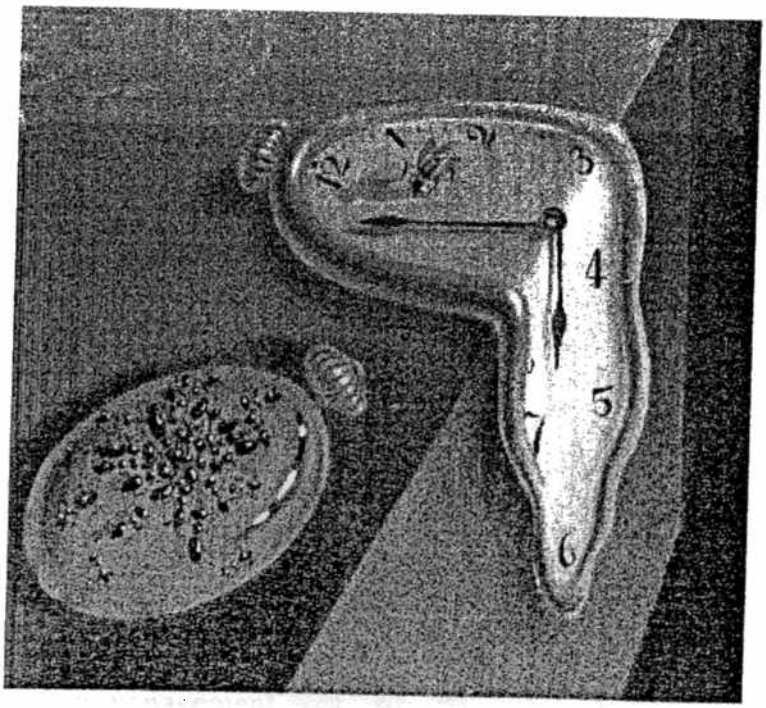
Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1–46
Mai 1994

SSCMP: The Sequenced Synchronized Clock Message Protocol

U. Schmid, A. Pusterhofer



Ausschnitt aus: Salvador Dali, "Die Beständigkeit der Erinnerung"

SSCMP: The Sequenced Synchronized Clock Message Protocol

U. SCHMID, A. PUSTERHOFER
Technical University of Vienna
Department of Automation
Treitlstraße 3, A-1040 Vienna.
Email: s@auto.tuwien.ac.at

September 13, 1994

Abstract

We present a novel timer-based connection management protocol SSCMP (*Sequenced Synchronized Clock Message Protocol*) providing reliable ordered at-most-once delivery of messages. Apart from being strikingly simple, it also surpasses existing protocols like Delta-t, VMTP, XTP, and CMSC with respect to correctness in case of clock synchronization failures. By means of a rigorous mathematical analysis, we show that our SSCMP implements a certain abstract specification on top of a fairly realistic system model, putting on a firm ground the very positive experiences we gathered from experiments with our actual implementation.

Keywords: timer-based connection management protocol, sequenced at-most-once message delivery, synchronized clocks.

1 Introduction

Connection-oriented protocols are important for several subtasks in distributed computing, ranging from reliable data transfer to remote procedure calls. Classical connection management protocols as used in TCP (see [Pos81]) employ *initial handshaking* for setting up connections: To ensure that a setup message is not a duplicate, sender and receiver must *exchange* setup messages before actual data transmission can take place.

Initial handshaking is perfectly reasonable for the infrequently setup but heavily used connections found in former generation computer networks, since the setup-overhead is effectively spread over all the messages sent via a connection. This, however, is no longer true for the communication patterns found in today's (client-server) distributed systems, where frequent connections to numerous servers with only a few (often two) messages exchanged per connection are common, cf. [CW89]. For example, the overhead of the initial handshake in TCP is one round-trip time, which is primarily determined by the signal propagation delay. It is therefore not decreased by the dramatically increasing transmission speeds and becomes in fact more and more unbearable as high-speed network technology evolves.

Timer-based connection management protocols like the pioneering *Delta-t* ([Wat81]) avoid that connection setup overhead completely, providing a promising alternative. The basic idea

underlying such protocols is to remember "recently" received messages in order to detect duplicated setups. This is made working by somehow enforcing a *maximum packet*[1] *life/validity time*, and the few existing timer-based protocols differ in how this is actually accomplished.

Any timer-based protocol relies on a common "idea" of time among all the nodes of the distributed system, and it has been realized early that such protocols may be both considerably enhanced and simplified by assuming that nodes are equipped with synchronized clocks; see [Che89, LSW91, BF93]. Much effort has been devoted to the development of (inexpensive) techniques for clock synchronization (see [SWL90, RSB90] for an overview), and high-accurate, inexpensive sources of *universal time coordinated* (UTC) are worldwide available now via the NAVSTAR *global position system* GPS (see [Wel87]). Actually, the development of the *network time protocol* NTP (see [Mil91]) has pushed synchronized clocks even into Internet-reality. In view of the trend towards integrating (large) distributed systems into our daily life (which is governed by time, i.e., UTC) in conjunction with the fact that the availability of a common notion of time greatly simplifies the design of distributed services (see [Lis93]), we think that it is not unreasonable to predict that future generation computer systems will be equipped with accurately synchronized clocks, see [Sch94] for related issues.

Our research into that area originates in the problem of providing a high-performance, reliable, low-level communication subsystem for a certain distributed real-time system[2]. In the course of that work, we eventually developed and implemented a novel timer-based connection management protocol SSCMP, which is a non-trivial extension of the (unsequenced) *Synchronized Clock Message Protocol SCMP* of [LSW91]. Our protocol has a number of advantages over existing ones, most notably the guarantee of ordered at-most-once delivery even in the case of clock synchronization failures.

The presentation of that *Sequenced Synchronized Clock Message Protocol SSCMP* contained in this paper is organized as follows: In Section 2, we discuss the system model underlying our investigations and the general features of the protocol. Section 3 is devoted to the detailled description of the SSCMP, Section 4 contains the analysis of its properties. A comparison to other existing protocols may be found in Section 5, and some conclusions and directions of further research are appended in Section 6.

2 Protocol Features and System Model

Connection management is of course needed for connection-oriented protocols at any layer of the protocol hierachy. Most existing work ([Wat81, Che86, SDW92, BF93]) on timer-based protocols, however, solely addresses the transport layer, adhering to the common style of transport protocol usage (involving an interface supporting connection open, transfer, and close phases explicitely) and treating issues like adressing, rate control, ... in full detail.

