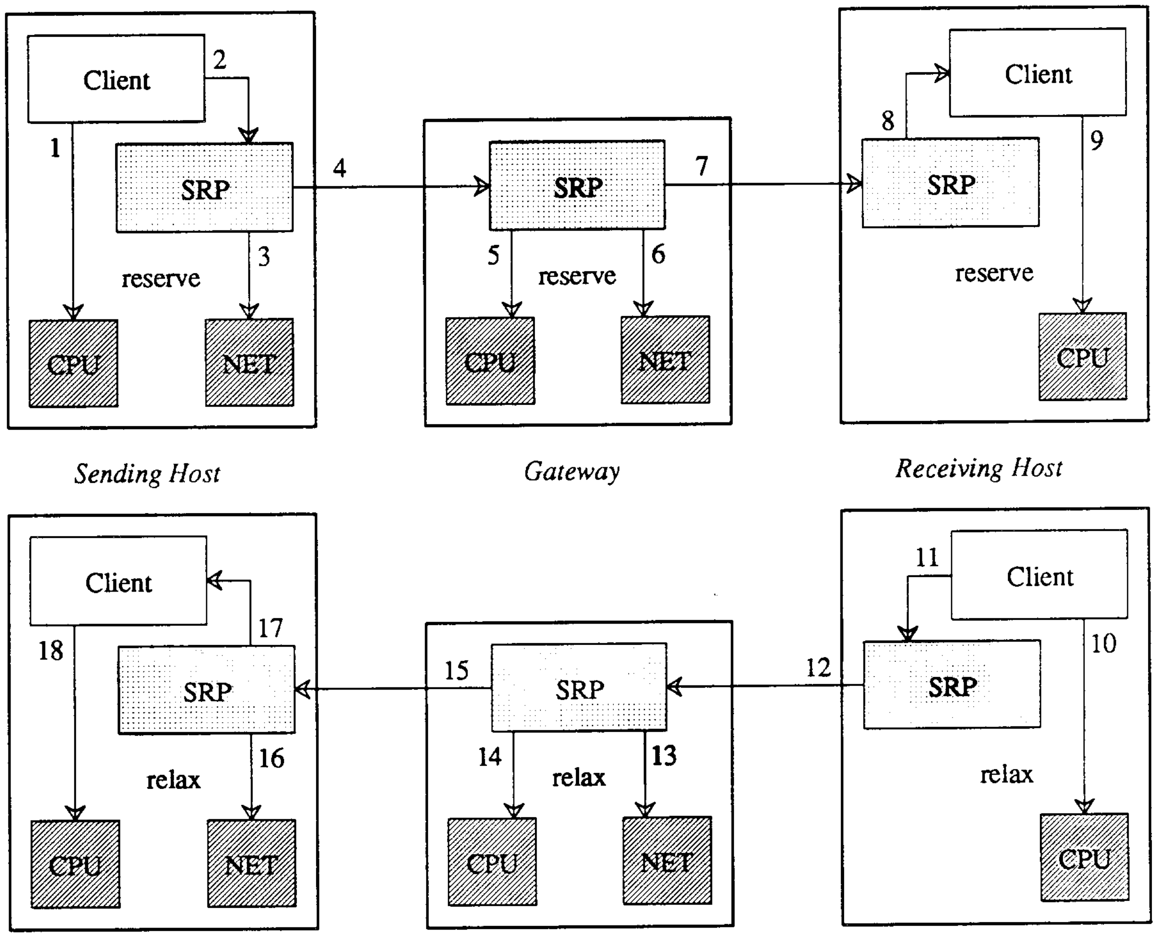

Figure 4: Scenario illustrating the operation of SRP.

these requirements, *e.g.*, by calling a corresponding function of the protocol module it uses (which in turn can call a function of the protocol *it* uses, and so on). The sending client then calls the local SRP module.

(3) The SRP module on the sending host examines the address of the receiving client and consults the IP routing table to determine the network a message on the associated connection is sent on and the next node within the connection. This node may either be the receiving client itself or a gateway. The SRP module then reserves the appropriate network resource and the necessary buffer space, and forwards the end-to-end session request to the next hop.

(4) On a gateway, the SRP module reserves both CPU and network resources and the corresponding buffer space. To make a CPU reservation, it first determines the maximum service time of the gateway software, which depends on the incoming and outgoing network of a connection. It then makes a reservation for the outgoing network just as

described in the previous paragraph.

(5) The SRP module at the receiving host notifies the receiving client, which reserves the CPU resource and the remaining resources related to the data sink (conversion hardware, *etc.*).

