RTP extension for Scalable Reliable Multic和平

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Distribution of this document is unlimited.

Abstract

This document describes how the Real-time Transport Protocol, RTP (RFC1189), could be extended to include support for parts of the framework called Scalable Reliable Multicast. The scheme proposed could be used for transporting a data flow reliably over the transport protocols supported by RTP in a light-weight way. This could be used for numerous applications, for instance white-boards, semi-reliable audio/video and messaging/data-transfers within group-ware applications.
1. Introduction

The Real-time Transport Protocol, (RTP) [1] is based on UDP which is a best-effort transport protocol meaning that packets can be lost. For some applications this is acceptable but for other, for instance white-board-applications, it is necessary to do retransmission so an end-user can rely on that he has received everything that everybody else has sent.

This document describe how RTP could be extended to include support for the framework of the so called Scalable Reliable Multicast [2]. The scheme proposed could be used for transporting a data flow reliably over the transport protocols supported by RTP.

This new RTP/SRM-platform could be used for numerous applications, including for instance semi-reliable audio/video and distributed group-ware applications.

2. The Scalable Reliable Multicast Framework

Scalable Reliable Multicast,(SRM) [2] is a reliable multicast framework for application level framing and light-weight sessions. The algorithms of this framework are efficient, robust and scale well to both very large networks and very large sessions. The framework has been prototyped in wb, a distributed white-board application, and has been extensively tested on a global scale with sessions ranging from a few to more than 1000 participants.

2.1 The SRM ideas

The ideas of SRM can briefly be described as follows:

1) Every packet transmitted is uniquely identified by a sender-identification and a sequence-number that is incremented by one for each transmitted packet.

2) Every member of the session buffers a certain amount of packets, even if she is only a receiver and not a sender, so if necessary she can participate in ‘repairs’ of lost packets.

3) When a receiver notices that she has lost packets (by checking if the difference of the sequence-numbers of the incoming packet and the last heard packet before that is greater than 1) she sends a negative acknowledgment, NACK, using multicasting so all members of the session sees the NACK. But before sending the NACK she waits for a random time determined by the distance of the original source and if she hears a NACK for the same packet...
from another member she suppress her own NACK-transmission. See section 2.2 for a discussion of how the timers and the distance is calculated.

4) Any member that gets a NACK and has the packet requested can send a repair but just as in the NACK-case she waits a random time before sending the repair. Again see section 2.2 for the calculation of the repair-timers.

5) Loss is detected by finding a gap in the sequence space but since it is possible the last data-packet is dropped, each sender sends a low-rate periodic message, a heartbeat, announcing the highest sequence number used.

Please note that this only gives the basic ideas of the algorithms used and implementers should read the SRM-paper first.

2.2 Calculating timers in SRM

Every timer is based on the distance, D, between the sender and the receiver. A member that only is active in repairs is also called a sender.

2.2.1 NACK timers (point 6)

When loss is detected a NACK-timer is started. The expire-time is chosen from the uniform distribution of

\[ 2^i \times [C1 \times Dsa, (C1+C2) \times Dsa] \]

seconds, where Dsa is host A’s estimate of the one-way delay to the original source S of the missing data, C1 and C2 are parameters of the request algorithm and i is the count of how many times back-off has been issued. See section 2.2.3 for initial values of C1 and C2. The initial value of i is 0. When the request timer expires, host A multicasts the NACK for the missing data, and doubles the request-timer by increasing i by one to wait for the repair.

If host A receives a NACK for the missing data before its own request-timer for that missing data expires, host A does a exponential back-off, and resets its request timer. This means that the new timers expire-time is randomly chosen from the uniform distribution of

\[ 2^{(i+1)} \times [C1 \times Dsa, (C1+C2) \times Dsa] \].

To improve the repair time we don’t back-off the request timer for
every duplicate request that is received. For example, if a member receives several duplicate requests immediately after receiving the initial request, that member only backs off its request timer once. After the initial timer back-off, we only back-off the timer again if a request is received close to the time when the timer is set to expire. More precisely, when we back-off the request timer, we set an ignore-back-off variable to a time halfway between the current time and the expiration time, and we ignore additional duplicate requests until the ignore-back-off time.

