Protocol Synchronization with Sparse Timestamps

Jorge A. Cobb* Mohamed G. Gouda* Prathima Agrawal†

*Department of Computer Sciences, The Univ. of Texas at Austin, Austin, TX
†AT&T Bell Laboratories, Murray Hill, NJ

Abstract

We present a family of transport protocols for transmitting messages from a sender process to a receiver process over a one-directional channel that may reorder, duplicate, or lose messages. In these protocols, each sent message is timestamped with the real-time of the sender when the message is sent. These timestamps are sparse, rather than uniform. We show how such timestamps, despite their sparseness, can be used by the receiver to deliver the received messages in order and without duplication. Because the communication is one-directional, the receiver cannot recover lost messages. Nevertheless, we show how the receiver can detect any message loss, then continue to receive and deliver each message that was sent after the lost message.

1. Introduction

In this paper, we present a family of transport protocols for transmitting messages from a sender process to a receiver process over a one-directional channel that can reorder, duplicate, or lose messages. In these protocols, each sent message is timestamped with the real-time of the sender when the message is sent. These timestamps serve as unique message identifiers, which allow the detection of message reorder and duplication, making time-to-live algorithms like those in [Sloa83] unnecessary.

Unfortunately, these timestamps are sparse (rather than uniform), and so the receiver needs additional information in order to be sure that it has received all the sent messages, and so can deliver them. The additional information needed by the receiver are lower and upper bounds on the time between successive messages from the sender.

Because of these bounds, our protocols are rate controlled, and so they are suitable for avoiding congestion in networks [Doe90]. Note that the lower bound ensures that the sender does not exceed the rate granted to it by the network, while the upper bound ensures that clock information is sent periodically from the sender to the receiver. Having the processes in a network exchange their clock information periodically can aid in the management of clocks in the network [Mil91] [Lis93].

Our family of protocols is based on the abstraction of a one-directional channel from the sender to the receiver. This abstraction is useful in two classes of applications: delivery of real-time data, and data transfer over channels with a large bandwidth-delay product. In the former, retransmission of data is uncalled for, since the data will most likely miss its deadline and be of no value. In the latter, a large amount of data may be transmitted before the first message arrives to the receiver [Kle92], and it is likely that all the data may have been sent before any feedback is received from the receiver. Thus, the receiver should be able to continue processing messages in order as they arrive, without waiting for the later retransmission of lost messages. Furthermore, future feed-forward correction mechanisms [Mca90] coupled with congestion control will reduce significantly the probability of message loss.

The paper is structured as follows. In Section 2, we introduce sparse timestamps, and discuss how they can be used to detect message reorder and duplication. The specification of our basic protocol is given in Section 3, and a proof of its correctness is given in Section 4. In Section 5, we discuss how to extend the basic protocol to detect message loss. In Section 6, we examine under what conditions can the sender change the sending rate without causing incorrect message delivery at the receiver. In Section 7, we address how can the protocol recover when the sender is forced to violate its expected behavior due to unexpected clock updates. A summary and future work are presented in Section 8.

2. Sparse Timestamps

We consider a system consisting of two processes, a sender and a receiver. The sender creates and sends a sequence of data messages to the receiver over a channel that may duplicate or reorder messages. Message loss is addressed in Section 5.

The receiver delivers the received messages one at a time to its application in the same order in which they were sent by the sender. Therefore, the receiver must be able to determine if a message it receives is a duplicate of a previously received message and must be discarded, or if the message has not been received earlier and must be delivered. Also, to deliver messages in the right order, the receiver must be able to determine, for any two messages p and q, if any messages were sent after p and before q.

If the sender sends message p before sending message q, we say that p is an *ancestor* of q, and q is a *descendant* of p. If q is the next message sent after sending message p, we say that p is the *predecessor* of q, and q is the *successor* of p.

The sender timestamps each data message with the time at which the message is sent, in order to convey information about its clock to the receiver. Since timestamps are unique and monotonically increasing, the receiver can use this timestamp to detect duplicates and to determine the relative order of any two messages. However, for any two messages p and q, the receiver cannot determine if q is the successor of p. That is, the timestamps of p and q give no indication to whether another message exists between p and q, with a timestamp larger than that of p but smaller than that of q. This is because timestamps are sparse message

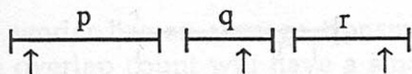

Figure 1

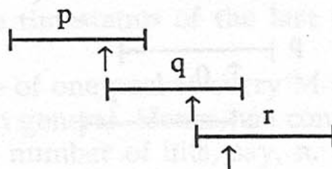

Figure 2

identifiers which, in contrast to sequence numbers, do not increase by one after each message.