(6) The receiving client compares its own requirements and those of the sending client with the obtained smallest possible maximum delay. If the obtained guarantees exceed the requirements, the receiving client relaxes the reservations of local resources, and returns the remaining excess delay to the SRP module.

(7) SRP completes the backward pass of the end-to-end session establishment algorithm through the gateways, relaxing reservations according to cost functions and remaining excess delay. It also adjusts burst sizes.

(8) The SRP module on the sending host relaxes the network resource, then returns to the sending client. The client relaxes its local resource reservations, completing the session establishment.

An SRP session establishment can fail due to communication failures or inadequate resources, in which case all reservations which have already been made for this session are canceled, and a failure indication is returned to the sending client. It is then up to the sending client to inform the receiving client that the performance guarantees for their connection cannot be achieved.

Once a session has been established, the sending client can begin sending data. As IP datagrams arrive at each host (gateways and receiver) the IP modules at these host associate these datagrams with the corresponding end-to-end session, and pass this information to the schedulers of the resources involved (CPU, network, etc.). Thus the end-to-end performance guarantees are achieved.

A session can be deleted by the client that has established it. The sending client contacts its local SRP module, which deletes its session with the network and notifies the SRP module on the next host. For this purpose, every SRP module has to keep information about the sessions it has already established and the next SRP module to be contacted. On a gateway, SRP deletes the basic sessions with both the CPU and the network and forwards the request. The last SRP module on the receiving host sends back an acknowledgement through the chain of SRP modules. Upon sending the acknowledgement, SRP modules can discard all information which has been stored for the purpose of end-to-end session administration. The receiving client is not informed by its SRP module when the session is deleted; this is the duty of the sending client.

4.2. Protocol Elements of SRP

SRP modules communicate using the Sun RPC protocol [12]. This protocol makes use of either the UDP or TCP protocols. Sun RPC offers reliable communication and has a convenient programming interface. It provides authentication mechanisms which could be used to prevent unauthorized users from reserving resources.

Each SRP module implements an RPC program with three procedures: a call to establish sessions, another to delete them, and a mandatory null call taking no parameters and returning no results (for determining round-trip time). The SRP module itself gets an identification number (which has to be assigned by a network authority in order to make it unique). In the Sun RPC interface specification language, the SRP program is as follows:

```
program SRP {
   version Original {
      void null_function (void) = 0;
      E_rep_data session_establish (E_req_data) = 1;
      void session_delete (E_session) = 2;
   } = 1;                           // first protocol version
} = ?;                              // to be assigned
```

The data types of this specification are to be defined in the XDR External Data Representation Standard [13]. The XDR language is similar to C and C++, so that some definitions resemble our previous type specifications. In order to avoid undue repetition, we refer to the definitions from Section 3. However, host-independent data representation is now associated with each definition.

The message formats of a session establishment RPC may vary slightly: In the request message the format depends on the amount of data needed to associate an end-to-end session with a network connection and the size of the cost function. The format of the reply message depends on whether the reply is positive or negative.

The format of a request message is the following, where he cost function is given by a variable-length array rather than a linked list because XDR contains no pointers:

```
struct E_req_data {
   E_session    esid;             // end-to-end session ID
   Address      destination;      // receiving host
   Address      net;              // incoming network
   Connection   connection<>;     // associated connection
   Rate         rate;             // rate
   Burst        in_burst;         // input burst limit
   Size         size;             // message size
   Time         tgt_ete_delay;    // target end-to-end delay
   Time         max_ete_delay;    // maximum end-to-end delay
   Time         acc_max_delay;    // maximum logical delay so far
   Time         acc_min_delay;    // minimum actual delay so far
   Point        acc_cost<>;       // cost function so far
};
```

End-to-end session identifiers contain the Internet address of the sending host to make them globally unique (for the life of the session).

```
typedef long Address;
struct E_session {
   Address      source;           // sending host
   int          id;               // unique number
};
```

The `net` parameter is needed to identify the incoming network if nodes are connected through more than one network.

Each end-to-end session is associated with an upper-level connection. These connections can be defined at any layer in the protocol hierarchy, at the transport level or above. Each connection is therefore identified by a set of data items (connection numbers, addresses, etc.), one per protocol layer. Each IP datagram sent on the connection contains these data items in its various headers. The length of the items, and their position within the IP datagram, depend on the upper-level protocols.