2.2.2 Repair-timers (point 7)

When host B receives a request from A that host B is capable of answering, host B sets a repair timer to a random value from the uniform distribution of

\[ [D1*Dab, (D1+D2)*Dab] \]

seconds where \( Dab \) is host B’s estimate of the one-way delay to host A, and \( D1 \) and \( D2 \) are parameters of the repair algorithm. See section 2.2.3 for initial values of \( D1 \) and \( D2 \). If host B receives a repair for the missing data before its repair timer expires, host B cancels the timer. Otherwise, when host B’s repair timer for that data expires host B multicasts the repair.

A host could receive duplicate NACKs immediately after sending a repair. In order to prevent these duplicate NACKs from triggering a responding set of duplicate repairs, host B ignores NACKs for data \( d \) for \( 3*Dsb \) seconds after sending or receiving a repair for that data. Host \( s \) is either the original source of data \( d \) or the source of the first NACK.

2.2.3 Initial values of the timer parameters

The initial values of the timer parameters should be set to \( C1=C2=2 \), \( D1=D2=\log_{10}(G) \), where \( G \) is the current number of members in the session.

An adaptive algorithm for changing these parameters is presented in [2].

2.3 Calculating the host-to-host distances

In order to be able to calculate the NACK and repair timers we need to have some knowledge of the host-to-host distance. This distance is calculated in seconds based on how long it takes for a packet to travel from host A to host B.
To be able to do this each member of the session sends a time-stamp which is used in the following way to calculate the host-to-host distance:

Assume that host A sends a session packet P1 at time t1 and host B receives P1 at time t2. At some later time, t3, host B generates a session packet P2, containing (t1, d), where \( d = t3 - t2 \) (time t1 is included in P2 to make the algorithm robust to lost session packets). Upon receiving P2 at time t4, host A can estimate the latency from host B to host A as \( \frac{t4 - t1 - d}{2} \). Note that while this estimate does assume that the paths are symmetric, it does not assume synchronized clocks.

3. How does SRM fit into RTP?

Here follows a description of how the SRM ideas fit into RTP according to the points in the earlier sections. Points marked with a needed changes.

1) RTP has a unique sender-ID, the Synchronization Source (SSRC) and each data-packet has a sequence-number.

2) The buffering can be accomplished without interfering with the protocol itself.

3*) The transmission of NACKs has to be incorporated into RTP using for instance the Real-time Transport Control Protocol, RTCP.

4*) The transmission of repairs have to be incorporated into RTP.

5*) The heartbeats have to be incorporated into RTP/RTCP.

6) The timer calculations don’t imply any changes to RTP/RTCP.

7*) The time-stamps should be incorporated into RTCP. The NTP-time-stamp in the RTCP/SR packet is to be used but this implies that all clients has to implement the usage of this field (the RTP specification has left it optional for clients to use this field). In order to calculate the host-to-host delay all clients need to have some notion of wall-clock time or elapsed time.

Out-of-order packets could initially be seen as lost packets and lead to a started NACK-timer but when the "lost" packet(s) arrives the NACK-timer should be cancelled and an ignore flag should be raised for a time of \( 3*Dsa \), where Dsa is the receivers notion of the distance to the sender. All NACKs received within this time should be
4. What extensions are needed to RTP/RTCP

This section explains the needed additions to RTP/RTCP. The new needed packet-formats are discussed section 5.

The additional SRM-generated-traffic can be incorporated into RTP basically in two ways; either incorporate all additional packets and data into the same channel as the original RTP or send all traffic on a separate multicast-group as a new channel.

The first method would allow for clients that don’t understand the SRM additions to benefit from the retransmitted repairs but would destroy their jitter-calculations and traffic-monitoring applications.

The second method means that all additional SRM-generated-packets are sent on a separate RTP-data/RTCP channel. Only "standard" RTP-data/RTCP packets are sent on the base-channel. This would allow clients that doesn’t understand the SRM extension to join and listen to the session. This is for instance preferable when doing semi-reliable audio/video transfers where clients understanding the extension could get extra value.

