A New Proposal for RSVP Refreshes*

Lan Wang, Andreas Terzis, Lixia Zhang
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{lanw, terzis, lixia}@cs.ucla.edu

Abstract

As a soft-state protocol, RSVP specifies that each RSVP node sends periodic control messages to maintain the state for active RSVP sessions. The protocol overhead due to such periodic messages grows linearly with the number of RSVP sessions. One may reduce the overhead by using a longer refresh period, which unfortunately leads to longer delays in re-synchronizing RSVP state.

In this paper we introduce a novel "state-compression" approach to reducing the overhead of periodic refreshes. Instead of per session refresh messages, an RSVP node sends periodically to each of its neighbor node a Digest *message that contains a compressed version of the entire RSVP state shared with that particular neighbor. In order to speed up state synchronization in face of message losses we also enhance RSVP with an acknowledgment mechanism. Our mechanisms achieve a constant message transmission overhead and low delay while retaining the soft-state nature of the RSVP protocol.*

1. Introduction

RSVP [3] is a resource reservation protocol that can be used to request specific quality of service for particular data flows across the Internet. RSVP carries such requests to all the nodes along the data path(s) to make the resource reservation. As a *soft-state* protocol, RSVP sets a finite lifetime for all the reservation state. The end points of RSVP data flows maintain their reservation by sending periodic refresh messages along the data paths; a session's state is automatically deleted when its lifetime expires. Thus the network is free from obsolete or orphaned reservations.

The periodic Refresh messages play the following important roles in assuring correct protocol operation:

1. **Automatic adaptation to route changes**. IP routing changes cause data flows to switch to different paths. By design RSVP refresh messages follow the data paths, thus the first RSVP messages along the new paths will establish the requested reservations, while the state along the old paths is either explicitly torn down or otherwise automatically timed out.
2. **Persistent state synchronization**. RSVP messages are sent as IP datagrams which can be lost on the way. RSVP state at individual nodes may change due to rare or unexpected causes (e.g. undetected bit errors). These factors may lead to momentary inconsistency in RSVP state along the data paths. Periodic refreshes serve as a simple repairing mechanism that correct any and all state inconsistencies in RSVP state for active sessions.
3. **Built-in adaptation mechanism for reservation adjustment**. When either a sender or a receiver needs to change its traffic profile or reservation parameters during a session, it simply puts the modified parameter values in the next refresh message.

There is, however, a price to pay for the simplicity and robustness that come with soft state: the protocol overhead grows linearly with the number of active RSVP sessions. Even in the absence of new control information generated by sources or destinations, an RSVP node sends to its neighboring nodes one message per active sender-session pair per refresh period. Another performance concern is the reservation setup or tear-down delay caused by occasional losses of RSVP control messages. Although periodic RSVP refreshes eventually recover any previous losses, the recovery delay, which is proportional to the refresh period, can be considered unacceptable in a number of circumstances. One may reduce the recovery delay by reducing the refresh period, however doing so would worsen the refresh overhead problem.

To overcome the dilemma between protocol overhead and responsiveness, in this paper we present a new approach

*We would like to thank Intel and Cisco for their generous support of this research.

to RSVP overhead reduction. The crux of our scheme is to replace all the refresh messages sent between two neighboring nodes for each of the RSVP sessions with a *digest* message that contains a compressed "snap shot" of all the shared RSVP sessions between two neighbor nodes. When an RSVP node, say N, receives a digest from a neighbor node, it compares the value carried in the digest message with the value computed from N's local RSVP state. If the two values agree, N refreshes all the corresponding local state; otherwise N starts a state re-synchronization process to discover and repair the inconsistency. To assure quick state synchronization in face of packet losses we also enhance RSVP with an acknowledgment option, so that instead of waiting for next refresh, any lost RSVP messages can be quickly retransmitted.

The rest of this paper is structured as follows: we briefly introduce the basic RSVP terminology and operations below. In Section 2 we present an overview of our *state compression* algorithm. Section 3 describes in detail the data structure and procedure for computing RSVP digests. Section 4 presents a list of proposed new RSVP messages and related processing rules that are needed to implement our new scheme. Section 5 analyzes the overhead of digest computation. After identifying the limitations of our scheme in Section 6, we discuss the relationship between this effort and previous work done in the area in Section 7. Finally, Section 8 summarizes our findings and discusses the applicability of our method to other soft-state protocols.

1.1. RSVP Basic Operations and Terminology

RSVP is a receiver-driven protocol. To provide receiver-driven reservation functionality, a data source sends PATH messages towards the receivers, leaving behind a trace of "path state" at each router they traversed. Receivers wishing to make a reservation then send RESV messages, which follow the path state traces upstream towards the data source, reserving resources at each intermediate node along the way. The state set up by PATH and RESV messages is called *path* and *reservation* state, respectively. The state is deleted if no matching refresh messages arrive before the expiration of its life timer. The state may also be deleted by either the sender or receiver using PathTear or ResvTear messages.

PATH and RESV messages are idempotent. When a route changes, the next PATH message will initialize the path state on the new route, and future RESV messages will establish reservation state there; the state on the now-unused segment of the route will time out. Thus, whether a message is "new" or a "refresh" is determined separately at each node, depending upon the existence of state at that node.