The sending client obtains this protocol-specific "association data" from the protocol layers it uses and passes it to SRP as part of the session establishment request.

```
enum Protocol
   { UDP, TCP, RPC, NFS, ...};
union Connection switch (Protocol proto) {
    case UDP:
       UDP_data udp_data;        // UDP datagram ID
    case TCP:
       TCP_data tcp_data;        // TCP connection ID
    ...
};
```

`UDP_data`, `TCP_data`, *etc.* are used to identify packets of the associated connection within the respective protocol. How this information is used is explained in Section 5.

Reply messages have the following format:

```
enum Rep_code
   { ESTABLISHED, NOT_ESTABLISHED };
struct E_rep_data {
   E_session       esid;        // end-to-end session id
   union Reply switch (Rep_code rep_code) {
      case ESTABLISHED:
         struct {
            Time    excess_delay; // remaining excess delay
            Burst   out_burst;    // acceptable burst
         } pos_info;
      case NOT_ESTABLISHED:
         struct {
            int     failure;      // failure code
            Address culprit;      // host where failure occurred
            char    msg<>;        // name of failing resource
         } neg_info;
   };
};
```

A failure can either be due to SRP or to an RPC failure further down the chain. The failure code can have the following values (chosen according to corresponding values in the RPC protocol):

FAIL_USAGE	-1	bad parameters
FAIL_PERM	-2	permanent rejection
FAIL_TEMP	-3	temporary rejection
FAIL_BUFF	-4	buffers exceeded
FAIL_DELAY	-5	delay exceeded
FAIL_SRP_NO	-11	remote host does not implement SRP
FAIL_SRP_VERS	-12	remote host does not implement this SRP version
FAIL_SRP_PROC	-13	remote host does not implement this SRP procedure
FAIL_RPC	-100	remote host does not implement this RPC version
FAIL_AUTH_BADCRED	-101	authentication error: bad credentials
FAIL_AUTH_RJTCRED	-102	authentication error: expired credentials
FAIL_AUTH_BADVERF	-103	authentication error: bad verifiers
FAIL_AUTH_RJTVERF	-104	authentication error: expired verifiers
FAIL_AUTH_TOOWEAK	-105	authentication error: rejected for security reasons

In addition, the address of the host signaling the failure and a failure message (perhaps the name of the failing resource) are given.

4.3. Application Interface

The application interface of SRP is node-dependent; the following C++ function prototypes merely define a model interface to the SRP service. In a real implementation these functions could be system calls, or they might be in a library.

The interface of the sending client consists of one function which issues the request to establish a session and delivers a reply. It blocks the client until a reply is available. The function to create a session is similar to the session establishment RPC.

```
int SRP::establish_session (
   // INPUT
   E_session     esid,            // end-to-end session ID
   Address       destination,     // receiving host
   Address       net,             // network used
   Association*  assoc_data,      // data to identify connection
   Rate          rate,            // rate
   Burst         in_burst,        // input burst limit
   Size          size,            // message size
   Time          tgt_ete_delay,   // target end-to-end delay
   Time          max_ete_delay,   // maximum end-to-end delay
   Time          acc_max_delay,   // maximum logical delay so far
   Time          acc_min_delay,   // minimum actual delay so far
   Delaycost*    acc_cost,        // cost function so far
   // OUTPUT
   Time*         excess_delay,    // remaining excess delay
   Burst*        out_burst,       // acceptable output burst limit
   Address*      culprit,         // host where failure occurred
   char*         msg              // name of failing resource
);
```

Note, that the client, since it has already reserved local resources, may specify accumulated delays and a cost function. The `excess_delay` and `out_burst` parameters are undefined if a failure occurred; `culprit` and `msg` are undefined if the session establishment was successful. The return code can have the same values as in the corresponding RPC message. `SUCCESS` is returned if no failure occurred.

The following types have not been declared before since the corresponding XDR data structure was defined as a discriminant union which is not provided by C++:

```
union Connect_id {
   UDP_data     udp_data;          // UDP datagram ID
   TCP_data     tcp_data;          // TCP connection ID
   ...
};
struct Association_data {
   Protocol     protocol;          // protocol number
   Connect_id   connect_id;        // connection ID
};
struct Association {
   Association_data assoc_data;    // this connection
   Association*     next;          // higher-layer connection
};
```