We, however, prefer a both simplified and more abstract point of view, which hides away even the concept of connections from the usage (i.e., the *service specification*, according to the terminology of [Sha91]) of the (basic) protocol: As advocated in [Wat81] and in particular in [Che86], we view our protocol as being responsible for providing reliable, sequenced, packet-oriented but otherwise unstructured data transmission on top of a necessarily imperfect

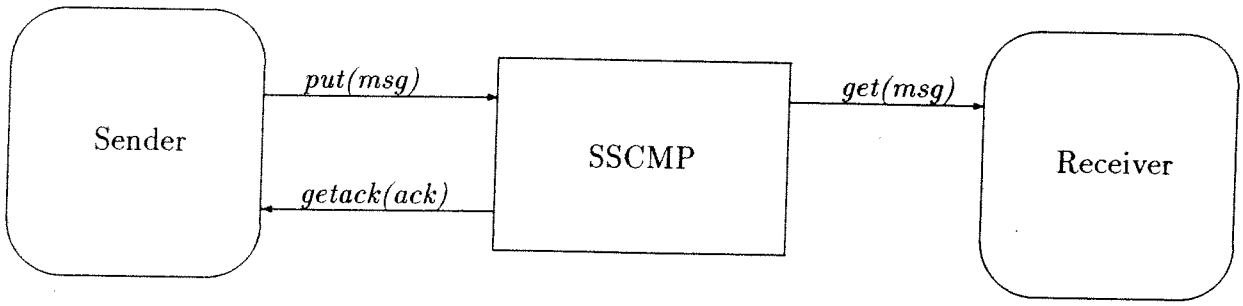
[1] We will use the terms *packet* and *message* synonymously.

[2] Our monitoring system *Versatile Timing Analyzer VTA*, a research project supported by the Austrian Science Foundation, grant no. P8390-TEC.

communication system. It is understood, however, that there are connections involved in the *protocol (entity) specification* of SSCMP, but they are internally managed and hence invisible from the outside.

This approach has several advantages: First of all, dealing with timer-based protocols invented to *avoid* connection setup overhead, we feel that adhering to a connection-based interface would —in some sense— give away some of that advantages. Moreover, relying on a low-level interface makes our basic protocol applicable for data link layer and transport layer protocols as well; constructing a proper transport layer interface dealing with all the important details like adressing, rate control, ... should be relatively straightforward. Last but not least, both presentation and analysis are considerably simplified by confining ourselves to the problem of reliable, sequenced communication.

We view our basic protocol as a (conceptual) black-box connecting a particular pair of sender and receiver for unidirectional data communications purposes as follows:



The interface to and from the protocol comprises only three functions (actions):

- *put(msg)* is used by the sender to submit a message *msg* to the protocol for transmission.
- *getack(ack)* is used by the protocol to deliver acknowledgments regarding the transmission of previously *put* messages; $ack \in \{\mathrm{OK}, ?\}$.
- *get(msg)* is used by the protocol to deliver messages to the receiver.

Any protocol that provides reliable, sequenced transmission in presence of exceptional[3] conditions should conform to the following specification:

(DR) *Delivery rule*: Under normal conditions, each *put* message is eventually delivered *exactly once* to the receiver via a *get*. Under exceptional conditions[4], each *put* message is delivered *at most once*.

(SR) *Sequencing rule*: Under any conditions, the order of messages delivered to the receiver by *get* is in accordance with the order they have been *put* by the sender. In conjunction with (DR) above, this implies that the sequence of received messages corresponds to the sequence of sent[5] ones, with arbitrarily many messages missing under exceptional conditions.

[3] Distinguishing between *normal conditions* and *exceptional conditions* is necessary since it is well-known that implementing reliable transmission in presence of exceptional circumstances like crashes is impossible (at least without stable storage), cf. [Bel76].

[4] *Under* exceptional conditions means *during and (some finite time) after* exceptional situations.

[5] There are of course applications where this assumption may be somewhat relaxed. However, a model without (SR) would not be applicable for fully sequenced traffic.

(AR) *Acknowledgement rule*: Under any conditions, *getack(ack)* is called at most once for each *put* message, acknowledging the messages exactly in the order as they have been *put*. If *ack*=OK, it is guaranteed that the corresponding message has been delivered to the receiver via *get*. Under normal[6] conditions, *getack* is eventually called and *ack* must be OK.

Note that this specification involves both *safety* (i.e., nothing bad ever happes) and *liveness* (i.e., something good eventually happens) properties. We will show that our SSCMP implements that specification on top of the following system model:

- We consider a distributed system consisting of multiple *nodes* connected by an *unreliable* network. In *normal situations*, there may be transient *packet losses*, *duplication*, *reordering*, and *late messages*.

  In *exceptional situations*, we also allow *network performance errors* to occur. A network performance error has occurred when a message (repeatedly) transmitted via a certain retransmission scheme during an interval D of real-time duration δ does not cause at least one acknowledgment to be delivered within D. Note that this is an end-to-end argument, covering both sender and receiver node *performance faults* and *non-transient packet losses* as arising in long-lasting network partitions. This is a fairly general assumption on message delivery times, since we do not assume bounds on the delivery of a single message (as usually employed), but only on the whole bulk of retransmitted ones; in fact, an acknowledgment message received within δ may be caused by the first transmission eventually arriving at the receiver and being acknowledged, however, with all the sender's retransmissions lost.

  We finally assume that corrupted packets are always detected (e.g., by checksumming methods) and that the network provides reliable service with low delay and high throughput most of the time.

- All nodes are equipped with mutually *synchronized clocks* of sufficiently fine granularity γ, that do not wrap-around during the lifetime of the whole system.[7] For example, NTP provides 64-bit timestamps with $\gamma = 236$ *ps* granularity that wrap around once every 136 years.

  In normal situations, we assume that the clock *S.time* of a node S progresses monotonically at (approximately) the rate of real-time. More specifically, for any interval duration Δ measured on S's clock, the corresponding real-time duration δ fulfills $\Delta(1-\alpha) \leq \delta \leq \Delta(1+\alpha)$ for some $0 \leq \alpha << 1$. In addition, the clock readings $S.time(t)$, $R.time(t)$ on any two nodes R, S at real-time t are within some maximum *skew* ϵ of each other, i.e., $|S.time(t) - R.time(t)| \leq \epsilon$.

  In exceptional situations, any kind of failure may occur: a clock's monotonicity may be violated, its rate (i.e., α) may deviate from approximate real-time, and the skew ϵ may be exceeded arbitrarily. Note however that a clock must eventually make progress; otherwise, we assume that its node has crashed.

[6] Actually, it is of course not difficult to ensure that *getack* is always called in exceptional situations not caused by sender faults.

[7] This is not an unreasonable and uncommon assumption. cf. [BF93] (but see also the concluding section of this paper).

- In exceptional situations, nodes may suffer from *crash faults* (*omission* and *performance faults* are already covered by the unreliable network above), but not from arbitrary (*byzantine*) ones[8].

- Each node is equipped with (limited) memory that survives node crashes, e.g., non-volatile memory or a stable storage service.