Before we proceed with our proposed RSVP state compression algorithm we give the definitions of some terms that will be used in the rest of this paper.

RSVP State An RSVP path or reservation state.

Regular/Raw RSVP Message RSVP messages defined in RFC2205 [3], e.g. PATH, RESV, PathTear and ResvTear messages.

Refresh Message An RSVP message triggered by a refresh timeout to refresh one or a set of RSVP state. It can be a PATH message for a path state, a RESV message for a reservation state or a Digest message (in our scheme) for aggregate state.

MD5 Signature The result of the computation of the MD5 algorithm.

Digest A set of MD5 signatures that represents a compressed version of the RSVP state shared between two neighboring RSVP nodes.

2. Design Overview

The goal of our proposal is to improve RSVP's scalability allowing efficient operation with large number of sessions (e.g. tens of thousands sessions). More specifically, we aim at reducing the number of refresh messages while still preserving the soft-state paradigm of RSVP. In this section we briefly describe our *state compression* approach; the details of the compression scheme are presented in the next section.

Instead of sending a refresh message per sender-session pair to a neighbor, our approach is to let each RSVP node send a *digest* message which is a compact way of representing all the RSVP session state shared between two neighboring nodes. In this way, the number of refresh messages per refresh period is reduced from being proportional to the number of sessions to being proportional to the number of neighbor nodes. Raw RSVP messages are sent either when triggered by state changes or after state inconsistency is detected to re-synchronize the state shared between two nodes.

These benefits cannot come without any overhead. Generally speaking, the protocol overhead of RSVP can be divided into two components, the *bandwidth overhead* for message transmissions, and the *computation overhead* for processing these messages. One can further subdivide the computation overhead to system overhead (e.g. system interrupts by packet arrivals) and message processing overhead. The state compression scheme can effectively decrease the bandwidth and system overhead, however at the cost of increased message processing overhead as we apply additional processing to compress RSVP state to a single digest message per neighbor. Therefore, one important part of our design is to minimize the cost of digest computation.

To compress RSVP state into a digest, one can simply concatenate the state of all the RSVP sessions into a long byte stream and compute a digest over it. However this brute-force approach suffers from a high overhead of recomputing the whole digest again whenever any change happens. To scale the digest computation we compute the digest in a structured way. First, we hash all the RSVP sessions into a table of fixed size. We then compute the signature of each session, and for each slot in the hash table we further compute the slot signature from the signatures of all the sessions hashed to that slot. On top of this set of signatures, we build an N-ary tree to compute the final digest (a complete description of the data structures used is given in section 3.3).

There are two advantages in using a tree structure to compute the digest:

1. Whenever the digests computed at two neighboring nodes differ, the two nodes can efficiently locate the portion of inconsistent state by walking down the digest tree;
2. When an RSVP session state is added/deleted/modified, an RSVP node only needs to update the signatures along one specific branch of the digest tree, i.e. the branch with the leaf node where the changed session resides.

In our current design, we use the MD5 algorithm to compute state signatures. As stated in [7], "it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest." We can therefore conclude, with a high level of assurance, that no two sets of different RSVP states will result in the same signature. However, it should be noted that our state compression scheme can work well with any hash function that has a low collision probability, such as CRC-32, as long as two neighboring nodes agree upon their choice of the hash function.

As a further optimization, we also add an acknowledgment option (ACK) to the RSVP protocol. The ACK is used to minimize the re-synchronization delay after an explicit state change request. A node can request an ACK for each RSVP message that carries state-change information, and promptly retransmit the message until an acknowledgment is received. It is important to note the difference between a soft-state protocol with ACKs and a hard-state protocol. A hard-state protocol relies solely on reliable message transmission to *assume* synchronized state between entities. A soft-state protocol, on the other hand, uses ACKs simply to assure quick delivery of messages; it relies on periodic refreshes to correct any potential state inconsistency that may occur even when messages are reliably delivered, for example state inconsistency due to undetected bit errors, or due to undetected state changes.

3. State Organization

One can suspect that the increase in refresh efficiency cannot come for free. This is indeed the truth and the tradeoff comes in the form of increased storage and computation. The increase in storage originates from the need to keep *per neighbor state*, since separate digests are sent to different neighbors. Consequently, computation costs are inflated since we have to compute the per-neighbor digests and we have to operate on the per-neighbor data structures. In the sections that follow we elaborate on the requirements for extra state introduced by the compression mechanism. Computation costs are further analyzed in Section 5.

3.1. Neighbor Data Structure

Current RSVP implementations structure the RSVP state inside a node as a common pool of sessions, regardless of their destinations. On the other hand, digest messages sent towards a particular neighbor contain a compressed version of the RSVP state shared *with that neighbor*. The need therefore arises to further organize RSVP state inside a node according to the neighbor(s) each session comes from or goes to. To satisfy this need we introduce the *Neighbor* data structure which holds all the information needed to calculate, send and receive digests to and from a specific node.