The interface of the receiving client consists of three functions, one to accept an indication for session establishment (blocking the client until an establishment request is received), one to deliver a positive acknowledgement and one to issue a negative reply.

```
int SRP::establish_session_ind (
    // INPUT
    E_session     esid,             // end-to-end session ID
    // OUTPUT
    Rate*         rate,             // rate
    Burst*        in_burst,         // input burst limit
    Size*         size,             // message size
    Time*         tgt_ete_delay,    // target end-to-end delay
    Time*         max_ete_delay,    // maximum end-to-end delay
    Time*         acc_max_delay,    // maximum logical delay so far
    Time*         acc_min_delay,    // minimum actual delay so far
    Delaycost*    acc_cost          // cost function so far
);

int SRP::establish_session_ack (
    // INPUT
    E_session     esid,             // end-to-end session ID
    Time          excess_delay,     // remaining excess delay
    Burst         out_burst         // acceptable output burst limit
);

int SRP::establish_session_nack (
    // INPUT
    E_session     esid,             // end-to-end session ID
    int           failure,          // failure code
    char*         msg               // name of failing resource
);
```

The indication function returns FAIL_USAGE if the session already exists, otherwise it returns SUCCESS. Both reply functions return FAIL_USAGE if the session does not exist, SUCCESS otherwise. The failure code returned as a parameter of the negative reply function is one of FAIL_PERM, FAIL_TEMP, FAIL_BUFF or FAIL_DELAY.

Sessions can be deleted at any time by the sending client. It is up to this client to inform its peer.

```
int SRP::delete_session (
    // INPUT
    E_session     esid              // end-to-end session ID
);
```

The function returns FAIL_USAGE if the session does not exist, one of the RPC failure codes should an RPC error occur, and SUCCESS otherwise.

5. SESSION ASSOCIATION IN IP

We want to achieve performance guarantees for conventional IP-based communication without changing the communication protocols involved. This means that data messages cannot contain an explicit identification of the session to which they belong. To correctly schedule guaranteed messages, hosts need to be able to distinguish them from other network traffic and know the parameters of the parent session. This section describes how the IP modules on these hosts can be modified to accomplish this.

5.1. Session Registration

To be able to identify the sessions of incoming IP packets, the IP module at a node must be informed of sessions through the node. Therefore, IP contains a *registration procedure* that is called by SRP when a session is established; it is passed the session ID and the association data obtained from the sending client. To achieve static routing for each same session, the registration call specifies the next machine to which all IP messages of the respective session will be sent, and the network to be used.

```
void IP::register (
    // INPUT
    E_session     esid,              // end-to-end session ID
    Association*  assoc_data,        // data to identify connection
    Address       out_net,           // outgoing network
    Address       next_node          // next node in this session
);
```

SRP calls `register()` on hosts to which incoming packets will arrive (gateways and the receiving host). On the receiving host, the `out_net` and `next_node` parameters are not used.

In practice, `register()` would add new entries to per-protocol "lookup tables" (hash tables keyed by protocol-specific ID fields) for each of the protocols in its association-data array. Each entry in a lookup table corresponds to a connection of the corresponding protocol. Its value is either a session ID (if the protocol is the last element of the association-data array) or identifies the next protocol up in the sequence.

If a session is deleted by the clients its registration is canceled.

```
void IP::unregister (
    // INPUT
    E_session     esid               // end-to-end session ID
);
```

5.2. Session Identification

When an IP packet arrives at a host, the IP module must find what session (if any) the packet is associated with. While various implementation approaches are possible, the following organization is suggested. For every upper-level protocol (UDP, TCP, RPC, NFS, *etc.*), there is an *identification procedure* that, given an incoming packet of that protocol, identifies the SRP session, if any, to which the packet belongs. These procedures have the following interface:

```
int IP::UDP_identify (
    // INPUT
    IP_packet*    packet,            // IP packet to be identified
    // OUTPUT
    E_session*    esid               // end-to-end session ID
);
```

`SUCCESS` is returned if a session is found, in which case `esid` contains the session ID. `NO_SESSION (-1)` is returned otherwise, and `esid` is undefined.

These procedures encapsulate the "lookup tables" maintained for each protocol (see above). They work as follows: The protocol-specific identifiers are extracted from the IP packet and used to hash into the lookup table. If no entry is found, `NO_SESSION` is returned. If the entry contains a session ID, `SUCCESS` is returned. Otherwise the number of the next-higher protocol is obtained from the packet, the identification procedure for that protocol is called, and its value is returned.

For each incoming packet, then, IP extracts the protocol field from the IP header and calls the corresponding identification routine. This is normally done at the interrupt level. If a session ID is found, the scheduling of subsequent handling of the packet (*e.g.*, the deadline of the process that handles it) is based on the logical arrival time of the packet and the session delay. Otherwise scheduling can be done by conventional means, *e.g.*, FIFO.

Outgoing packets on the sending host or on a gateway leave the node through the network resource. They are handled by IP in the following way: We assume that the SRP session ID is known. On the sending host, it can be passed down from the client using an additional parameter of the `send` function; on gateways, it was obtained as above. Again, the session ID is used to calculate the logical arrival time of the message into the network resource, which in turn is used to determine the queuing order of the packet and perhaps the manner in which it is transmitted on the medium.