- Without loss of generality, we assume that there is only one (client-)process per node that communicates in a request/reply-fashion, i.e., basically unidirectionally[9], with a possibly large number of (server-)processes on other nodes during its lifetime. Each process has a *process identifier* that may be used for uniquely adressing a particular sender or receiver. Note that multiple data streams per process are easily provided by means of (statically) assigned *port numbers*, extending process identifiers as usual.

In addition, the implementation of the protocol should provide maximum performance even in presence of limited (memory) resources. In particular, we require:

- *Pipelining*, i.e., multiple outstanding (unacknowledged) messages. Using stop-and-wait techniques would intolerably limit network performance in case of networks with large bandwidth-delay products, and would also prohibit any speedup by means of network coprocessors.

- *No connection setup overhead* by the reasons introduced in Section 1.

- *Reuse of connection records*, i.e., no static allocation of dedicated memory holding the connection record for each server the client might ever connect to.

Note that the latter is particularily interesting when (usually limited) non-volatile memory is available, as in our particular real-time system application.

3 The Sequenced Synchronized Clock Message Protocol SSCMP

In [LSW91], a strikingly elegant and simple protocol providing (unsequenced) at-most-once message delivery has been presented. That *Synchronized Clock Message Protocol SCMP* surpasses existing timer-based protocols like *Delta-t* ([Wat81]), *VMTP* ([Che89]), *XTP* ([SDW92]), and *CMSC* ([BF93]) with respect to at-most-once delivery properties. Most notably, it relies on synchonized clocks only for performance but *not* for correctness and is end-to-end in that it does not require support from intermediate (gateway-)nodes. However, unlike the other protocols mentioned, SCMP does not provide for message sequencing.

One might of course think of extending SCMP by adding sequence numbers. However, it turns out that some clever way of integrating SCMP and sequence numbers is required. To describe how we accomplish this in our SSCMP, we have to introduce the original SCMP first.

[8] In particular, our (basic) protocol may of course be defeated by (deliberate) intrusions forging messages.

[9] Whereas bidirectional communication may always be formed by two independent unidirectional channels, our protocol may be adapted to support piggybacked acknowledgment techniques as well, since our acknowledgement messages require no special treatment and carry only few information.

3.1 The original SCMP

SCMP deals with messages m containing a connection identifier $m.conn$ and the time of creation of the message $m.ts$; we use the pair of process identifiers of sender S and receiver R, and the sender's time $S.time(t)$ of *put*-ing the message, respectively. All possible (retransmitted, duplicated, ...) instances of a message m will have the same connection identifier and timestamp.

Each server process (i.e., receiver R) maintains a *connection table* $R.CT$ containing connection information for all the different clients (i.e., sender). For SCMP, the only connection information of interest is the timestamp of the last message accepted from a particular client S, (symbolically) written as $R.CT[S].ts$. If a new message m from S arrives, it is accepted only when $m.ts > R.CT[S].ts$, clearly rejecting any duplicates.

However, with SCMP, there is no statically preallocated connection table entry for each possible sender. Such entries are *dynamically* allocated and only retained for a reasonably large period of ρ seconds (as measured on R's clock) after the last message has been received over that connection. That is, R is free to remove (and reuse) the entry for a sender S from $R.CT$ at some real-time t provided that $R.CT[S].ts \leq R.time(t) - \rho$.

However, to retain the ability to detect duplicates, some of the lost information is preserved by maintaining an upper bound $R.upper$ on *all* the timestamps (that is, from all the different connections) that have been removed from the table. Thus, if a new message m from a sender having no entry in $R.CT$ arrives, it may be safely accepted and a connection record created if $m.ts > R.upper$. Otherwise, it is not clear whether that message is actually a duplicate or not since the connection information required for deciding that question has been discarded; hence the message must be rejected. However, if ρ is reasonably large, the probability of incorrectly rejecting an acceptable message is quite small since $m.ts \leq R.upper \leq R.time(t) - \rho$ shows that m must be very late.

Maintaining $R.upper$ is of course trivial for normal receiver operation. However, when at-most-once delivery is to be preserved even across receiver crashes, a way of safely reinitializing the connection table after a crash is needed. There are at least two possibilities, with decreasing quality:

1. *Sufficient non-volatile memory* is available for keeping the conncetion table and $R.upper$ in it. Nothing will be lost here since all the required information survives a crash of R. Note however that one should delay any connection record release —advancing $R.upper$— after recovery for some reasonable period of time to allow any old message to be accepted.

2. Only *few non-volatile memory* or a (slow) *stable storage service* is available for maintaining a stable upper bound $R.latest$ enforced on the (timestamps of) *all* the messages accepted up to real-time t (over any connection). That is, the algorithm ensures that up to time t no message m has been accepted with timestamp $m.ts > R.latest$, where $R.latest$ is periodically updated (at real-time t_u) to $R.time(t_u) + \beta$ for some fixed β; monotonicity of $R.latest$ must of course be maintained. β should be chosen small enough to prevent actual upper-bound enforcement (i.e., delaying too early messages) from occurring too frequently.

   After a crash, $R.latest$ is used to initialize $R.upper$; the connection table is (re)initialized to being empty. This initialization is feasible since it is guaranteed that no message m with $m.ts > R.latest$ has been accepted before the crash. Note that only messages with

timestamps older than or equal to *R.latest* are possibly (unnecessarily) rejected after that recovery; messages generated during long crash periods are usually not concerned since $R.time(t_{\text{crash}}) \in [R.latest - \beta, R.latest]$.

For further details on SCMP, in particular a proof of its correctness and a discussion of suitable parameter values (ρ and β), the interested reader is referred to [LSW91].

It is clear that the original SCMP is not concerned about sequencing even under normal conditions: If two messages are transmitted and the first one gets lost due to a transient error, the (usually) successful reception of the second message will cause the retransmitted first one to be rejected (since its timestamp is obviously older). Simply adding sequence numbers does not help, and SCMP's discarding and reestablishing of connection information makes such approaches even more hopeless.

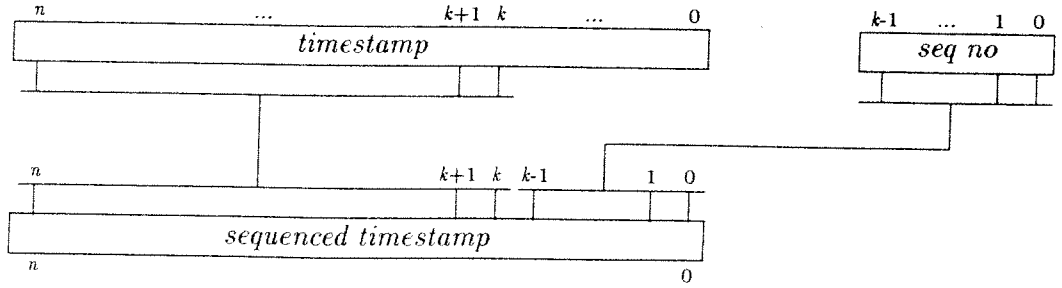
However, if we could find a clever way of encoding both message timestamps and sequence numbers in a *sequenced timestamp ST* ensuring that

- a ST may be interpreted uniquely as a timestamp,
- a ST may be interpreted uniquely as a sequence number,

then it should be possible to define a protocol that switches between SCMP and a suitable *sliding window protocol* without sacrificing the at-most-once delivery property of the former and the sequencing property of the latter. Note that this is of course not the same as using timestamps and sequence numbers orthogonally.