The second method is the one chosen and discussed in the rest of this draft. The added channel is called the "SRM-channel".

4.1 NACKs (point 3)

The NACK-transmissions should be implemented using RTCP by adding a new RTCP packet-type, 205 is proposed for now.

4.2 Retransmission of data-packets (point 4)

The retransmissions can be handled in three ways:

* Either we just let applications that don’t understand the SRM-extension to see the retransmitted packets as original data and interpret them as duplicates or late packets. This would be nice because applications that don’t understand SRM would benefit, but it would unfortunately 'break' traffic estimation and analysis programs. It would also break applications, like for instance the MBone-tool VAT, because they adjust the play-out delay according to the jitter of the incoming packets.

* Or all the retransmitted packets could be encapsulated using a
new payload-type for redundant data. This would not break existing applications as RTP states that unknown payload-types should be ignored.

* Or the extra data-packets could be sent on a separate RTP-channel using layered encoding. This means that the extension-understanding client would listen to at least two channels, the base-channel (an ordinary RTP-channel) and a secondary channel, the SRM-channel, where all SRM-originated traffic is sent. The SRM-channel should be run on a separate multicast group to benefit from group pruning.

The third method is to be used.

4.3 Heartbeats (point 5)

The heartbeats could be incorporated into the SR-packets using the "profile-specific-extensions" but two things argue against this:

* This isn’t inter-operable with other payload-specific-extensions as there can only be one header-extension.

* The SR-packets are only sent during and once right after a sender transmits data but the heartbeats should be sent all the time, so a receiver that have lost several packets directly after the end of a data-flow would notice that she has lost packets.

Instead the heartbeats should be sent using the new RTCP-SRM packet-type on the SRM-channel.

These heartbeats should be sent more often right after a sender has stopped sending so receivers can notice that they have lost packets more quickly. This isn’t a problem for small groups as the dynamic delay between RTCP-packets is small but in larger groups the delay could become a problem.

4.4 Time-stamps (point 7)

Time-stamps for active senders on the RTP-channel are calculated using the Sender- and Receiver reports (SR and RR) in RTP as described in section 6.3.1 of [RFC1889]. No extra information is needed for this.

This section describe how time-stamps for passive members should be calculated.

The time-stamps are added into the new RTCP-packet-type and divided into two types.
The first type is "time-stamp queries" where a member sends out his wall-clock time-stamp and every other member of the session is expected to answer this query using the second type "time-stamp replies".

A time-stamp query should be included in the first RTCP-packet that a new member sends out after joining the session.

Replies to pending queries (if any) should be sent each time we send a RTCP-packet. Several replies can be contained in the same RTCP-packet as it would lower the overhead. The time-stamp replies for a new member should have higher priority than replies to an old member.

When old receivers see a new member they should set a query-timer chosen randomly from the interval \([0.5, 5]\)*current_RTCP_interval with a minimum of 5 seconds and a maximum of 2 minutes. If they before the expire don’t see another query they send out a query of their own. If they on the other hand do see a query they reset their query-timer and choose a new random expire-time.

Each 5*current_RTCP_interval since last query, (minimum 2 minutes, maximum 5 minutes) the member sends out a new query.

5. Packet format

This section explains the format of the new RTCP-SRM packets.

A new RTCP packet type is defined, PT = 205.

5.1 Header format

The header is the same for all RTCP-SRM packets:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|CM | Count |       PT      |             length            | header
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Version (V): 2 bits
Identification the version of RTP, two (2).

Command (CM): 2 bits
The SRM-command as defined below.

Count: 4 bits
The number of command-data-fields minus one included in this
packet. How this field is used depends on the command, see below.

<table>
<thead>
<tr>
<th>Packet type (PT)</th>
<th>8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains the constant 205 to identify this RTCP packet as a SRM packet.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packet length (length)</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>The length of this RTCP packet in 32-bit words minus one, including the header. (The offset of one makes zero a valid length and avoids a possible infinite loop in scanning a compound RTCP packet, while counting 32-bit words avoids a validity check for a multiple of 4.)</td>
<td></td>
</tr>
</tbody>
</table>

Depending on the Command (CM) field the header is followed by any of the following:

### 5.2 Heartbeat (CM=0)

For heartbeats CM contains the constant 0 and the header is followed by 16-bits containing the latest sequence number used by the sender when this report was issued. The next 16 bits contain the cycle number which indicate how many times the sender has wrapped his sequence number. This allows for clients detect that they have lost more that $2^{16}$ packets (when the sequence number counter wraps around) which can happen during net-failure and/or high transmission-speeds.