If higher-layer packets are fragmented, an IP packet may not contain association data. In this case session identification works as follows: Because of static routing all fragments arrive in order. The first fragment has the *more fragments* field set. We assume that this fragment contains the ID of the associated connection (a reasonable assumption since 1) IP packets are sufficiently large, and 2) in all Internet protocols the address field is contained in the packet header). IP calls the appropriate identification procedure for this packet and stores the obtained session ID together with the *packet ID* of the IP packet. Future fragments can be identified by having the same packet ID. Once the last fragment arrives (without the *more fragments* field set), IP can delete the fragment identification information.

In order not to have to call the identification procedure for *every* incoming IP packet, we request that all IP packets belonging to SRP sessions are sent with the *low delay* field set. This field is a hint to IP that a packet may have performance guarantees associated with it.

6. INTERACTING WITH NODES THAT DO NOT IMPLEMENT SRP

If one of the nodes in the route of an IP-based connection does not implement SRP, the end-to-end establishment will fail and a `FAIL_SRP_NO` return code will be delivered to the sending client. In some cases, a *partial session* among nodes supporting SRP is a useful alternative to no session at all. Of course, it is impossible to guarantee a bounded delay for a partial session, but if the performance bottlenecks are within the sites involved in the partial session, or non-SRP systems are sufficiently fast, a partial session may be adequate for a given application. We cannot ignore the fact that quite a few continuous-media system prototypes are available which function surprisingly well without resource reservations (though usually only under light system load or on a single-tasking system).

In this section we extend the SRP protocol presented in Section 4 to establish partial sessions. We keep the messages and client functions used for this purpose separate from the previously presented ones because they express significantly different client requirements: A request for a partial session may be granted with a complete session; the converse is never true. Also, partial session establishment involves additional parameters and algorithms which are not needed for the establishment of complete sessions. In addition, support for partial sessions is an extension of SRP that some systems may choose not to implement.

6.1. Partial Session Establishment

To establish a partial session, at least one of the client nodes must have an SRP module. We call this module the *anchor module*. The establishment of a partial session proceeds from the sending end towards the receiving end, similar to that of a complete session, Therefore, if both sending and receiving clients support SRP, the SRP module on the sending host is the anchor module. Only if the sending host does not implement SRP, will the anchor module be located on the receiving host.

Let us assume the anchor module is located on the sending host and the complete route of the session is known. In this case, after making local reservations, the anchor module sends RPCs to other SRP modules, starting with the second node in the route, until the call is successful. The session parameters and the remaining route are forwarded in this RPC. The contacted SRP module tries to continue the session establishment by contacting the next node of the route which implements SRP.

The accumulation of delay parameters and cost functions takes place just as in the establishment of complete sessions. Burst parameters and used network specifications, however, may not be valid if a previous node did not participate in the session establishment. The values of these parameters are merely hints, and worst-case assumptions for buffer space and service time have to be made. On the way back, output burst relaxations should only be made if the resource offering to accept a larger burst is the immediate successor in the session. If not all immediate successors of a node implement SRP, the *low delay* field can no longer be used as a hint that a packet may belong to an SRP session.

The major problem in the establishment of partial sessions is to determine the nodes involved in this session. There are different solutions for this problem:

(1) The most convenient situation for partial session establishment is if the route is available beforehand, *i.e.*, if the sending client can specify the route a message will take through the network.

(2) Should the route information not be available, the sending client can send an IP datagram with the *record route* option set. Once the datagram reaches the receiving host, every host in between has added its Internet address to the packet. This solution requires that the sending and receiving client have direct access to IP.

(3) If neither of the previous solutions is feasible, SRP modules have to cooperate to determine the route – just as in the establishment of complete sessions. Once an SRP module cannot determine a route beyond a node which does not implement SRP, no more reservations can be made, even if SRP modules exist further down the route. If SRP modules succeed in determining the complete route, this method is precarious because a node which does not implement SRP may choose to route messages past those nodes on which reservations have been made. To avoid this, IP messages have to be sent with the *loose source routing* option set. This option is used to specify a desired target address for the message (the address of the host with the next SRP module of the session), but allows multiple network hops in between (for those nodes which do not implement SRP). If the