To solve this problem, we need to restrict the timestamp values so that it can be determined if a message is the successor of another. One way in which the receiver can determine if one message succeeds another is if the timestamp of each message is a function of the timestamp of its predecessor. This, however, is too restrictive, because it would force the sender to transmit each message at a particular point in time. Instead, we associate with each message an interval of values in which its timestamp must occur. This interval is a function of the timestamp of the message's predecessor. In this way, the timestamp of a message is not restricted to a single value, and the receiver can determine if q is the successor of p by computing the interval of p's successor and checking if q's timestamp is within this interval.

Consider Figure 1, which depicts three messages, p, q, and r, and their associated timestamp intervals. Symbol ├───┤ represents the expected interval for the timestamp of a message, and the timestamp within this interval is indicated by an up arrow. The first message sent is p, the second q, and the last is r.

Let p be the last message delivered by the receiver. The receiver determines the interval for the timestamp of p's successor as a function of p's timestamp. When message q arrives, the receiver delivers q to the application because its timestamp is within the computed interval. If message r arrives before q, then r is not delivered to the application because its timestamp is not within the computed interval, and r is stored in the receiver's memory until its predecessor (namely, q) is received and delivered.

The above scheme can be implemented once an interval function has been selected. We next consider the effects of a simple function: all intervals have a fixed size M, and each interval begins immediately after the timestamp of the previous message. Let ts(p) denote the timestamp of p. For any pair of messages p and q, if q is the successor of p, then

$$ts(p) < ts(q) \leq ts(p) + M$$

The sender process can implement this by ensuring that no more than M seconds elapse between sending two consecutive messages.

The above function allows the intervals of different messages to overlap, which introduces a problem if the timestamp of a message is contained not only within its own interval, but also within the intervals of its ancestors. Consider the example of Figure 2, where the timestamp of r is contained within the inter-

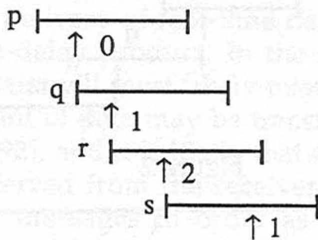

Figure 3

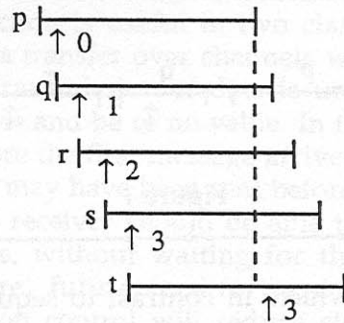

Figure 4

val of q, and p is the last message delivered. In this case, if q and r were reordered by the channel, the receiver will incorrectly accept r as the successor of p. Thus, when intervals overlap, additional information is needed in each message to prevent confusion at the receiver.

To indicate that the timestamp of a message is contained within the intervals of some of its ancestors, we include with each message an overlap count. The overlap count of a message p indicates how many intervals of the ancestors of p contain the timestamp of p.

Consider now Figure 3, where we write the overlap count alongside the up arrow indicating the message's timestamp. Since p has no ancestors, its overlap count is zero, i.e., its timestamp is not contained within any interval of its ancestors. The timestamp of q is contained in the interval of p, and thus its overlap count is 1. The timestamp of r is contained in the intervals of both p and q, and thus its overlap count is 2. Finally, the overlap count of s is 1, since its timestamp is contained only in the interval of r.

Let p be the last message that the receiver has delivered to its application. Since the receiver delivers messages in order, all ancestors of p have been received and delivered, and the receiver can store the intervals of these messages in its memory. To test if a message q is the successor of p, the receiver calculates the interval of the successor of p from p's timestamp. If q's timestamp is not contained in this interval, q is not p's successor. Otherwise, the receiver calculates how many ancestors of p (including p) have intervals which contain q's timestamp. This number is equal to q's overlap count if and only if q is the successor of p. In this manner, the receiver can guarantee to deliver messages in order.

The above follows from the observation that if q's timestamp is contained in the interval of its ancestor p, then it is also contained in the intervals of all messages sent after p and before q. Thus, if q's overlap count is larger than the count computed by the receiver, then other messages were sent after p and before q.

Note that the receiver need not remember the intervals of all messages received. Since timestamps are monotonically increasing, it only needs to remember those intervals whose right endpoint is larger than the timestamp of the last delivered message, because these are the only intervals which can contain the

successor of this message. Similarly, the sender only needs to remember those intervals whose right endpoint is larger than the timestamp of the last message sent.