More specifically, for a new connection, the SCMP scheme may be used to decide whether a message may safely be accepted and a (properly initialized) connection record created; this is done by intepreting the message's ST as a timestamp. For subsequent messages, the state information in the connection record is available for providing sequenced communication via the sequence number interpretation of STs. Finally, the last accepted sequence number (i.e., ST) interpreted as a timestamp may be used to update *R.upper* when the connection record is released after a (silent) period of duration ρ.

Our *Sequenced Synchronized Clock Message Protocol SSCMP* is based on that idea. The encoding of sequenced timestamps used is strikingly simple (once it is discovered): *We use the last few bits of the timestamp to encode the sequence number.* For example, in case of n-bit timestamps and k-bit sequence numbers, an n-bit sequenced timestamp is constructed as follows:



Note that clock granularities are usually fine enough to guarantee unique timestamps for successively generated messages even with a few low-order bits lacking; for example, we mentioned already that the 64-bit timestamps of NTP have 236 *picoseconds* resolution!

With that encoding, the first conversion (ST $\rightarrow$ timestamp) mentioned above is (almost) a null-operation since we are dealing with sequencing in time. Actually, sequenced timestamps are automatically monotonic except at the instant where the sequence number part wraps around.[10] There are at least two possibilities how to cope with that problem:

- We enforce total monotonicity of sequenced timestamps at the sender, by slightly delaying the assignment of the ST of a message that would be less or equal to the ST of its predecessor due to wrap-around. With that method, the whole ST may be used as a timestamp (conversion is a null-operation), which reduces the number of unneccessarily rejected messages after certain exceptional situations like a network partition, cf. the next item. This approach is even (and in particular) useful when the granularities γ are relatively large. Finally note that the modified SSCMP introduced in Section 4 (Theorem 1) implicitely provides total monotonicity of sequenced timestamps by means of this method.

- We use only the timestamp portion for SCMP-related purposes, filling up the sequence number portion with all zeroes (or all ones when assigning to $R.upper$). This is feasible since the SCMP-related scheme takes over only after a silent period of duration ρ under normal conditions, of course amply fulfilling any monotonicity requirement. However, after an exceptional situation like a network partition, messages with a sequence number higher than the last accepted one (but with the same timestamp portion) are unnecessarily rejected. This deficit becomes of course less important when the granularities γ are small.

The second conversion (ST $\rightarrow$ sequence number) has to extract the sequence number portion only. Note however that the timestamp portion is also required for detecting late messages; obviously, if the second of the previously mentioned conversion functions is used, it must be ensured that the sequence number space is large enough to prevent wrapping around within the granularity γ.

This idea in fact provides a suitable basis for a whole family of protocols: Any sliding window or credit based protocol (without and with NACK's, go-back-n or selective repeat, cf. [Tan81, SDW92]) may be adopted[11] to fit into our SSCMP-framework. Fortunately, we may safely hide almost all the details of the underlying sliding window algorithm in our subsequent presentation, thus showing the essentials more clearly.

3.2 The SSCMP

SSCMP relies on messages m consisting of a *message header* and an arbitrary *message body*. The header contains

- a *connection identifier* $m.conn$ uniquely specifying sender S and receiver R,

- a *sequenced timestamp* $m.st$ consisting of a timestamp and a sequence number portion as specified above,

[10] We are grateful to an anonymous referee for drawing our attention to that point.

[11] We should mention that it has been observed in [Sha91] that connection management and data transfer protocols are orthogonal, so our approach (re)confirms this observation in this special case.

- an additional *initialization field* *m.init* capable of holding a sequence number (without timestamp).

All (retansmitted, duplicated, ...) instances of a message have the same header. An acknowledgement message a does not require any special treatment; it contains $m.conn$ and $m.st$ of the message m it acknowledges.

Our SSCMP is built upon (1) any of the two variants of the *original SCMP* mentioned in the previous subsection and (2) an arbitrary (but correct) *sliding window protocol* adapted to our encoding scheme. SCMP views $m.st$ as a timestamp (formerly $m.ts$) and incorporates the usual $R.upper$, possibly $R.latest$, a *receiver connection table* $R.RCT$ (formerly $R.CT$), and an additional *sender connection table* $S.SCT$.

Naturally, organizing the connection tables as arrays (indexed by R and S, respectively) would amount to (pre)allocation of memory, something we wanted to avoid in our implementation (remember the end of Section 2). Hence, some dynamic memory management in conjunction with a suitable mapping —hashing— function must be utilized. Note that the mapping function must be evaluated every time a message (or an acknowledgement message, respectively) is received, cf. Section 6. Memory is requested dynamically when a new connection is established and released after an interval of duration ρ without transmission activities. In the following section, we will establish a lower bound for ρ (Theorems 3 and 4). There is no explicit upper bound for ρ, but choosing it too large wastes memory for idle connections, cf. [LSW91] for a more thorough discussion.

The sender S's connection record in the receiver node's RCT contains:

- $R.RCT[S].expire$ (or *expire* for short, if the context is clear), the time when the receiver's state record for sender S is to be released.

- $R.RCT[S].lst$ (or *lst* for short), the sequenced timestamp of the last message accepted (i.e., delivered) from sender S.

Note that *expire* is not required in the original SCMP since $lst + \rho$ determines the expiration time; it need not be utilized in SSCMP either, but it helps improving the protocols performance in (rare) exceptional situations and is also necessary for some protocol extensions, cf. Section 6.

The receiver R's connection record in the sender node's SCT contains:

- $S.SCT[R].expire$ (or *expire* for short, if the context is clear), the time when the sender's connection record for receiver R may be released.

The sliding window protocol utilized should provide the following functions at the sender-side (acting upon a private *sender state record* included in $S.SCT[R]$:

- *init.slw.s()* appropriately initializes the sender's state record for receiver R; in particular, the initial sequence number is set (to 0).

- *put.slw.s(m)* forwards a message m to the sliding window protocol for transmission; the timestamp portion of $m.st$ must have been appropriately set, with the sequence number portion initialized to zero (by a logical *and* with an appropriate bit-mask SEQM). Note that we assume that *put.slw.s* is blocked (disabled) when the protocol cannot handle further messages. Before actually transmitting a message m over the channel, the field $m.init$ is set by *put.slw.s* to the sequence number of the first outstanding message (that is, the lower edge of the transmit window).