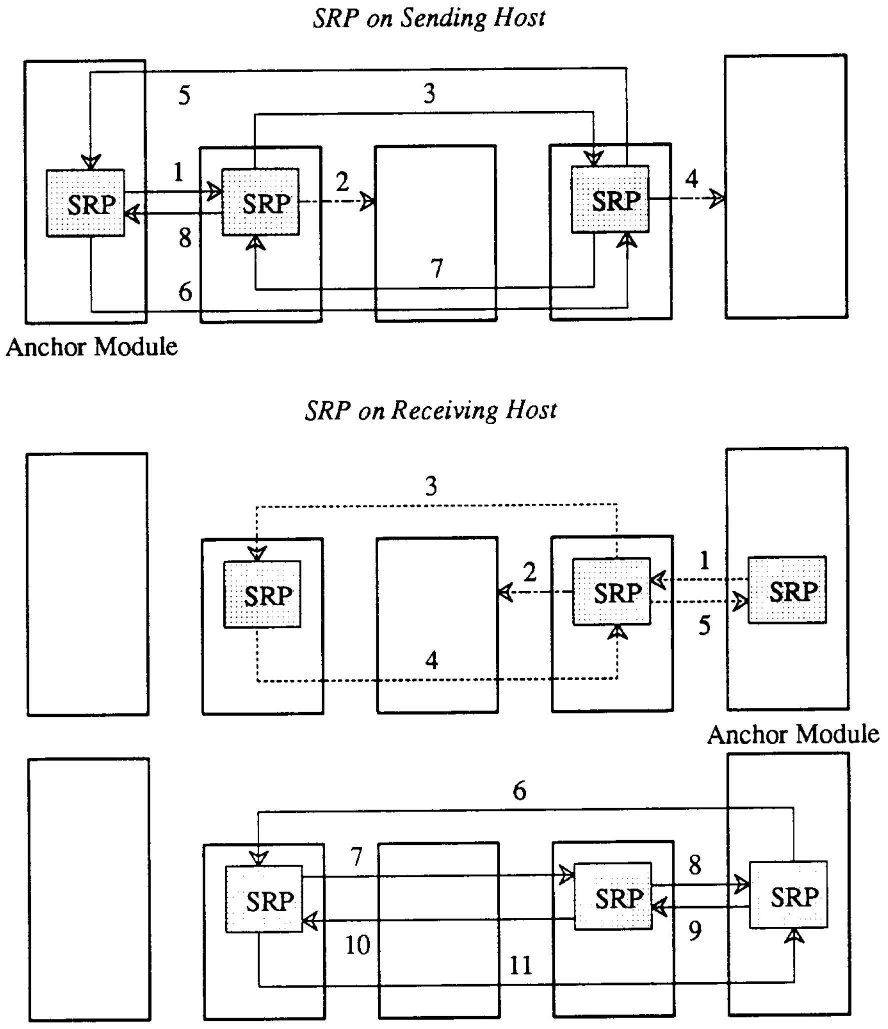

Figure 5: Two examples of partial session establishment.

sending host does not implement SRP and this method is chosen, the path from the sending client to the first SRP module has to be singular and invariant.

(4) If the anchor module is located on the receiving host and the route is unknown, an additional protocol phase becomes necessary to determine the session route. Under the assumption that a communication path is reversible, the SRP modules try to find a route from the receiving to the sending client. The anchor module issues an RPC to its adjacent node on a route towards the sender. If the node implements SRP it will forward the request to its own predecessor. If not, the anchor module has to find the next SRP node further down the route. Again, this process comes to an end once a module is unable to determine the route beyond a node which has no SRP module. On the way back each SRP module adds the address of its host to the output parameter of the RPC. The anchor

module then contacts the first SRP module in the route obtained, which then initiates the reservation procedure.

The establishment protocol for complete sessions requires the receiving client to be contacted after the first phase of the protocol. In partial sessions, however, the receiving client may have no access to SRP. The sending client will then play the role of the receiving client in deciding about the final delay parameters. If the last SRP module on the route is not located on the receiving host, it calls back the anchor module on the sending site where an indication is delivered to the sending client. To enable the client to accept the session establishment indication, a non-blocking session establishment primitive is needed. Such a function is also used by the receiving client if this client initiates the session establishment.