In essence the Neighbor data structure is the collection of RSVP sessions that the current node sends to or receives from a particular neighbor. For efficiency neighbor data structures may not actually store the sessions but contain pointers to the common pool of sessions. This way a session shared with multiple neighbors is not copied multiple times to the corresponding neighbor structures. In addition to sessions, the neighbor structure contains the digest computed from the sessions shared with the neighbor and some auxiliary information such as retransmission and cleanup timers.

A node needs to compute two digests for each neighbor, one for the state refreshed by messages received from that neighbor and one for the state the local node is responsible for refreshing towards that neighbor. We call these two digests InDigest and OutDigest respectively. OutDigest is sent in lieu of raw refreshes while InDigest is used to for comparison when receiving a Digest message from that neighbor. In the next section we present how we compress each session state into an MD5 signature. In section 3.3 we delve into the details of the data structure and algorithm we use to derive a digest from the session signatures.

3.2. Session Signature

To compress a session state into a signature, we first need to identify which session parameters need to be constantly synchronized between neighbors. Table 1 shows the RSVP objects included in the digest computation. A session is uniquely identified by a session object which contains the IP destination address, protocol ID and optionally a destination port number of the associated data packets. A Path State Block (PSB) is comprised of a sender template (i.e. IP address and port number of the sender), and a Tspec that describes the sender's traffic characteristics and possibly objects for policy control and advertisements. A Reservation State Block (RSB) contains filterspecs (i.e. sender templates) of the senders for which the reservation is intended, the reservation style and a flowspec that quantifies the reservation. It may also contain objects for policy control and confirmation. Although PSBs and RSBs contain some other fields such as incoming interface and outgoing interfaces, these fields have only local meaning to a specific node and therefore should be excluded from the digest computation. As to RSVP objects defined in the future, the digest computation can also be applied to them if necessary.

RSVP Objects	Sub-objects to Include
Session	session object
PSB	sender template, sender tspec, adspec, policy
RSB	filterspec, flowspec, reservation style, policy

Table 1. RSVP Objects Included in Digest Computation

We noticed that, in the current RSVP specification, RSBs record only reservations requests received from downstream neighbors, but not reservation requests forwarded upstream. However, for a multicast session or many-to-one unicast session, the reservation request a node receives from a downstream neighbor may not be the same as the one it sends to an upstream neighbor if the node is a merging or splitting point. Since the sender of a digest has to compute the digest based on what flowspec and filterspec are sent to its neighbor, we require such information to be kept in associated RSBs to facilitate the digest computation.

3.3. Hash Table and Digest Tree

The existence of the structures described in this section is not fundamental for the correct operation of our compression scheme. However given the context where our proposed solution will be most useful (e.g. tens of thousands of sessions), these structures provide the desired performance to make the scheme practically viable. Two are the principal reasons that impelled us to include these data structures:

- Given the need for expeditious response to state changes and the high volatility resulting from the high volume of sessions, updates, insertions and deletions must be done efficiently. This requirement can be translated to two subgoals: a data structure that supports efficient session insertions and deletions and second, *incremental* digest computation. Unfortunately, the design of the MD5 algorithm does not allow incremental digest computation. To overcome this limitation we compute the state digest *recursively*, by applying the algorithm to session sets of increasing size.
- State inconsistencies must be resolved rapidly without requiring complete state retransmission. To do so, we need to quickly locate which part of RSVP state contains the inconsistency and then send raw refreshes only for these sessions.

In addition to the two primary reasons, simplicity and robustness are essential if this mechanism is to supersede the minimalism and potency of raw refreshes. With this set of goals in mind, we continue by presenting each one of the two data structures next.

Sessions are stored inside a hash table. The size of the hash table is M and sessions are *hashed* to one of the M hash table slots. Hashing is done over some fixed session fields (e.g the session's address). If multiple sessions hash to the same slot, they are inserted in a linked list. Sessions inside the linked list are stored ordered according to their destination address. Figure 1 shows the session hash table. Slot i contains a pointer to the head of the linked list of all stored sessions that hash to i. It also contains an MD5 signature that is computed by concatenating all the sessions' MD5 signatures and applying the MD5 algorithm on the compound message.

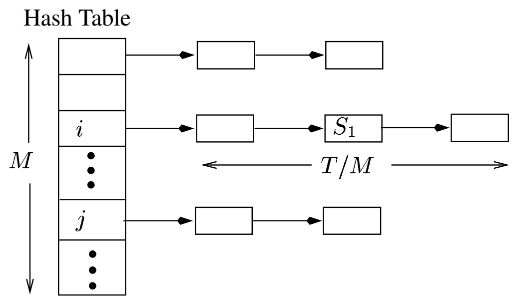

Figure 1. Hash Table

The second step is to reduce the total number of signatures from M to N, the number of signatures that can fit inside a single message. To do that we have introduced a complete N-ary tree whose leaves are the slots of the hash table. This *digest tree* is shown in Figure 2.