- *gotack.slw.s(ack.st)* handles an acknowledgement *ack* for message m with $m.st = st$ received over the channel. It is responsible for stopping retransmission activities, calling *getack(ack)*, and advancing the transmit window iff

  (GC) st equals $m.st$ of the oldest (i.e., latest *put*) outstanding message m.

  Note that acknowledgments for messages *put* later than the first outstanding one may of course be buffered internally (causing retransmissions to stop), so that the arrival of the acknowledgment for the first outstanding message might trigger multiple *getack*'s and transmit window advances.

Similarily, we assume the following functions at the receiver-side of the sliding window protocol (acting upon a private *receiver state record* contained in the sender S's entry *R.RCT[S]* in the receiver's *R.RCT*):

- *init.slw.r(m)* appropriately initializes the receiver R's state record for sender S. Note that this requires some additional information, in particular, the (next) expected sequence number, which is obtained from the additional initialization field *m.init* in the (initial) message m; we will elaborate on that topic later on.

- *put.slw.r(m)* is used to submit a message received over the channel to the sliding window protocol. It is responsible for eventually calling *get(m)* and sending back acknowledgments over the channel. More specifically, a message m from sender S is accepted iff

  (AC1) *m.st* is larger than *R.RCT[S].lst*, which must be maintained by *put.slw.r* to hold the timestamp *l.st* of the last accepted message l,

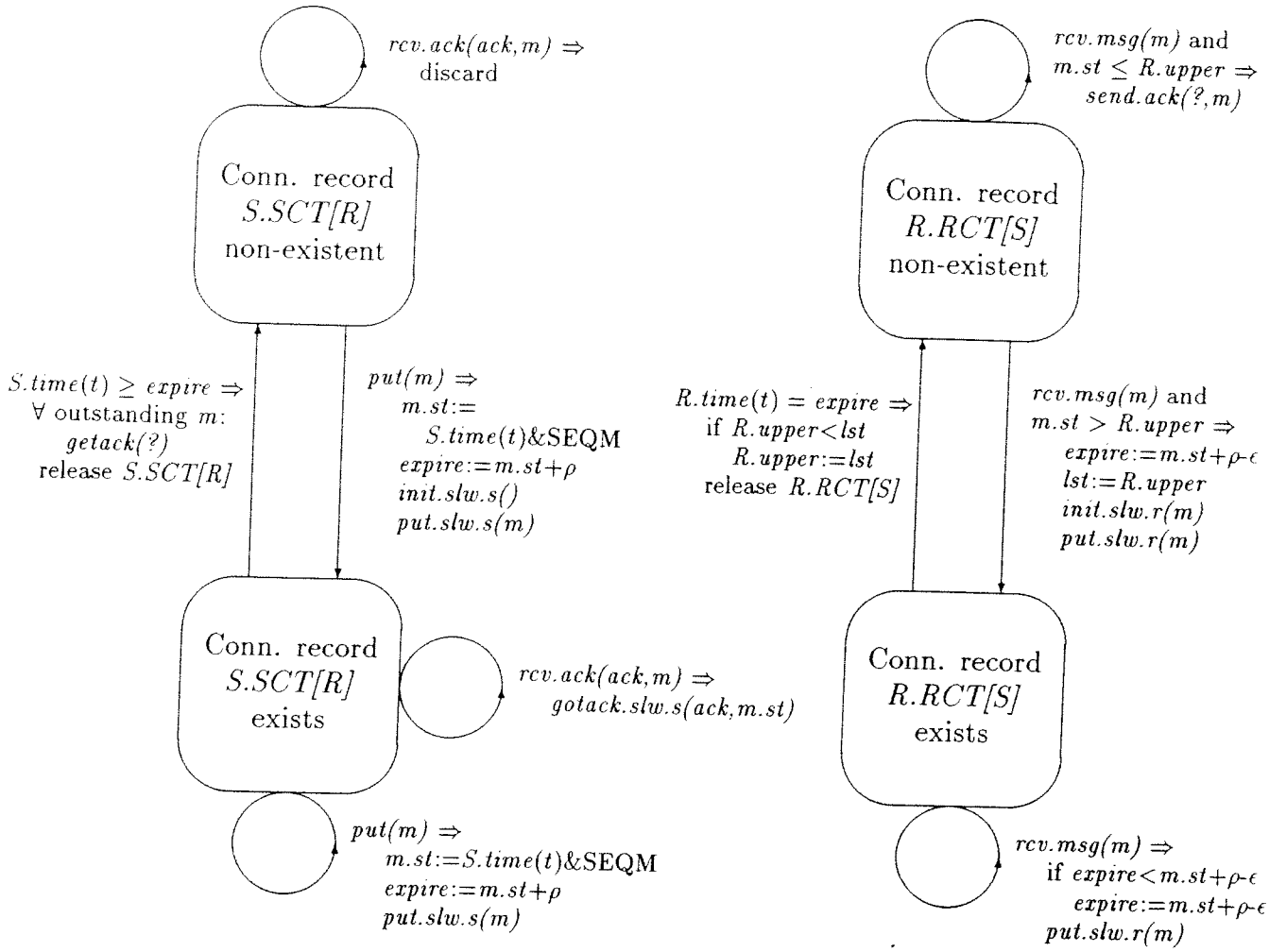
  (AC2) the sequence number portion of *m.st* equals the next expected sequence number.

  However, messages arriving out of sequence at the receiver may of course be buffered internally for performance purposes.

We finally assume that the implementation of the sliding window protocol is built upon the basic communication functions *send.msg(m)* and *rcv.msg(m)* for (unreliably) sending/receiving a message m over the channel. Similarily, *send.ack(ack,m)* and *rcv.ack(ack,m)* are provided for (unreliably) sending/receiving an acknowledgment *ack* for a message m.

Now we are ready to provide the following (reasonably high-level) state-machine representation of the sender's and receiver's part of SSCMP[12]. Actions in our protocol are executed in response of certain triggering events like calling *put(m)*; this is modelled by state transitions labelled with constructs of the form *event* $\Rightarrow$ *actions*; multiple actions are executed atomically, as usual.

[12]Actually, we present only a slightly simplified version of our actual implementation of SSCMP here, and discuss some extensions later on. In particular, we assume that sufficient non-volatile memory is available for keeping the connection tables and *R.upper* in it, assume totally monotonic sequenced timestamps, and omit some add-ons for improving the protocol's performance under exceptional conditions.



Sender S **Receiver R**

As depicted above, both sender S and receiver R engaged in a particular connection may or may not have an existing connection record. More specifically, the connection record in the sender connection table *S.SCT[R]* is created when the first *put(m)* of a message to R occurs — either the very first one or the first one after a long period ($\geq \rho$) of no transmission activities.