If the sender has an average transmission rate of one packet every M seconds, then the overlap count will have a small value in general. Hence, we consider restricting the overlap count to a small and fixed number of bits, say, n. This implies that the sender must delay the transmission of a message until the current value of the clock is not contained within more than $2^n - 1$ previous intervals. Also, it implies that the receiver and the sender must remember at most the intervals of the last $2^n - 1$ messages delivered and the last 2^n messages sent, respectively.

As an example, let n = 2, and consider the scenario in Figure 4. In this case, the sender does not send message t until the interval of message p expires, since doing otherwise requires a overlap count of 4, which cannot be represented with only two bits. All intervals of p's ancestors expire before p's interval expires, and thus the sender only needs to remember the intervals of messages p, q, r, and s when sending message t.

We next consider how to eliminate the need of the sender to remember the intervals of the last 2^n messages sent. First, we would like to schedule the transmission of messages in a way that the sender does not need to check the intervals of previous messages to determine the time at which it can send the next message. To do so, we bound the time between sending two consecutive messages to a minimum of m and a maximum of M seconds. That is, if q is the successor of p,

$$m \leq ts(q) - ts(p) \leq M \qquad \text{(i)}$$

In this manner, at most $\lfloor (M-m)/m \rfloor$ messages other than q can occur in the interval [ts(p)+m, ts(p)+M], or equivalently, a timestamp can be contained in at most $\lfloor (M-m)/m \rfloor$ previous intervals. Thus, we require $\lfloor (M-m)/m \rfloor \leq 2^n-1$, which is equivalent to

$$M < (2^n + 1)\cdot m \qquad \text{(ii)}$$

if M/m is non-integer, and to

$$M \leq 2^n\cdot m \qquad \text{(iii)}$$

if M/m is integer. Therefore, the sender is guaranteed that it can send message q in the interval [ts(p)+m, ts(p)+M], provided (ii) and (iii) hold.

Although the sender no longer needs to check if old intervals have expired before sending the next message, by including an overlap count in the message, the sender still needs to remember the last 2^n intervals. However, we can eliminate this by the following observation. Since at most 2^n messages are contained in each message interval, we can uniquely label each of these messages using n bits. That is, we replace the overlap count by a sequence number (sn) of n bits. If q is p's successor, we set

$$sn(q) = (sn(p) + 1) \bmod 2^n \qquad \text{(iv)}$$

In this way, the receiver knows that q is the successor of p if and only if both (iv) and (i) hold. Furthermore, the sender is only required to remember the times-

tamp and sequence number of the last message sent. Similarly, the receiver is only required to remember the timestamp and sequence number of the last message delivered.

The sequence number provides the sender with flexibility in its sending rate. The allowed difference between the maximum and minimum sending rate increases exponentially with n, as indicated by (ii) and (iii). Thus, n need not be large, and may even be zero, in which case $m < M < 2 \cdot m$ must hold.

3. Protocol Specification

In this section, we specify the timestamp protocol described informally above. We use the specification language of [Gou93], which we briefly overview next.

A program consists of a set of *processes*. Our protocol consists of two processes, a sender S and a receiver R. Each process consists of a set of *actions* separated by the symbol [], using the following syntax:

begin action [] . . . [] action **end**

Each action is of the form *guard* → *command*. A guard is either a Boolean expression on the state of the process or a receive (**rcv**) statement. A command is constructed from sequencing (;), iterative (**do od**) and conditional (**if fi**) constructs. These constructs group together assignment statements and **send** statements.

Since messages may be received in an order different from that in which they were sent, a channel is represented as a set of messages. Sending a message consists of adding the message to this set, and receiving a message consists in removing an arbitrary element from this set. Since S sends messages to R, the output channel of S is the same set as the input channel of R.

An action is said to be *enabled* if its guard is a Boolean expression that evaluates to true, or if its guard is a receive statement and there is a message in the process' input channel.

An execution step of the system consists of choosing any enabled action and executing the action's command. If the guard of the action is a receive statement, then, before the action's command is executed, a message is removed from the input channel and copied into the indicated local variables. If the statement being executed in the command is of type send, then the message indicated in the statement is added to the output channel. Executions are fair — no action can remain continuously enabled without eventually being executed.

For conciseness, we use the multiple assignment statement

$$v_1, v_2 := E_1, E_2;$$

which evaluates the two expressions E_1 and E_2 before assigning any of them to its variable v_1 or v_2.