6.2. Protocol Elements of Extended SRP

The protocol elements to handle partial sessions form the *extended* version of the SRP protocol. If a node that implements SRP but not the partial session extension is asked to participate in a partial session, a `FAIL_SRP_VERS` failure code is returned.

```
program SRP {
   version Original {
      ...
   } = 1;

   version Extended {
      void null_function (void) = 0;
      E_rep_data session_establish (E_req_data) = 1;
      void session_delete (E_session) = 2;
      Address<> p_session_route (Address) = 3;
      E_rep_data p_session_establish (PE_req_data) = 4;
   } = 2;
} = ?;                           // to be assigned
```

Two additional RPCs exist in the extended protocol version. `p_session_route()` is provided to determine a session route. It is used to start the session establishment procedure from the receiving host. It takes as its single argument the address of the sending host and returns a route from the sending to the receiving host. The other RPC, `p_session_establish()`, is used to establish partial sessions. No special RPC is needed to delete partial sessions, though the client that established the session must be the one to delete it.

Only the parameters for a partial session establishment request are different from the original RPC. The return values are identical.

```
struct PE_req_data {
   E_session     esid;            // end-to-end session id
   Node          node<>;          // (partial) route
   Address       net;             // incoming network (hint)
   Connection    connection;      // associated connection
   Rate          rate;            // rate
   Burst         in_burst;        // input burst limit (hint)
   Size          size;            // message size
```

```
      Time          tgt_ete_delay;  // target end-to-end delay
      Time          max_ete_delay;  // maximum end-to-end delay
      Time          acc_max_delay;  // maximum logical delay so far
      Time          acc_min_delay;  // minimum actual delay so far
      Point         acc_cost<>;     // cost function so far
   };
```

In this request message, a route of nodes can be specified as a variable-length array. A single element of this array has the following format:

```
   enum Node_state
      { NOT_CONTACTED, SRP, NO_SRP };
   struct Node {
      Address       addr;           // address of this node
      Node_state    state;          // status of this node
   };
```

In the beginning, the route contains only the addresses of the sending and receiving hosts. For every node up to the current node, it has already been determined if the node implements SRP or not. All future nodes are labeled `NOT_CONTACTED`. Depending on its own routing information, each SRP module can extend or modify the future route.

6.3. Application Interface

Just as in previous sections we also specify an application interface for partial session establishment. Since only one client may be involved in the establishment process, two different session request functions, one blocking and one non-blocking, are provided. The blocking function is almost identical to the one used for complete session establishment.

```
   int SRP::establish_p_session (
      // INPUT
      E_session     esid,           // end-to-end session id
      Route*        route,          // (partial) route
      Address       net,            // used network (hint)
      Association*  assoc_data,     // data to identify connection
      Rate          rate,           // rate
      Burst         in_burst,       // input burst limit (hint)
      Size          size,           // message size
      Time          tgt_ete_delay,  // target end-to-end delay
      Time          max_ete_delay,  // maximum end-to-end delay
      Time          acc_max_delay,  // maximum logical delay so far
      Time          acc_min_delay,  // minimum actual delay so far
      Delaycost*    acc_cost,       // cost function so far
      // OUTPUT
      Time*         excess_delay,   // remaining excess delay
      Burst*        out_burst,      // accept. output burst limit (hint)
      Address*      culprit,        // host where failure occurred
      char*         msg             // name of failing resource
   );
```

The only difference to the original function is the specification of the route as a linked list.

```
   struct Route {
      Node          node;           // this node
      Route*        next;           // next node
   };
```

The non-blocking function is used if only one client machine has an SRP module. If a client uses this function, it will obtain the session establishment indication itself. Some parameters of the request (for example, an accumulated cost function) may not be available if the reservation starts on the receiving side. The outcome of the establishment request is indicated by a separate confirmation primitive which blocks the client until the result becomes available.

```
int SRP::establish_p_session_req (
    // INPUT
    E_session     esid,             // end-to-end session id
    Route*        route,            // (partial) route
    Address       net,              // used network (hint)
    Association* assoc_data,        // data to identify connection
    Rate          rate,             // rate
    Burst         in_burst,         // input burst limit (hint)
    Size          size,             // message size
    Time          tgt_ete_delay,    // target end-to-end delay
    Time          max_ete_delay,    // maximum end-to-end delay
    Time          acc_max_delay,    // maximum logical delay so far
    Time          acc_min_delay,    // minimum actual delay so far
    Delaycost*    acc_cost          // cost function so far
);

int SRP::establish_p_session_con (
    // INPUT
    E_session     esid,             // end-to-end session id
    // OUTPUT
    Time*         excess_delay,     // remaining excess delay
    Burst*        out_burst,        // accept. output burst limit (hint)
    Address*      culprit,          // host where failure occurred
    char*         msg               // name of failing resource
);
```

The establishment indication gives the route for which the partial session is available. The client can use this information in its decision about the suitability of the performance parameters. For example, it may reject a session that does not provide guarantees for nodes which are well-known performance bottlenecks.

```
int SRP::establish_p_session_ind (
    // INPUT
    E_session     esid,             // end-to-end session id
    // OUTPUT
    Route*        route,            // (partial) route
    Address*      net,              // used network (hint)
    Rate*         rate,             // rate
    Burst*        in_burst,         // input burst limit (hint)
    Size*         size,             // message size
    Time*         tgt_ete_delay,    // target end-to-end delay
    Time*         max_ete_delay,    // maximum end-to-end delay
    Time*         acc_max_delay,    // maximum logical delay so far
    Time*         acc_min_delay,    // minimum actual delay so far
    Delaycost*    acc_cost          // cost function so far
);
```

The acknowledgement and delete functions are the same as for complete sessions.