The sender is identified by the SSRC field in the SR or RR-report that must come before this SRM-packet.

<table>
<thead>
<tr>
<th>Sequence number</th>
<th>Cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1</td>
<td>++++++++</td>
</tr>
<tr>
<td>Sequence number: 16 bits</td>
<td></td>
</tr>
</tbody>
</table>
The highest sequence number used by the sender when sending this report. |
| Cycle number: 16 bits |
How many times the sender has wrapped his sequence number. |

### 5.3 Time-stamp queries (CM=1)

To be able to calculate the host-to-host delay a member has to send out a time-stamp. The time-stamp is composed of the 32 middle bits of
a 64 bits NTP time-stamp, meaning the 16 first bits signify the seconds and the later 16 bits signify the fraction.

The CM-field in the header contains the constant 1 and the count field is not used.

The header is followed by the following:

```
+----------------+-----------------+-----------------+-----------------+
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
| 0 1 2 3 4 5 6 7| 8 9 0 1 2 3 4 5| 6 7 8 9 0 1     |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
| 0 1 2 3 4 5 6 7| 8 9 0 1 2 3 4 5| 6 7 8 9 0 1     |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
```

```
<table>
<thead>
<tr>
<th></th>
<th>time-stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-------------------------------------</td>
</tr>
<tr>
<td></td>
<td>-------------------------------------</td>
</tr>
<tr>
<td></td>
<td>-------------------------------------</td>
</tr>
</tbody>
</table>
```

The senders current wall-clock time as defined above.

5.4 Time-stamp replies (CM=2)

Several time-stamp replies can be contained in the same SRM-packet and the number of replies are count+1.

A reply-packet for a certain receiver should only be issued if we earlier have received a time-stamp query.

The header is followed by the following:

```
+----------------+-----------------+-----------------+-----------------+
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
| 0 1 2 3 4 5 6 7| 8 9 0 1 2 3 4 5| 6 7 8 9 0 1     |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
| 0 1 2 3 4 5 6 7| 8 9 0 1 2 3 4 5| 6 7 8 9 0 1     |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
|                 |                 |                 |                 |
```

```
<table>
<thead>
<tr>
<th></th>
<th>SSRC_1</th>
<th>block1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Last time-stamp(LTQ)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DLTQ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SSRC_2</td>
<td>block2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

SSRC_n: 32 bits
The SSRC of the sender to whom we are answering.

Last time-stamp query (LTQ): 32 bits
The time-stamp that SSRC_n sent out earlier

Delay since last time-stamp query (DLTQ): 32 bits
The delay, expressed in units of 1/65536 seconds, between receiving the last time-stamp query packet from source SSRC_n and sending this reply.

If SSRC_n receives this reply at time A the host-to-host delay, D, can be calculated as

\[ D = \frac{(A - LTQ - DLTQ)}{2} \]

5.5 NACKs (CM=3)

Three different types of NACK-packets are currently defined:

- Single NACKs for specific packets from one sender.
- Sequential NACKs requesting a series of lost packets.
- Application specific NACKs.

Each RTCP-NACK can include NACKs for one sender, but several NACKs for different senders may be included into a compound RTCP-packet.

Different NACK-types are distinguished by the NACK-Type-field, NT.

Each NACK-request also include an urgent-flag (U) indicating (if raised) that this request should be prioritized over requests that don’t have the flag set. How this flag is used is application specific (see section 6).

5.5.1 Single NACKs, (NT=0)

This type of NACK is used for requesting lost packets by specifying each lost packet.

The CM-field contains the constant 0 and the count field contains the number of NACKs minus one included in this SRM-packet. This makes zero a useful number as it doesn’t make much sense to send an empty NACK packet just containing the SSRC.

The header is followed by the following:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------+-----------------+-----------------+-----------------+
| NT | U | not used | Cycle count |
|----------------+----------------+----------------|
+-----------------+-----------------+-----------------+-----------------+
| SSRC            |
+-----------------------------------------------+
```

Peter Parnes

[Page 11]
NACK-Type (NT): 2 bits
Indicates the type of the NACK as a single NACK, 0.