The specification of the sender process S is given in Figure 5. The constant n indicates the number of bits for the sequence number sn, and constants m and M indicate the intended minimum and maximum interval for two consecutive timestamps. The real-time clock is modeled with the variable clock. Its is defined as an integer and not as a real, since in practice, a real-time clock has discrete val-

```
process S
const  n, m, M  :  integer
var    lts      :  integer,          {last timestamp sent}
       lsn      :  0 .. 2^n - 1,     {last sequence number sent}
       clock    :  integer
begin
   lts + m ≤ clock  →   lts, lsn := clock, (lsn+1) mod 2^n;
                        send data(lts, lsn) to R
[] clock < lts + M  →   clock := clock + 1
end
```

Figure 5: Sender process

ues. The timestamp and sequence number of the last message sent are stored in lts and lsn, respectively.

The process contains two actions, one to send a message, and the other to increment the clock. The guard of the first action ensures that at least m seconds have elapsed from the transmission of the previous message. The action sends a message to the receiver consisting of a timestamp, which is taken from the variable clock, and a sequence number, which is the last sequence number plus one modulo 2^n. These two values are stored in lts and lsn. The second action increments the clock, without increasing it beyond the expected interval for the next timestamp.

The specification of the receiver process is given in Figure 6. The receiver shares constants n, m, and M with the sender. The timestamp and sequence number of the last message delivered are stored in variables dts and dsn, respectively. Since messages may arrive out of order, the receiver stores incoming messages into the set buff until they are delivered. The function min(buff) returns the message with the smallest timestamp stored in buff.

The process has two actions. In the first action, a message is received and stored in the buffer. In the second, the message with the smallest timestamp in the buffer is found. If this message is the successor of the last delivered message, the message is delivered and removed from the buffer. Otherwise, the buffer remains unchanged.

The initial state of this system satisfies lts = dts, lsn = dsn, and clock ≤ lts + M. In Section 7 we show how the initial state can be relaxed to lts = 0, lts = dts, and lsn = dsn.

4. Protocol Verification

Let T.1, T.2, . . . be the timestamps of the messages sent by the sender, where T.i is the timestamp of the ith message sent, and let T.0 be the initial value of lts and dts. The index into T of element ts is denoted ind(ts), i.e., T.ind(ts) = ts. The last

```
process R
const  n, m, M  :  integer
var    dts      :  integer,              {last timestamp delivered}
       dsn      :  0 .. 2^n - 1,         {last sequence number delivered}
       ts       :  integer,
       sn       :  0 .. 2^n - 1,
       buff     :  set of data(integer, 0 .. 2^n - 1)      {buffer}
begin
   rcv data(ts, sn) from S
                    → if ts > dts → buff := buff ∪ { data(ts, sn) }
                      fi
[]  buff ≠ ∅        → data(ts, sn) := min(buff);
                      if  sn = (dsn+1) mod 2^n ∧ ts > dts ∧ ts - dts ≤ M
                         →        deliver data(ts, sn);
                                  buff := buff - data(ts, sn);
                                  dts, dsn := ts, sn
                      fi
end
```

Figure 6: Receiver Process

element of T is T.ind(lts). Moreover, since sequence T is increasing, for any ts and ts' in T,

$$ts > ts' \Leftrightarrow ind(ts) > ind(ts') \qquad (v)$$

The following are invariants of the protocol, that is, they are satisfied by the initial state, and can be easily shown to continue to hold after the execution of any action.

1. $lsn = ind(lts) \bmod 2^n$.
2. for all data(ts, sn) in the channel, $ts \in T$, and $sn = ind(ts) \bmod 2^n$.
3. $dts \in T$, and for all data(ts, sn) $\in$ buff, $ts \in T$, $sn = ind(ts) \bmod 2^n$, and $ind(dts) < ind(ts)$.

Our obligation is to show that a message in the buffer with timestamp ts is delivered if and only if

$$ind(ts) = ind(dts) + 1 \qquad (vi)$$

In order to show this, we present the following theorem, which provides the foundation for the argument.

Theorem 1

If

a) $T.i < T.(i+1)$, for any i, $0 \le i \le max(k, j+2^n)$

b) $T.j + 2^n \cdot m \le T.(j+2^n)$

c) $M < (2^n+1)\cdot m$

d) $T.(k-1) + m \le T.k$

then:

$$T.j < T.k \wedge T.k \le T.j + M \wedge j + 1 \bmod 2^n = k \bmod 2^n$$
$$\Leftrightarrow$$
$$k = j + 1 \wedge T.k \le T.(k-1) + M$$

Proof

($\Leftarrow$)

$k = j + 1$ implies $k > j$. Since T is increasing up to index k (see a), this implies

$$T.k > T.j$$

Furthermore, $k = j + 1$ implies that

$$j+1 \bmod 2^n = k \bmod 2^n$$

Lastly, $k = j + 1$ implies $k - 1 = j$, and hence $T.k \le T.(k-1) + M$ implies

$$T.k \le T.j + M$$

($\Rightarrow$)