7. CONCLUSION

We have presented a scheme to achieve guaranteed-performance communication in the framework of an existing and widely used protocol architecture. The mechanisms described in this paper, though implemented for IP-based network communication, are by no means restricted to that framework. This choice, however, enables a large number of systems to participate in guaranteed-performance communication, requiring a minimum degree of modification to existing systems.

The most fundamental decision in the design of SRP was the use of the DASH resource model and the model of data traffic associated with it. A variety of other models, both statistical and deterministic, could have been used instead, probably offering the possibility to describe a broader range of traffic properties (for example, average message loss). The model we chose has the advantage of being simple and useful for deriving performance guarantees for a variety of scheduling disciplines. The primary application area of SRP, continuous media, is well supported by our model. Some potential drawbacks are:

- During the establishment of an end-to-end session other establishment requests may be rejected needlessly because maximum reservations are made for each resource. This problem can be avoided easily by implementing a lock for each HRM, blocking each new request until pending requests are completed.
- SRP only establishes and deletes end-to-end sessions. No means to relax the requirements on such sessions are provided. However, should the requirements on a connection change (rare for continuous-media applications), a new end-to-end session can be associated with it.
- The algorithms we have presented are conservative. Optimistic approaches (similar to those in transaction processing) may be suitable for the real-time application domain, especially in regard to bandwidth reservation.

Finally, the protocol in its present form (just like the established Internet protocols) deals with one-to-one communication only. Since continuous-media systems will be used to a large extent for groupware applications like conferencing, multi-point connections are an important issue. Our future work will address this problem, extending the model of end-to-end sessions to be compatible with IP multicasting [14].

Acknowledgements

Martin Andrews, Robert Wahbe, Shin-Yuan Tzou and Ramesh Govindan contributed to the development of the DASH resource model. Thanks also to Domenico Ferrari for helpful comments and discussions.

References

1. A. C. Luther, *Digital Video in the PC Environment*, McGraw-Hill, 1989.
2. S. Newman, ''The Communications Highway of the Future'', *IEEE Communications Magazine 26*, 10 (October 1988), 45-50.
3. K. A. Frenkel, ''The Next Generation of Interactive Technologies'', *Comm. of the ACM 32*, 7 (July 1989), 872-881.

4. J. Postel, ''Internet Protocol'', *DARPA Internet RFC 791*, Sep. 1981.

5. J. Postel, ''Transmission Control Protocol'', *DARPA Internet RFC 793*, Sep. 1981.

6. D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan and M. Andrews, ''Support for Continuous Media in the DASH System'', Technical Report No. UCB/CSD 89/537, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Oct. 1989.

7. R. L. Cruz, ''A Calculus for Network Delay and a Note on Topologies of Interconnection Networks'', Ph.D. Dissertation, Report no. UILU-ENG-87-2246, University of Illinois, July 1987.

8. D. Ferguson, Y. Yemini and C. Nikolaou, ''Microeconomic Algorithms for Load Balancing in Distributed Computer Systems'', *Proc. of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

9. S. Cheng, J. A. Stankovic and K. Ramamritham, ''Scheduling Algorithms for Hard Real-Time Systems'', in *Hard Real-Time Systems*, J. A. Stankovic and K. Ramamritham (editor), IEEE Computer Society, 1988, 150-173.

10. C. L. Liu and J. W. Layland, ''Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment'', *J. ACM 20*, 1 (1973), 47-61.

11. M. Andrews, ''Guaranteed Performance for Continuous Media in a General Purpose Distributed System'', Masters Thesis, UC Berkeley, Oct. 1989.

12. ''RPC: Remote Procedure Call Specification, Version 2'', Internet RFC 1057, Sun Microsystems, June 1988.

13. ''XDR: External Data Representation Standard'', Internet RFC 1014, Sun Microsystems, June 1987.

14. S. Deering, ''Host Extensions for IP Multicasting'', Internet RFC 1112, August 1989.