A node constructs the digest tree in the following way. As we said earlier, the leaves of the tree are the signatures stored in the slots of the hash table. The signatures

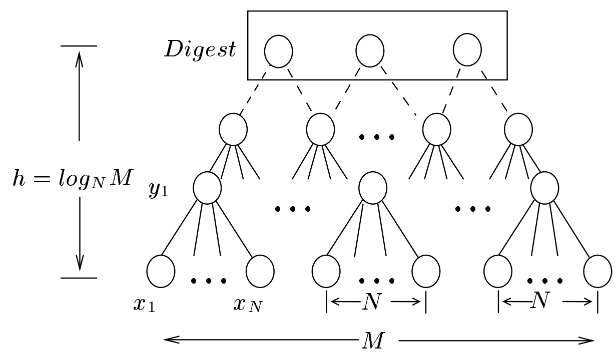

Figure 2. Digest Tree

of N slots are concatenated and the MD5 algorithm is applied on the compound message. The result is stored at the parent node on the tree. Looking at Figure 2, signatures $x_1, \ldots, x_N$ are concatenated and the MD5 algorithm result is stored in node y_1. This grouping results in M/N level-1 signatures. If the number of level-1 signatures is still larger than N, the node continues on to group each of N level-1 signatures to compute a level-2 signature to get M/N^2 level-2 signatures. If C_i is the number of level-i signatures, we repeat the grouping until C_i is less than or to equal to N. The top level signatures represent the *digest* of that RSVP state.

We have chosen the degree of the tree to be the same as the maximum number of MD5 signatures inside the digest object to simplify the data structure and to reduce the number of parameters. Note that all insertions and deletions are done in the hash table while the purpose of the digest tree is to reduce the number of signatures from M to N and to store intermediate results that will be used during the recovery phase, after inconsistencies are detected.

The hash table size M and the maximum number of signatures N in a digest are two important factors that affect the performance of our digest scheme. A smaller M results in more sessions being hashed to each slot on average, which means more overhead in updating the signature of a slot. However, given an N, a smaller M also leads to a lower digest tree and consequently fewer intermediate levels of the tree to maintain and fewer messages to exchange during the recovery procedure. This suggests that, when choosing M, one needs to take into consideration both the target tree height and the expected number of sessions in each hash slot. Furthermore, if the actual number of sessions differs greatly from the expected number of sessions, the sender of a digest may need to change its M to achieve better performance. As a result, the sender should nofify the receiver of the modified M and the receiver needs to use the new value of M in its digest computation. A larger N means each node in the tree has a larger fan-out and therefore the digest tree will have fewer levels. In general, one would like N to be the largest value allowed by the link MTU.

4. Mechanism Description

4.1. New RSVP Messages and Objects

Our compression mechanism requires three new RSVP messages, namely: *Digest*, *ACK* and *DigestErr*. A Digest message carries a timestamp object that uniquely identifies it and a digest object that represents the state shared between a node and its neighbor (i.e. the receiver of the message). After a node discovers a neighbor capable of exchanging digests (see section 4.2), it periodically sends Digest messages refreshing the total RSVP state of that neighbor. If a node wishes to send Digest messages at a different interval than the standard, it can specify that interval in the Digest message. In this way, the receiver will know when to expect Digest messages and in their absence when to delete the associated state.

ACK messages are used to acknowledge raw RSVP messages or Digest messages. Since many messages may be outstanding when an ACK is received at the sender side, the ACK message contains the timestamp of the message it acknowledges. The receipt of an ACK message indicates that the original message was received and processed by the receiver. Moreover, the message was processed at the receiver side without creating any errors. Otherwise, an error message (ResvErr, PathErr or DigestErr) would have arrived instead of the ACK message.

A DigestErr message acts as a negative acknowledgment to a Digest message. Similar to ACK messages, the DigestErr message carries the timestamp of the received digest message. In addition, it contains the digest value computed at the receiver side, which is used later in the recovery process (see section 4.4).

The timestamp object mentioned before contains two basic fields: the *Timestamp* field which is the time that the packet was sent and the *Epoch* field which is a random 32-bit value initialized at boot time. All timestamp objects sent from a node should use the same Epoch value as long as the node is not rebooted. If, after the initialization phase, a node receives two consecutive messages with different Epoch values, it can conclude that the sender of these messages has rebooted. The receiving node must then purge all state associated with that sender.

We chose to use time as the message identifier because it is always increasing and so a sender does not have to check if the value is in use or has been used before. It also helps the receiver to identify which of the RSVP messages for the same state is the most recent one. However, depending on the node's processing speed and timer granularity, two

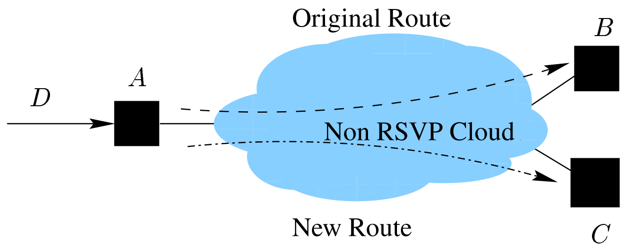

Figure 3. RSVP Session over a non-RSVP cloud

consecutive messages may get the same timestamp value. Therefore, we define the timestamp to be $max(t, t_{last} + 1)$, where t is the current time and t_{last} is the last timestamp value used.

Furthermore, the timestamp object carries a flag indicating whether the sender is requesting an acknowledgment for this message. This flag should be turned off in the timestamp objects carried by ACK and DigestErr messages to avoid an infinite exchange of ACK messages.