Since T is increasing up to both index k and index j (see a), $T.j < T.k$ implies

$$k > j$$

From (d) and $T.k \le T.j + M$ it follows that

$$T.(k-1) + m \le T.k \le T.j + M$$

Dropping T.k, and using (c)

$$T.(k-1) + m < T.j + (2^n+1)\cdot m$$

Subtracting m to both sides,

$$T.(k-1) < T.j + 2^n\cdot m$$

The above and (b) imply

$$T.(k-1) < T.(j+2^n)$$

Since T is strictly increasing up to index k and index $j + 2^n$ (see a),

$$k - 1 < j + 2^n$$

Adding 1 to both sides, and, because $k > j$,

$$j < k < j + 2^n + 1$$

The above together with $j+1 \bmod 2^n = k \bmod 2^n$ imply that

$$j + 1 = k$$

The above and $T.k \le T.j + M$ imply

$$T.k \le T.(k-1) + M$$

■

Let data(ts, sn) be the message the receiver is considering for delivery. From Invariant 3, $ts \in T$ and $ind(ts) \bmod 2^n = sn$. Let $k = ind(ts)$ and $j = ind(dts)$. For the moment, assume $ind(lts) \ge j + 2^n$.

Since the sender timestamps messages at least m seconds apart, conditions (a), (b) and (d) of Theorem 1 hold. Also, condition (c) is the known requirement on the constants of the protocol. Hence, all conditions of the theorem are satisfied.

The left hand side of the equivalence of Theorem 1 corresponds to the test the receiver applies to the message to decide if it should be delivered next. From the

theorem, if the message passes the test, then k = j+1, or equivalently, ind(ts) = ind(dts)+1, as desired.

For progress to occur, the message must pass the receiver's test when k = j+1. Note that the sender ensures that $T.k \le T.(k-1)+M$ holds. Thus, from the theorem, the receiver's guard must be true, causing the message to be delivered and progress to occur.

We next consider the case $j \le ind(lts) < j+2^n$. Conditions (a) and (b) no longer hold since entries in T with index larger than ind(lts) have not been assigned a value, and thus we can't use Theorem 1. Instead, from Invariant 3, k > j, and hence,

$$j < k \le ind(lts) < j+2^n$$

The receiver tests whether $j+1 \bmod 2^n = k \bmod 2^n$, which, in conjunction with the above, implies k = j+1, as desired.

To show progress, the message must pass the receiver's test when k = j+1. In this case, from (v), T.k > T.j. Furthermore, k = j+1 implies $j+1 \bmod 2^n = k \bmod 2^n$, and it also implies $T.k \le T.j+M$. Hence, the receiver's guard is true.

Notice that, in our argument that the next message delivered is T.(j+1), we made very weak assumptions about the messages delivered in the past and about messages yet to be delivered. In (a), we only require that past messages have increasing timestamps, and place no bounds on the difference of two consecutive timestamps. Also, in (b), we restrict the value of $T.(j+2^n)$, but not of elements beyond this one. This hints that it may be possible to change the values of m and M during the course of the data transmission by informing the receiver of this change in data messages. This possibility is examined in Section 6.

5. Message Loss

It is desirable for the receiver to be able to detect message loss in the incoming stream of data messages, so that it can continue processing the incoming messages in order, rather than wait for the lost messages. In this section, we describe how the timestamps of incoming messages can be used to detect message losses.

Let xdel and ndel be the maximum and minimum delays that a message may experience in the channel from the sender to the receiver. The maximum difference between the delays of any pair of messages is therefore xdel - ndel.

Assume that two messages p and q have been received, and that the following relation holds.

$$ts(q) > ts(p) - m + (xdel - ndel)$$

In this case, all ancestors of p not yet received must have been lost. This is because any ancestor of p must have a timestamp of at most ts(p) - m, and thus can arrive no later than time ts(p) - m + xdel. Since q must arrive no earlier than time ts(q) + ndel, from the above relation, ts(q) + ndel > ts(p) - m + xdel. Hence, no ancestor of p can arrive after the arrival of q.

The detection of message loss requires no changes to the sender's specification. The receiver requires only a small addition to its second action. The new action is shown in Figure 7, where max(buff) returns the message with the largest times-

```
buff ≠ ∅        →   data(ts, sn) := min(buff);
                    data(t, s) := max(buff);
                    if  (sn = (dsn+1) mod 2^n ∧ ts > dts ∧ ts - dts ≤ M) ∨
                        (t > ts - m + xdel - ndel)
                        →       deliver data(ts, sn);
                                buff := buff - data(ts, sn);
                                dts, dsn := ts, sn;
                    fi
```

Figure 7: Message loss detection

tamp contained in buff. A message is delivered if, as before, it is the successor of the last delivered message, or if it is determined that its ancestors have been lost.