Urgent (U): 1 bit
Indicating that this is an urgent request.

Cycle count: 16 bits
The cycle count for the first sequence number as reported in an earlier heartbeat.

SSRC: 32 bits
The SSRC of the sender from whom we have lost the packets.

First/Last Sequence number: 32 bits
The sequence numbers of the packets lost from this SSRC. The number of sequence numbers is determined by the count-field in the header.

5.5.2 Sequential NACKs, (NT=1)
This type of NACK is used for requesting lost packets by specifying a sequence number and the number of following lost packets.

The CM-field contains the constant 1 and the count field contains the number of NACKs minus one included in this SRM-packet. This makes zero a useful number as it doesn’t make much sense to send an empty NACK packet just containing the SSRC.

The header is followed by the following:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------+-----------------+-----------------+-----------------+
| NT |U| Number of lost packets | Cycle count |
+-----------------+-----------------+-----------------+-----------------+
| SSRC            |
+-----------------+-----------------+-----------------+-----------------+
| First Sequence number |
+-----------------+-----------------+-----------------+-----------------+
NACK-Type (NT): 2 bits
```
Indicates the type of the NACK as a sequential NACK, 1.

Urgent (U): 1 bit
Indicating that this is an urgent request.

Number of lost packets: 13 bits
The number of lost packets following the specified sequence number.

Cycle count: 16 bits
The cycle count for the first sequence number as reported in an earlier heartbeat.

SSRC: 32 bits
The SSRC of the sender from whom we have lost the packets.

First Sequence number: 32 bits
The sequence number of the first packets lost from this SSRC in this sequence of lost packets.

5.5.2 Application specific NACKs, (NT=2)

This type of NACK is used for requesting lost packets in a way that is specific for the application using this RTP/SRM scheme. It can be used by applications to optimize the buffering by allowing repairers to reconstruct repair-packets from some other representation of the data. This could be used for file-transmissions where the incoming data is transformed into a file.

The CM-field contains the constant 2 and the length of this packet is determined by the length field in the header.

The header is at-least followed by 4 bytes where the first byte is defined as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|NT | Un-specified |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
: ... : ...
: ... : ...
: ... : ...
```

NACK-Type (NT): 2 bits
Indicates the type of the NACK as a application specific NACK, 2.
6. Semi-reliable sessions

Not all applications can wait a long time-period before receiving a repair for a lost packet. For instance, real-time data as audio and video that is played out almost immediately as received would have to get their repairs almost immediately. This leads to that the NACKs for such streams has to be prioritized over other NACKs if contained within the same RTP-session. The U-flag in the NACK-RTCP-packet can be used for this.

For some applications it might not make any sense in receiving the repair after a certain time-period (as the real-time data might already have been played out). These applications might decide not to retransmit a certain repair if the time since they received the NACK plus the network distance is larger that some number. This number is application specific.

This type of sessions are called semi-reliable sessions.

7. Further issues

[2] presents algorithms for dynamically adjusting the timer parameters C1,C2,D1 and D2. These algorithms should be included but do not imply any changes to the actual packet-format.

[2] also presents methods for "local recovery" meaning that we don’t multicast NACKs to the whole session but instead minimize the scope of the NACKs by adjusting the TTL. This TTL is adaptively changing as clients get to know their "loss neighbourhood".

8. Acknowledgments

I’d like to thank Jon Crowcroft, Anders Klemets and Todd Montgomery for creative comments and encouragements.

This work is done within the Multimedia Assisted distributed Tele-Engineering Services, MATES, project (ESPRIT 20598). I want to thank the Department of Computer Science and the Centre for Distance-spanning Technology at the Lulea Technical University for giving me the opportunity to do this work.

9. Author’s Address

   Peter Parnes

Peter Parnes
10. References