Last, the digest object carries a set of MD5 signatures. These signatures can be either the digest or some set of MD-5 signatures from some other level of the digest tree.

4.2. Neighbor Discovery

To use the compressed refresh scheme, a node needs to discover which of its neighbors are capable of exchanging digests. For this reason, when a RSVP node starts sending (raw) RSVP messages for a session, it should request that the neighbor(s) acknowledge these messages by including a timestamp object with the ACK_Requested flag turned on. If the node receives an ACK message in response from a neighbor whose address is not currently on the Neighbor Structure list, it has then discovered a new compression-capable neighbor. If on the other hand, that neighbor does not understand timestamp objects (legacy node), it will return an error message. We can then conclude that this neighbor is compression-incapable.

When a non-RSVP cloud exists between two RSVP neighbor nodes, although the nodes can discover each other using acknowledgments during the initial message exchanges, the upstream neighbor may not be able to detect whether sessions crossing the cloud switch next hops. These changes are caused by route changes inside the non-RSVP cloud and are not detectable if the upstream neighbor's outgoing interface remains the same. The original RSVP specification does not share this problem since RSVP messages for individual sessions carry the session's address and therefore naturally follow any route changes. In the compression scheme however, digest messages are explicitly addressed to particular next hops and therefore the same solution cannot be used.

Figure 3 illustrates our point. In this scenario node A originally has B as its downstream neighbor for session D. After a route change, node C becomes A's downstream neighbor for that session. However, since A's outgoing interface remains unchanged, A will not notice the route change, hence it will continue to include session D when calculating the digest to B. Node B will not be informed of the change either as long as A sends it the same digest. Therefore, node C will never get a PATH message from A. As a result, resources will be reserved on the path between A and B while data packets will follow the path from A to C.

The digest scheme therefore, cannot be used over non-RSVP clouds until an effective way of detecting route changes is found. Fortunately, the existence of non-RSVP clouds can be detected by mechanisms described in [2]. If a non-RSVP cloud exists between two nodes, regular refreshes should be used instead of the compression mechanism.

4.3. Normal Operation

Neighboring nodes start by exchanging regular RSVP messages as usual. Once a node discovers a compression-capable neighbor, it creates a digest for the part of its RSVP state that it shares with each of this neighbors. Subsequently, the node sends Digest messages instead of raw RSVP refreshes at regular refresh intervals. When an event that changes the RSVP state (e.g. a sender changes its traffic characteristics (Tspec)) occurs, raw RSVP messages are sent immediately to propagate this change.

Raw RSVP messages are sent as before, with the added option of asking for an ACK. A sender requesting an acknowledgment, includes in the message a timestamp object with the ACK_Requested flag turned on. The sender also sets a retransmission timer for the packet sent. Processing at the receiver side includes updating the digest of the session that the message belongs to as well as updating the digest tree. If during processing a condition occurs that requires sending back an error message back to the sender (e.g a ResvErr) then the receiver sends back to the sender that error message. This error message will cancel any pending retransmissions of the original message.

If no ACK is received before the retransmission timer expires, the sender retransmits the message up to a configured number of times. Each of the retransmissions carries the same timestamp contained in the original message. If an updated message (i.e. a PATH message from the same sender but with different Tspec) is sent before an ACK is received, the original message becomes obsolete and no longer needs to be retransmitted. If no ACK arrives even after the message has been retransmitted for the maximum number of times, the message is purged from the node's list of pending messages. Any inconsistencies created by the possible loss of this message will be later resolved by digests.

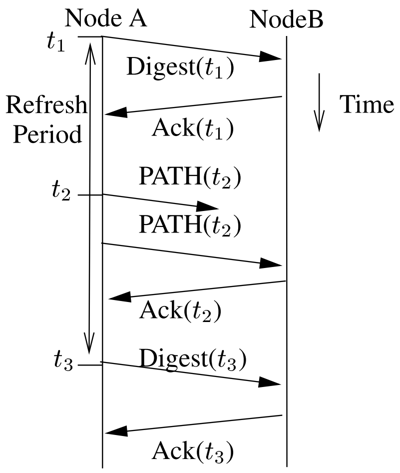

Figure 4. Message Exchange

Digest messages are always sent with the ACK_Requested flag turned on. Digest messages are also retransmitted for a maximum number of times in the absence of ACK messages. However, following the original RSVP design where an RSVP node never stops sending refresh messages for each active session, a node should not stop sending digest refreshes even if it fails to receive an acknowledgment in the previous refresh interval. If the neighbor node crashed and becomes alive again, it will find the digest value different from its own and the two routers will start the re-synchronization process. When the digest value is changed, the node needs to cancel any pending retransmission of the obsolete Digest message and promptly send a Digest message with the new digest value.

When a node receives a Digest message, it checks to see if the state reported by the Digest message is consistent with the corresponding state stored locally. To do so the node does a binary comparison between each of the MD5 signatures contained in the digest object and the corresponding MD5 signatures in the InDigest (see section 4.2). If all of them agree then the state is consistent and an ACK is sent back. Otherwise the receiver returns a DigestErr message containing its InDigest and the process described in the next section begins.