6. New Interval Boundaries

In this section, we discuss the conditions necessary for the sender to change the bounds on the time between sending consecutive messages, without causing messages to be delivered out of order at the receiver. The number of bits for the sequence number, n, is assumed to remain constant.

Let the sender and receiver have knowledge of two pairs of bounds on the difference between consecutive timestamps, namely, (m.0, M.0) and (m.1, M.1). Without loss of generality, assume the sender is using bounds (m.0, M.0), and wishes to switch to bounds (m.1, M.1). To notify the receiver of which bounds are in use, each message contains a control bit b indicating which bounds will be used to send its successor. That is, let q be the successor of p. If b(p) = 0, then

$$ts(p) + m.0 \le ts(q) \le ts(p) + M.0$$

If b(p) = 1, then

$$ts(p) + m.1 \le ts(q) \le ts(p) + M.1$$

Thus, when testing whether a message should be delivered next, the receiver uses the bounds indicated by the control bit of the last delivered message.

Due to message reorder, the bounds that the receiver currently uses may be different from those used to create the incoming message, and the receiver may deliver the message out of order. To prevent this, when changing from (m.0, M.0) to (m.1, M.1), the sender may temporarily need to timestamp messages in a manner satisfying *both* (m.1, M.1) and Theorem 1 with (m, M) instanciated as (m.0, M.0). We next examine whether this is feasible for different values of m.1 and M.1.

Consider first when m.0 ≤ m.1. In this case, conditions (a) through (d) of Theorem 1 hold. That is, m.0 ≤ m.1 ensures consecutive timestamps differ by at least m.0, and thus (b) holds. By the same token, (d) also holds. Hence, the receiver delivers messages in order, and the sender can switch immediately from using (m.0, M.0) to (m.1, M.1).

```
process S
const   n           :  integer,
        m, M        :  array [0 .. 1] of integer
var     lts         :  integer,
        lsn         :  0 .. 2^n - 1,
        clock       :  integer,
        em, eM      :  integer,
        b           :  0 .. 1,            {current bounds}
        bts         :  integer            {time of last change to b}
begin
   lts + em ≤ clock  →  if true → b, bts := ~b, clock;
                                   em, eM := max(m[0], m[1]), min(M[0], M[1])
                        [] true → skip
                        fi
                        lts, lsn := clock, (lsn+1) mod 2^n;
                        send data(lts, lsn, b) to R
[] clock < lts + eM  →  clock := clock + 1
[] (m[b] ≥ m[~b]) ∨ (clock > bts + max((2^n-1)·m[~b], M[~b] - m[~b]))
                     →  em, eM := m[b], M[b]
end
```

Figure 8: Sender with multiple interval bounds

Next, assume m.1 < m.0, and M.1 < m.0. With the new bounds, consecutive timestamps must differ by at most M.1 seconds, and thus, they cannot differ by at least m.0 seconds, which is necessary to satisfy condition (b). Therefore, new interval bounds satisfying m.1 < m.0 and M.1 < m.0 are disallowed.

Finally, assume m.1 < m.0 and m.0 ≤ M.1. In this case, it is possible to satisfy both the old and the new bounds by having all messages timestamped at least m.0 seconds and at most M.1 seconds from their predecessor. However, this should not continue indefinitely, since we would like to take advantage of the smaller interval bound m.1 and send at a faster rate. This should be done without violating conditions (b) and (d) of Theorem 1.

Let j be the index of the last delivered message, b(T.j) = 0, and c be the index of the first descendant of T.j such that b(T.c) = 1.

Condition (b) is satisfied if $j+2^n \le c$, because each message from T.(j+1) up to and including $T.(j+2^n)$ has a timestamp at least m.0 seconds larger than that of its predecessor. The case $j < c < j+2^n$ yields two scenarios. Consider first:

$$T.c + (2^n-1)\cdot m.0 \le T.(j+2^n)$$

Since bound m.0 is still in effect when T.c is sent, T.j + m.0 ≤ T.c. Thus,

$$T.j + 2^n \cdot m.0 \le T.(j+2^n)$$

and (b) holds. Consider next:

$$T.c + (2^n-1) \cdot m.0 > T.(j+2^n) \qquad \text{(vii)}$$

Relation (vii) does not imply that condition (b) holds. Furthermore, since the value of j is local to the receiver, the sender cannot test for (vii), so it must always assume that it's true. Thus, the sender must satisfy (b) by sending messages T.(j+1) through $T.(j+2^n)$ using lower bound m.0. It can do so without knowledge of j by continuing to use bound m.0 until the timestamp of the next message to send, T.k, satisfies the following.