Subsequently, the sender uses its sliding window protocol for sequenced data transmission of *put* messages, generating *getack*'s as (valid) acknowledgment messages are received via *rcv.ack(ack,m)*. Note that the original message m acknowledged by *ack* is uniquely identified by its sequenced timestamp *m.st*; even late messages —defeating ordinary sliding window protocols— do not cause any problems. With each newly *put* message m, the expiration time of the sender's connection record *S.SCT[R]* is set forward to $m.st + \rho$.

When no messages have been *put* for a period larger than ρ (as measured on S's clock), then the connection record *S.SCT[R]* may be released. If there are still outstanding messages (due to a long lasting network partition, for example), they are acknowledged with ? and all (re)transmission activities are cancelled. The sender finally reverts to the state where no connection record for R exists: subsequently arriving acknowledgment messages are of course discarded.

For the receiver, the situation is quite similar. If there is no connection record *R.RCT[S]* for a sender *S* in *R*'s receiver connection table, incoming messages are checked against *R.upper* according to the original SCMP protocol. If an acceptable message *m* (with $m.st > R.upper$) arrives, a connection record *R.RCT[S]* is created and appropriately initialized. In particular, its expiration time *R.RCT[S].expire* is set to the sender's one (i.e., $m.st + \rho$) minus the maximum clock skew ϵ, and the timestamp of the last accepted message *lst* is (safely) initialized to *R.upper*.

Moreover, the (next) expected sequence number internally required by the sliding window protocol is initialized to *m.init* by *init.slw.r*; remember that this field contains the sequence number of the sender's first outstanding message. We will show in the next section that this initialization is indeed feasible provided ρ is chosen large enough. Note that it is also necessary[13]; for example, naively initializing the expected sequence number to the sequence number portion of *m.st* would cause the sequencing problem encountered with the original SCMP again!

Subsequently received messages are forwarded to the sliding window protocol via *put.slw.r*; the expiration time *expire* is also set forward when necessary. If no (valid) message is received up to the expiration time *R.RCT[S].expire* (as measured on *R*'s clock), the receiver *must* instantaneously release its connection record. We will show in the next section that our particular setting of $R.RCT[S].expire = m.st + \rho - \epsilon$ guarantees that —under normal conditions— the receiver's connection record is *always* released earlier or simultaneously as the sender's, cf. Theorem 3, Item (3.b.ii) in the following section.

Strictly speaking, what is actually needed for that instantaneous release is enforcing that the receiver reinitializes its sliding window protocol via *init.slw.r* before processing a message *m* at real-time *t* with $R.time(t) \geq R.RCT[S].expire$. Updating *R.upper* and releasing the connection record may of course take place at any time t_r with $R.time(t_r) \geq R.RCT[S].expire$, as in the sender. This behaviour is easily provided by introducing an intermediate state in the receiver's state machine above, representing an expired but not yet released connection record; we decided not to incorporate this detail into the figure above for the sake of simplicity.

4 Analysis of the SSCMP

The primary purpose of this section is to show that (a slightly modified version of) SSCMP implements the specification contained in Section 2. Our proofs usually involve two seperate parts: conformance under exceptional conditions and conformance under normal ones. In the former case, we have to deal with *safety properties* only, whereas in the latter we have to consider *liveness properties* as well. We start with the exceptional conditions for the *delivery rule* (DR).

Lemma 1 *Under any conditions, SSCMP guarantees that messages put by the sender are received via get at the receiver at most once.*

Proof: Following the line of reasoning in [LSW91], we consider a particular receiver *R* and its connections to multiple senders maintained via the receiver connection table *R.RCT*. From the description of the SSCMP algorithm in the previous section, we know that the variable *R.RCT[S].lst* is assigned *l.st* of the last accepted (i.e., delivered via *get*) message *l*, cf. (AC1) of Section 2. Using the abbreviation $lst^S = R.RCT[S].lst$ (or just *lst* when the sender is obvious

[13]However, there are other possible initialization methods as well; in particular, it is possible to employ a *syn/fin* flag and (only occasional) handshaking as in TCP (see [Pos81]) for that purpose.

from the context), let $\{lst_i^S\}_{i\geq 1}$ denote the sequence of those[14] assignment values and $\{t_i^S\}_{i\geq 1}$ the assignment (real-)times, taken for *all* the (subsequent) connections between S and R.

It is not hard to establish the following most useful property of the sequence introduced above:

Lemma 2 *For any sender S, the sequence $\{lst_i^S\}_{i\geq 1}$ is strictly monotonically increasing.*

Proof: Let t be any real-time within the interval $t_{i-1}^S \leq t < t_i^S$; hence, $i-1$ is the index of the last assignment lst_{i-1}^S that took place before or at t. At time t_i^S, there are two possibilites regarding the previously used connection record *R.RCT[S]*: (1) it still exists, or (2) it has already expired. In the former case (1), we know from (AC1) in Section 2 that a new message m is only accepted if $m.st > lst_{i-1}^S$. Hence, $lst_i^S = m.st > lst_{i-1}^S$ and we are done.

In case (2), we know that a new state record is to be created or may have been already ("improperly") created; the latter situation arises in case of two (new) messages m, m' successively transmitted by the sender, the first of which got lost. When m' arrives, *R.RCT[S]* is created and *lst* is initialized to *R.upper* so that this value is available when (the retransmitted) m eventually arrives. Anyway, if the connection record is created at time $t \leq t_i^S$, it is clear from the protocol's execution that

$$lst_i^S > R.upper \tag{1}$$

at time t. So what we need to complete the proof of Lemma 2 is the following lemma:

Lemma 3 *Let t be any real-time where the connection record R.RCT[S] for a sender S does not exist. Let $j-1$ be the index of the last assignment lst_{j-1}^S to lst^S determined by $t_{j-1}^S \leq t < t_j^S$. Then, R.upper is monotonically increasing and fulfills $R.upper \geq lst_{j-1}^S$ at time t.*

Proof: First, the statement above follows trivially when there was no crash of the receiver between t_{j-1}^S and t: According to the original SCMP, *R.upper* is always set to the maximum of its former value and the maximum of the values lst^s of all connection records for senders $s \in \mathcal{S}$ which are to be released. Since S is among the set of all released ones, we are finished here.

When there was a crash of R, we have to distinguish the two different SCMP-variants; cf. the description of SCMP in Subsection 3.1. If the receiver connection table *R.RCT* and *R.upper* are kept in non-volatile memory, a crash does not cause any difficulties here. If the method employing a stable *R.latest* is used, we must show that the reinitialization of *R.upper* after a crash does not violate the statement of our lemma. This, however, is trivial since it is enforced that no message m with $m.st > R.latest$ has been accepted prior to the crash. Hence, we have

$$R.upper_{\text{postcrash}} = R.latest > R.upper_{\text{precrash}} \geq lst_{j-1}^S,$$

completing the proof of Lemma 3.□

[14]There are also (initial) assignments $R.RCT[S].lst := R.upper$ when initializing a connection record, which we do not take into account.