Figure 4 gives an example of message exchanges between two nodes under normal condition. Nodes A and B had consistent state at time t_1. A sent a Digest message to B and received an ACK message for it. A then had a state change at time t_2 which triggered a PATH message sent to B. This message was lost, so A didn't receive an ACK until it timed out and it had to retransmit the PATH message (using the same timestamp t_2). B received the retransmitted PATH message and sent an ACK message back to A. Up to this point, the two nodes were synchronized. When the digest refresh timer timed out at t_3, A sent a Digest message with updated digest value to B. Since A and B were still consistent, B sent an ACK to A for the Digest message.

4.4. Recovery Operation

Two RSVP neighbors may become out-of-sync due to a number of reasons. For instance, a state-changing RSVP message got lost and the sender did not ask for ACK. It may also happen that a node crashed and lost part or all of its state. Since it is impossible to enumerate all the possible reasons, the best that one can do is to detect state inconsistencies once they arise and have a way of repairing the damaged state.

As we mentioned in section 4.3, a node sends a DigestErr message if the received digest value disagrees with the local digest. The timestamp and digest value in the DigestErr message help the two neighbors localize the problem. If the timestamp acknowledged is smaller than the timestamp of the last Digest message sent, this error message is for an obsolete message. This message should be ignored since it may not represent the current state of the neighbor. If they are equal, the node starts a depth-first search of the mismatching signatures from the root of the digest tree.

When a node receives a DigestErr message it compares the digest value with its own to find the states that are inconsistent. When it finds the first mismatching signature (call it S_1), it sends a Digest message containing the signatures used to compute S_1. A DigestErr is expected for this Digest message since at least one of the children signatures should not match. The node again looks for the first mismatching signature (S_2) in the second DigestErr message and sends the children of S_2 in a Digest message. This procedure is repeated until the leaf signature (S_h)[1] causing the problem is found. Now, the node knows that one or more of the sessions in that hash table slot (represented by S_h) must be inconsistent with those in the neighbor. It can then locate these sessions by further exchanging the session signatures with the receiver. However, we found that locating specific sessions may get quite complicated in some cases, for example, when the sender or receiver has sessions that do not exist in the peer. When a node encounters these cases, it can simply send raw refreshes for all the sessions in that particular bin. After refreshing these sessions, the node reexamines S_{h-1} (the parent of S_h) for other inconsistencies and continues to traverse the tree until all the mismatching sessions are located and refreshed.

[1] $h = \lceil log_N M \rceil$, see Figure 2

Notice there is a tradeoff between the latency of the recovery procedure and the transmission efficiency. For example, if the tree has many levels, many RTTs are needed to exchange the digests at all the tree levels in order to find the leaf-level sessions that contribute to the inconsistency. However, if speed of convergence is more important than efficiency, one can stop at an intermediate tree level and refresh all the states represented by the mismatching signature at that level.

4.5. Time Parameters

There are two time parameters associated with digest messages: the refresh period between successive digest refreshes R and the retransmission timeout T. A node sends digests at intervals of r, where r is randomly chosen from the range $[0.5 * R, 1.5 * R]$. Randomization is used to avoid the synchronization of digest messages. If an acknowledgment is not received after time T from the transmission of a digest, the node will retransmit that digest message.

The current RSVP specification [3] states that the default refresh period for regular RSVP messages is 30 seconds but the interval "should be configurable per interface". To be consistent, digest refreshes are also sent every 30 seconds by default and this interval should be configurable. As we mentioned in Section 4.3, digest messages are explicitly acknowledged and therefore there is no need to decrease R to protect against lost digest messages. However, R affects the frequency of consistency checking between neighbors, so smaller values of R should be used in environments where prolonged periods of inconsistency are undesirable. The retransmission timeout T should be proportional to the round-trip time between two directly connected neighbors. A node can measure the time interval between a message and the corresponding ACK and estimate the mean RTT by performing exponential averaging on the measurements.

Another important time parameter is the state lifetime L. If state represented by a digest is not refreshed for a period L, it is considered stale and is deleted. The naive approach would be to set L to be equal to the refresh period R. This would however lead to premature state timeouts at the receiving side. There are at least two reasons for this: first, clocks at neighboring nodes may drift and second as we said before the refresh timer is randomly set to a value in the range $[0.5 * R, 1.5 * R]$, which means that the sender may send digests at intervals larger than R. These examples illustrate that L should be larger than R. Following the current RSVP specification we decided to set $L = (K + 0.5) * 1.5 * R$, where $K = 3$.

4.6. Backward Compatibility

The extensions we have introduced are fully compatible with the existing version of RSVP. If an RSVP node sends a message with a timestamp object and subsequently receives an "Unknown Object Class" error, it should stop sending any more messages with attached timestamp object and start using regular refreshes instead of digest refreshes. Digest messages do not pose a compatibility problem since a node will start sending Digest messages only when it discovers that its particular peer is compression-capable using the procedure outlined in section 4.2.

5. Computation Costs

In this section we focus on the operations applied to the data structures described in Section 3 and analyze their requirements in terms of processing.