$$T.k \ge T.c + (2^n-1) \cdot m.0 \qquad \text{(viii)}$$

We still need to satisfy condition (d). This is a condition on the current message the receiver is considering for delivery, T.k. If $k \le c$, then (d) holds, since the bound m.0 was still in effect when T.k was sent. If $k > c$, then (d) may not hold, and the receiver may incorrectly deliver the message. To prevent this, the sender must continue to satisfy (d) for all descendants of T.c until the timestamp of the message to send is large enough that the message will fail the receiver's test. That is, the sender can use m.1 and violate (d) provided

$$T.k > T.j + M.0$$

The sender cannot check the above directly, since it does not know the value of j. However, recalling that $T.j + m.0 \le T.c$, the above holds if

$$T.k > T.c + M.0 - m.0 \qquad \text{(ix)}$$

Combining relations (viii) and (ix), the sender can use the new bounds and disregard the old bounds when the timestamp of the next message to send, T.k, satisfies the following.

$$T.k > T.c + \max((2^n-1) \cdot m.0,\ M.0 - m.0)$$

We next present the specification of the protocol for varying interval bounds. The specification of the receiver is very similar to that of Figure 6, and is not presented here. Its only change is to make constant M into an integer variable, and update its value according to the control bit of the last delivered message.

The specification of the sender is shown in Figure 8. The bounds (m.0, M.0) and (m.1, M.1) are represented by the parallel arrays m and M. Bit b indicates which bounds are currently in use. Variable bts indicates the time of the last change to b. The bit-complement of b is denoted by ~b.

Variables em and eM indicate the effective bounds. That is, they are set to the current bounds if the previous bounds are no longer in effect. Otherwise, they correspond to the intersection of the previous and current bounds.

The first action is enhanced to non-deterministically choose whether or not to change the current bounds. If new bounds are chosen, em and eM are set to the intersection of the previous and current bounds. The second action allows no more than eM seconds between consecutive timestamps. A third action is added to update em and eM to the current bounds provided the previous bounds are no longer in effect.

7. Interval Violations

Although the sender may in general send data messages whose timestamp is within the expected interval, it is possible that the sender is unable to do so due to unforeseen events. In particular, the sender may have to update its real-time clock, making it impossible to send a message in the specified interval. It is also possible that the sender may be under a heavy processing or network load, and is unable to send a message within M seconds of sending the previous message. In this section, we examine each of these two problems in order.

Periodically, the sender process executes a clock synchronization algorithm, which may cause updates to its clock. The clock updates interfere with the timestamping of messages. If the clock is moved forward, it may no longer be possible to timestamp a message with a value at most M seconds larger than the timestamp of the last message. If the clock is moved backward, more than 2^n messages may overlap a single interval, and n bits are no longer sufficient to distinguish which message is the next to deliver. Next, we discuss the effects of clock updates and how to deal with them.

Let the timestamp of the last sent message be ts(p), the current time be t, and assume that the clock is updated forward by d seconds at time t, such that

$$t < ts(p) + M < t + d$$

In this case, the next message to be sent, q, will have a timestamp of at least t + d, violating the expected interval of the message. To resolve this problem, the sender includes in message q a deviation field dev with a value of d, and ensures that q's timestamp satisfies the following.

$$ts(p) + m \le ts(q) - dev(q) \le ts(p) + M \qquad (x)$$

This relation implies that q would have a timestamp in the interval [ts(p)+m, ts(p)+M] had the clock update not occurred. The receiver determines that q is p's successor if their sequence numbers are consecutive and relation (x) holds. If the dev field is not present in a message, the receiver assumes that its value is zero. Moreover, there is no risk that a future message r be mistaken for p's successor, since ts(r) - dev(r) is also at least m seconds larger than r's predecessor timestamp, and thus no more than 2^n messages can have a timestamp in the interval [ts(p)+m, ts(p)+M].

Regressing the clock is also a potential cause for problems. A clock regression can cause an incorrect delivery order even if the sender allows at least m and at most M seconds to elapse between consecutive messages. For example, let q be the successor of p, and let the clock be regressed after sending p and before sending q. The clock regression may cause q to have a timestamp and sequence number equal to those expected from an ancestor of p. If q is reordered in the channel, it may be delivered before p.

One approach to solve this problem consists of including a negative dev field to indicate that a clock regression occurred [Cob95]. This dev field must be included in all messages until the clock reaches a determined value, and it requires a limit on how frequent clock regressions may occur.