Applying Lemma 3 to equation (1) also finishes the proof of Lemma 2.□

By virtue of Lemma 2, however, the statement of Lemma 1 follows immediately: Since the sequenced timestamps $m_i.st = lst_i^S$ of the messages $\{m_i\}_{i\geq 1}$ delivered to the receiver via *get* are strictly monotonically increasing, no duplicate message is ever delivered. This eventually completes the proof of Lemma 1.□

The latter observation also provides the key to the exceptional conditions part of the *sequencing rule* (SR): It is immediately apparent that sequencing is guaranteed by SSCMP given that the sequenced timestamps of the messages *put* are strictly monotonically increasing. However, in case of a clock synchronization fault, that precondition might be violated. Therefore, we must enforce monotonicity of the sequenced timestamps by a slight modification of the sender's procedure for assigning *m.st*.

This may be accomplished by a clock-reading procedure that enforces strong clock monotonicity by internally keeping the last value read, delaying the current read access if it is made (seemingly) too early. Note that this provision may be adopted to provide total monotonic sequenced timestamps, cf. the end of Section 3.1, and even allows to detect certain clock anomalies. However, to preserve timestamp monotonicity across crashes, additional measures must be provided: Either, the last value read is kept in non-volatile memory or, alternatively, a stable upper bound *S.usedlatest* is enforced on the timestamps used for *m.st* of *any* message *put* at the sender node *S*. The latter method relies on the same technique as used for *R.latest* in the original SCMP (cf. Subsection 3.1); it is particularily interesting since it allows to detect (faulty) clocks that jumped forward unreasonably.

We will refer to the SSCMP enforcing monotonic sequenced timestamps as the *modified SSCMP*. Thus, we immediately obtain the following theorem:

Theorem 1 *Under any conditions, the modified SSCMP guarantees sequenced at-most-once delivery.*□

The exceptional conditions part of the *acknowledgment rule* (AR) is covered by the following theorem:

Theorem 2 *Under any conditions, SSCMP guarantees that put messages are acknowledged by getack(ack) at most once and in exactly the same sequence as they have been put. If ack = OK, the corresponding message has been delivered to the receiver via get.*

Proof: First, since the sender executes both *put* and *getack* locally, sequenced at-most-once acknowledgments are guaranteed by virtue of condition (GC) on *gotack.slw.s*, cf. Subsection 3.2. The second statement of the theorem is trivial since a *getack* with *ack* = OK for some message *m* does only occur when an acknowledgment message with *ack* = OK has been received for message *m*. Such an acknowledgment message, however, is only produced by the receiver when *get* has been called, cf. *put.slw.r*. Hence, the proof of Theorem 2 is completed.□

Note that the at-most-once acknowledgment guaranteed by Theorem 2 means that *put* messages are *getack*'ed in exactly the same order as they have been put. However, it may happen that the sequence of acknowledgments is finite (when the protocol gets stuck under exceptional conditions), that is, liveness is not guaranteed.

Now we turn our attention to SSCMP's operation under normal conditions, which involves liveness properties as well. We start with the most important *delivery rule* (DR):

Theorem 3 *Under normal conditions, the modified SSCMP with ρ large enough to satisfy the inequality $(\rho - 2\epsilon)(1 - \alpha) > 2\delta$ (cf. Section 2), each message put is eventually delivered to the receiver exactly once and in the same sequence as they have been put.*

Proof: Since Theorem 2 holds under any conditions, it suffices to show that each message is eventually delivered to the receiver. More specifically, we will prove that each message is delivered (in fact, even acknowledged) within δ (real-time) seconds.

Let us assume the contrary, i.e., that there is some (first) message m from sender S that is not delivered to the receiver via *get* within δ. Using case analysis, we show that this yields a contradiction in any case.

Let $j-1$ be the index of lst^S_{j-1} containing $l.st$ of the last accepted message l, cf. the proof of Lemma 1. There are only three exhaustive possibilities why m might not be delivered:

1. m does not arrive at the receiver within δ.

   However, since the clocksynchronization condition must hold, m is retransmitted by the sender during an interval of (real-time) duration at least $\rho(1-\alpha)$ with $0 \leq \alpha << 1$, cf. Section 2. Since $\rho(1-\alpha) \geq (\rho - 2\epsilon)(1-\alpha) > 2\delta > \delta$ according to our precondition on ρ, this is only possible when a network performance error has occurred. This violates the normal conditions assumption.

2. m successfully arrives within δ at the receiver which has no connection record for sender S, but is rejected.

   This is only possible if $m.st \leq R.upper$. However, we have the following simple lemma:

   Lemma 4 *Under normal conditions, at any real-time t, we always have*

   $$R.upper \leq S.time(t) - \rho + 2\epsilon.$$

   Proof: Clearly, if $t_r \leq t$ denotes the time of the most recent connection record release at the receiver node before or at t, we have

   $$\begin{array}{rcll} R.upper & \leq & R.time(t_r) - \rho + \epsilon & \text{(by the protocol)} \\ & \leq & R.time(t) - \rho + \epsilon & \text{(by clock monotonicity)} \\ & \leq & S.time(t) - \rho + 2\epsilon & \text{(by clock skew)} \end{array}$$

   and the lemma follows.□

   Hence we find that $m.st \leq S.time(t) - \rho + 2\epsilon$ at the time t of reception of m. This implies that S must have retransmitted m during an interval of duration at least $D = \rho - 2\epsilon$ on its clock. Since this corresponds to a real-time interval of duration $d \geq (\rho - 2\epsilon)(1-\alpha) > 2\delta > \delta$ according to our precondition on ρ, a network partition error must have occurred. This violates our assumption of normal conditions and provides the required contradiction.

3. m successfully arrives within δ at the receiver which has a connection record for sender S, but is rejected. Here we have to distinguish two more cases:

   (a) An "improperly" initialized connection record *R.RCT[S]* exists at the time of reception of m.

As already mentioned in the proof of Lemma 2, this may happen in case of an out-of-sequence message m' (*put* later than m but arriving earlier). At the time t' of reception of m', *lst* has been initialized to *R.upper* (to keep this value available for the time the retransmitted message m eventually arrives), and the expected sequence number in *R.RCT[S]* has been set to *m'.init*. Note however, that no message has been accepted along that connection yet.

Now, rejection of m may only occur in the following two exhausive cases:

(i) $m.st \leq lst$; however, due to the initialization of *lst* to *R.upper*, case 2 above applies here.