We begin with some definitions. Let the number of sessions be T, the size of the hash table be M and the maximum number of MD5 signatures inside the digest message be N. Let's further define the cost of computing the MD5 signature of a message of size x to be $f(x)$. To determine the behavior of $f(x)$, we have to study the algorithm's behavior. Summarizing the description in RFC 1321 [7], the algorithm divides the input message to 64-byte blocks and applies a sixty-four step process to each one of these 64-byte blocks. In each of the sixty-four steps, a number of bit-wise logical operations are applied to that 64-byte block. The results of the computation on the nth block are used as input for the computation of the $(n + 1)$th block. After all the blocks have been processed, the message's signature is produced. From this description, one can see that $f(x)$ is a linear function of x, the size of the input message measured in bytes.

When a session is modified, a new signature for that session as well as a new digest of the whole RSVP state has to be computed. To illustrate this procedure, imagine that we want to update session S_1 inside the hash table of Figure 1. First, we look up the session inside the hash table. In our example, we would come up with the index i. If multiple sessions map to the same hash table slot, we traverse the linked list of sessions until we find the session in question. Once the session is found and its new MD5 signature is computed, we have to compute the new MD5 signature stored at the base of the linked list which represents all the sessions mapped to that hash table slot. On the average $\lceil T/M \rceil$ sessions will occupy the same slot. The total time needed for this operation is therefore $f(16 * \lceil T/M \rceil)$, since each MD5 signature is 16 bytes long. The next step is to update the values on the digest tree. We begin by computing the MD5 signature of the contents of slot i concatenated with its $N - 1$ *siblings* which will be stored in their *parent* node on the digest tree. We continue this procedure until we reach the top of the tree. Since there are $\lceil log_N(M) \rceil$ levels on the tree and at each level we apply the MD5 algorithm on a message of size $16 * N$ (the combined size of N MD5 signatures), the time spent during this step is $(\lceil log_N(M) \rceil - 1) * f(N * 16)$. Notice that the term is $\lceil log_N(M) \rceil - 1$ since we do not calculate an MD5 signatures out of the N topmost signatures.

From the discussion above, we can conclude that the total time needed to calculate the new digest after a session is modified is given by the following formula, where S is the size of a session in bytes:

$$f(S) + f(16 * \lceil T/M \rceil) + (\lceil log_N(M) \rceil - 1) * f(N * 16) \quad (1)$$

When a new session has to be inserted in the hash table, we locate the slot this session hashes to and insert the session to that slot's linked list, if one exists. Given that the list is ordered, the new session has to be inserted in order inside the list, which means traversing the list until we find a session whose destination address is larger than the destination address of the session we want to add and inserting the new session before that session. Deleting a session, involves finding the slot it hashes to, searching for it inside the linked list, and "splicing" its predecessor to its successor on the list.

The computation cost for the creation of the new digest after an insertion or deletion operation, is almost identical to the update cost. The only difference is that in the case of deletion we don't calculate the MD5 signature of the session (since we are deleting it). Equations 2 and 3 respectively, show the insertion and deletion costs.

$$f(S) + f(16 * \lceil T/M \rceil) + (\lceil log_N(M) \rceil - 1) * f(N * 16) \quad (2)$$

$$f(16 * \lceil T/M \rceil) + (\lceil log_N(M) \rceil - 1) * f(N * 16) \quad (3)$$

We can see from Equations 1, 2 and 3 that when the size M of the hash table is small compared to the number of sessions T, the cost of updating the linked list of sessions will be linear to T. In this case, updating the linked list becomes the most expensive operation, forcing the total cost to also be linear to T. The size M of the hash table should therefore be comparable to T to avoid increased update complexities.

6. Limitations of our Approach

The ability of two RSVP neighbors to exchange digests in place of raw RSVP messages relies on the assumption that the two nodes know precisely all the RSVP sessions that go through these two nodes in sequence. Whenever a route change occurs, the upstream node must be able to receive a notification from the RSVP/Routing interface and synchronize the state with the new downstream node (as well as tear down the session with the old downstream neighbor). For multicast sessions, another complication arises if a router is attached to a broadcast LAN. A router must detect all changes of membership in the downstream neighbors, for example when a downstream router on the broadcast LAN joins or leaves a group, which does not affect the list of outgoing interfaces of the associated RSVP state. Again the proposed scheme relies on the RSVP/Routing interface to provide notification of such changes.

Furthermore, we have identified two cases where an RSVP node must resort to the current refresh scheme. The first case is when both compression-capable and compression-incapable downstream neighbors exist on the LAN. To accommodate the compression-incapable neighbors one must use per session RSVP refresh messages. The second case is when two RSVP nodes are interconnected through a non-RSVP cloud as we explained in Section 4.2.

In summary, a seemingly inevitable limitation of the state compression approach is the loss of RSVP's automatic adaptation to routing changes. Because refresh messages for each RSVP session follow the same path as data flow, RSVP reservation can automatically adapt to routing changes including multicast group membership changes. When a node compresses the RSVP sessions currently shared with a neighbor node to a single digest, however, RSVP loses the ability to trace down the paths of individual flows.