We adopt instead the approach in which the sender does not send a message after the clock regression until relation (i) holds. That is, if the clock is regressed d seconds after sending p, the sender is required to wait at least m + d and at most M + d seconds after sending p before it can send q. This approach causes the sender to wait an additional d seconds, which is acceptable under the assumption that most clock updates are relatively small. The receiver is oblivious to the clock regression, except for perceiving an increased delay for message q.

Consider now the case when the sender, due to a high processing demand or network load, is unable to send the next data message within M seconds of sending the previous one. If this is the case, the receiver must be informed that the packet has a timestamp outside of its expected interval. Notice that this is similar to the case of the forward clock update which causes a packet's timestamp to be outside its intended interval, except that the clock of the sender has not been increased. Therefore, the sender can also treat this case by including a dev field in the next message such that relation (x) holds.

The use of deviations in the protocol can be shown to be correct by augmenting Theorem 1 to include a deviation value [Cob95]. Similarly, the strategy for changing interval boundaries of Section 6 can be slightly modified to allow deviations.

The introduction of the deviation makes possible an initial state of the system with lts = 0 and dts = 0. In this manner, the sender and the receiver begin from a state in which the receiver need not have prior knowledge of the sender's clock. The large difference between dts and the timestamp of the first message may be viewed from the receiver's perspective as a large clock update occurring at the sender before the first message is sent, which will be reflected in the message's dev field.

8. Summary and Future Work

We presented a family of transport protocols for transmitting messages from a sender process to a receiver process over a one-directional channel that may reorder, duplicate, or lose messages. In these protocols, each message is timestamped with the time in which it is sent. The protocols feature an upper and lower bound on the transmission rate of the sender. For some M, the sender ensures that no more than 2^n messages are sent in any interval of M seconds. This is an example of the uniform property of a flow of data discussed in [CG93]. This property allows the calculation of an upper bound on the buffer capacity required and the delay introduced at every stage of a network.

The protocol was specified using the notation of [Gou93] and shown to be correct. We examined the conditions under which the bounds on the sending rate may be changed during the course of the data transfer without causing improper message deliveries at the receiver. Finally, we examined how the bounds on the sending rate can be violated occasionally due to clock updates and late transmissions.

Future work includes the design of a totally ordered multicast protocol that takes advantage of the messages' timestamps and the bounds on the time interval between consecutive messages to ensure that the relative delivery order of

any pair of multicast messages must be the same for all processes in the system [CG94]. Unlike other multicast protocols [May92], this can be achieved without having a single node as a coordinator of message ordering, and also without the sender performing multiple passes with its intended receivers. Since the protocol establishes a total order while maintaining the local sending order, the resulting order is both total and causal [FT93].

References

[Cob95] Cobb J., ``Flow Theory and the Analysis of Real-Time Protocols", Ph.D. Thesis, in preparation. Expected, 1995.

[CG93] Cobb J., Gouda M., ``Flow Theory: Verification of Rate-Reservation Protocols", *International Conference on Network Protocols*, 1993. Submitted for journal publication.

[CG94] Cobb J., Gouda M., ``Inception-Time Multicast", 1994, in preparation.

[Doe90] Doeringer W. et. al., ``A Survey of Light-Weight Transport Protocols for High-Speed Networks", *IEEE Transactions on Communications*, Vol. 38, No. 11, p. 2025.

[FT93] Florin G., Toinard C.,``A New Way to Design Casually and Totally Ordered Multicast Protocols", *Operating Systems Review*, 1993, pp 77-83.

[Gou93] Gouda M., ``Protocol Verification Made Simple", *Computer Networks and ISDN Systems*, Vol. 25, 1993, pp. 969-980.

[Kle92] Kleinrock L., ``The Latency/Bandwidth Tradeoff in Gigabit Networks", *IEEE Communications Magazine*, April 1992, Vol. 30, No, 4.

[Lis93] Liskov B., ``Practical Uses of Synchronized Clocks in Distributed Systems", *Distributed Cmputing*, Vol. 6, pp 211-219, 1993.

[May92] Mayer E., ``An Evaluation Framework for Multicast Ordering Protocols", *Proceedings of the 1992 ACM SIGCOMM Conference*, pp 177-187.

[Mca90] McAuley A. J., ``Reliable Broadband Communication Using a Burst Erasure Correcting Code", *Proceedings of the 1990 ACM SIGCOMM Conference*, p 197.

[Mil91] Mills D., ``Internet Time Synchronization: The Network Time Protocol", *IEEE Transactions on Communications*, Vol. 39, No. 10, Oct 1991, p. 1482.

[Sloa83] Sloan L., ``Mechanisms that Enforce Bounds on Packet Lifetimes", *ACM Transcations on Computers* , Vol. 1, No. 4, 1983.