7. Related Work

There have been several recent efforts that address the issues of protocol overhead due to periodic transmission of refresh messages. The scope of the proposed solutions vary widely. At one end of design spectrum, proposals such as [1] wish to convert RSVP's soft state design to a hard-state protocol with *keep-alive* probes. In [1], instead of refreshing reservation state RSVP neighbor nodes periodically exchange hello messages, assuming that the state between neighbor nodes will remain consistent as long as (1) all the messages are reliably delivered once, and (2) no link or node failure is detected. While this proposal avoids the state refresh overhead problem, it fails to preserve the full semantics of RSVP's soft state design with its associated benefits.

Other proposals, such as the work on "Scalable Timers" [8], try to limit the total bandwidth consumed by control traffic. To do so, the length of the refresh period increases proportionally with the amount of state that has to be refreshed. In that respect these approaches are similar to the solution of increasing the refresh period of RSVP messages. As we have argued, this approach unfortunately trades off the protocol latency problem.

The work closest in spirit to our proposal is [5], where the authors proposed a scalable naming scheme for SRM. In SRM each sender creates a *namespace* that describes the data it has already multicast to the session. To assure reliable data reception, each sender's namespace is periodically transmitted in a soft state fashion. Receivers can then detect whether they have lost any data, and request retransmissions whenever necessary. Members that joined a multicast session late can also learn how much data has been previously transmitted, and request retransmissions when needed. To scale the protocol, instead of sending the full namespace description, in [5] each sender computes an MD5 signature of its namespace and periodically multicast the signature to the session.

At first sight the work in [5] seems similar to our RSVP state compression scheme. There exist, however, fundamental differences between the two. While in SRM the goal is to assure ultimate delivery of the same set of long-lived data to all the receivers in the multicast session, in our case the goal is to keep each session's RSVP state consistent along *individual* data flows' paths; the RSVP state one node shares with different neighbors is different. In addition, the RSVP state at each node is the result of processing RSVP messages, one must take extreme care of the signature computation for each of the neighbors. Furthermore, RSVP state can be highly dynamic especially in a large network environment, with sessions coming and going all the time. These difference lead to our digest data structure design that supports efficient insertions and deletions, as well as the need to store per neighbor state.

[6] is a recent work that studies the data consistency and performance tradeoff under different loss rate with soft-state based data communication. Although this study is carried out in a different context than ours (data delivery rather than state synchronization), its result confirms that using feedback (acknowledgment) can greatly speed up the consistency between data senders and receivers with little cost of bandwidth usage.

8. Summary

To improve the RSVP performance, in this paper we presented two changes to RSVP. We let each node compress aggregate RSVP state to a digest that can be carried in a single packet, which is then exchanged between neighbor RSVP nodes. The digest computation is done in a structured way, so that state inconsistency between two neighbors can be quickly located and repaired. This state compression approach considerably reduces the message overhead of RSVP. We also enhance RSVP with an acknowledgment option, so that lost messages can be quickly retransmitted. Our work suggests that the acknowledgment mechanism should be considered as a complement to, rather than a conflict with, soft-state protocol designs. Although reliable delivery of control messages cannot be used to replace soft-state refreshes, use of acknowledgment speeds up state synchronization in case of message losses.

As a next step we plan to explore the feasibility of applying our approach to other soft-state protocols. For example, PIM-SM (Sparse Mode) is also a soft-state protocol. A PIM-SM node sends periodic messages upstream to refresh each of all the existing multicast trees. One could envision a state compression mechanism, similar to that proposed here for RSVP, be deployed to reduce the PIM refresh overhead. As another candidate to consider, the current BGP design uses TCP as the transport mechanism to achieve efficient operation, so that any routing information is transmitted once only; in the absence of link or node failures the routing table state is never refreshed. However operational experience has shown that neighboring BGP state may occasionally become inconsistent due to unexpected causes, consequently new mechanism has been proposed to add route refresh capability to BGP [4].

We would like to explore the possibility of achieving the same protocol efficiency by using a soft-state approach with state compression instead of a hard-state protocol such as TCP.

9. Acknowledgments

The phrase "overhead reduction by state compression" was first suggested by Van Jacobson. Vern Paxson suggested the use of hashing to simplify Digest computation. Our design also benefited from discussion with Steve McCanne regarding the roles of acknowledgment in soft-state protocol design. We also thank the IETF RSVP working group for valuable feedback on our initial design.

References

[1] L. Berger, D.-H. Gan, and G. Swallow. RSVP Refresh Reduction Extensions. *Internet-Draft, work in progress*, April 1999.

[2] R. Braden and L. Zhang. Resource ReSerVation Protocol (RSVP), Version 1 Message Processing Rules. *RFC 2209*, Sept. 1997.

[3] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP), Version 1 Functional Specification. *RFC 2205*, Sept. 1997.

[4] E. Chen. Route Refresh Capability for BGP-4. *Internet-Draft, work in progress*, August 1999.

[5] S. Raman and S. McCanne. Scalable Data Naming for Application Level Framing in Reliable Multicast. In *Proceedings of ACM Multimedia 98*, Sep. 1998.

[6] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proceedings of SIGCOMM'99*, 1999.

[7] R. Rivest. The MD5 Message-Digest Algorithm. *RFC 1321*, April 1992.

[8] P. Sharma, D. Estrin, S. Floyd, and V. Jacobson. Scalable Timers for Soft State Protocols. In *Proceedings of the IEEE INFOCOM 1997*, 1997.