(ii) $m.st > lst$ but does not have the right (i.e., the expected) sequence number.
To rule out that possibility, we must show that the initialization of the next expected sequence number to *m'.init* is feasible, i.e., that *m'.init* equals the sequence number portion of *m.st* and not that of an earlier one; note that we even allow $m = m'$ here. Anyway, m' must of course have passed the SCMP-check successfully.
Let t_S^l denote the real-time when the last accepted (i.e., acknowledged) message l has been *put* at the sender S; of course, $S.time(t_S^l) = l.st$. Consequently, the connection record at the receiver *R.RCT[S]* must have expired when R's clock read $l.st + \rho - \epsilon$. Denoting the real-time when m' arrived at the (already expired) receiver by t', it follows that $R.time(t') \geq l.st+\rho-\epsilon$ due to monotonicity. Because of the clock skew, we thus obtain $S.time(t') \geq l.st + \rho - 2\epsilon$ and hence

$$t' - t_S^l \geq (\rho - 2\epsilon)(1 - \alpha) > 2\delta.$$

But now, since l has been successfully delivered, an acknowledment must have arrived at the sender at $t_a^l \leq t_S^l + \delta$. Similarily, m' must have been sent at $t'_S \geq t' - \delta$, because otherwise a network performance error would have occurred for m' (and hence for m). We therefore find

$$t'_S - t_a^l \geq t' - \delta - t_S^l - \delta > 0,$$

finally establishing that *m'.init* must contain the sequence number of m.
This fact provides the required contradiction for this case.

(b) a "properly" initialized connection record for sender S exists but m is rejected.
Here, at least one message has been accepted over that connection; let l be the last one; clearly, $l.st = lst_{j-1}^S$. Again, we have only two exhaustive possibilities why rejection might occur:

(i) $m.st \leq lst_{j-1}^S$; however, this is impossible since m is neither a duplicated nor a late message: According to the modified SSCMP, *m.st* must be larger than *l.st* of the last acceped message, providing the necessary contradiction.

(ii) $m.st > lst_{j-1}^S$ but the sequence number of m is not the expected one.
It is obvious that rejection in this case is impossible if the sender still uses the same instance of its connection record *S.SCT[R]* as used for transmitting l, since the ordinary sliding window protocol is employed in this case (remember that the corresponding *R.RCT[S]* exists).
Thus, we only have to consider the case where S has established a new instance of a connection record but R uses its old one. Hence, the sender's expiration

must have taken place earlier than the receiver's. However, it is easy to show that (under normal conditions) the receiver's connection record *R.RCT[S]* for a particular instance of a connection between R and S expires earlier or at the same time as the sender's *S.SCT[R]*: Since

$$R.RCT[S].expire := m.st - \epsilon = S.SCT[R].expire - \epsilon,$$

the actual expiration real-times must fulfill $t_S \geq t_R$; note however, that enforcing the receiver's expiration (cf. the description of SSCMP in Subsection 3.2) is vital here — otherwise, the protocol might block or even accept messages out of sequence.
Thus, we have provided the required contradiction for this (last) case too.

This eventually completes the proof of Theorem 3.□

What remains to be done is to prove the normal conditions part of the *acknowledgment rule* (AR):

Theorem 4 *Under normal conditions, the modified SSCMP with ρ large enough to satisfy the inequality $(\rho - 2\epsilon)(1 - \alpha) > 2\delta$ guarantees that each message is acknowledged by getack with ack = OK exactly once and in the same sequence as the messages have been put.*

Proof: Since all messages *put* are delivered according to Theorem 3, acknowledment messages for all of them are generated. Since there must not be a network performance error, that acknowledgments must arrive at the sender within δ (real-time) seconds from the time the messages have been *put*. In order to prove that *getack*'s are eventually generated by *gotack.slw.s* according to (GC), we only have to show that the connection record *S.SCT[R]* still exists at that time. This, however, is amply fulfilled since the sender maintains its state for at least $\rho(1 - \alpha) > 2\delta > \delta$ (real-time) seconds. Theorem 2 eventually establishes that those *getack*'s occur in the right sequence. □

This finishes our analysis of the properties of SSCMP: Theorems 1–4 show that it indeed implements the specification of Section 2 on top of our system model. Those results confirm theoretically the encouraging experiences we gathered from our actual implementations. More specifically, it has been the major responsibility of the second author to provide a *C*-implementation of SSCMP —including a suitable simulation environment— for experimental evaluation and improvement of our protocol. We have been playing around with certain varieties e.g. of the underlying sliding window protocol for some time; the current version employs an elaborate selective repeat scheme with negative acknowlegments. However, we are still working on some extensions (see Section 6).

5 Comparison to Other Protocols

In this section, we relate our protocol to the existing timer-based ones we are aware of: *Delta-t*, *VMTP*, *XTP*, and *CMSC*. The first thing to mention is that our SSCMP —unlike most of the other ones— is not a fully engineered connection management protocol in the sense that it does not deal with addressing, rate control and dozens of other "practical" issues. However, since we are only interested in the "core" of the protocols, we think that this comparison is nevertheless

appropriate. Although we tried to find a common level of description in order to make the comparison meaningful, it is not possible to give more than a rather informal quantification of characteristics like connection record size and protocol complexity (i.e., performance).

For each protocol, we give a short description of the features meaningful for our purposes, and summarize some characteristics in the table below. Finally, we briefly discuss how SSCMP relates to those protocols.

The pioneering *Delta-t* ([Wat81]), the first timer-based connection management protocol available, is a fully engineered transport protocol designed for system architectures without special hardware like synchronized clocks and stable storage. Connection records are created "on demand" and become automatically (i.e., timer-based) released when it is guaranteed that no old packet is alive. This is made working by incorporating a *time-to-live* field into each message sent, which is appropriately decremented as the message travels through the network and intermediate nodes. To that end, some (reliable) link-transit-time protocol is required; the one presented in [Slo83] assumes that all node's clocks are running at approximately the same rate (although Delta-t does not need synchronized clocks). The receiver node retains a connection record just long enough to guarantee that any duplicate of a message originated during the lifetime of a connection has its time-to-live expired, exploiting the fact that any node that encounters a message with zero time-to-live must discard it.

As far as the characteristics given in the table below are concerned, we first note that in case of clock rate faults or underestimated link-transit-times, Delta-t may fail since a retransmitted message may arrive after the previous connection record has been discarded. For each active connection, there are three[15] timers required: two for the sender's part (connection record release and retransmissions) and one for the receiver's connection record release. The connection record size of Delta-t is reasonably small since there are only the usual connection identifiers and some sequencing information to be stored (letting aside engineering add-ons, of course). Relating the connection release times provided in [Wat81] to our framework, we find that the release time T_R of the receiver's connection record fulfills $T_R \approx 2\delta$: The sender's connection record is established for a period of approximately δ following the first transmission of a new message, and Delta-t demands (cf. condition C1 in [Wat81]) that all messages originated during that interval must find the receiver stateful. Since there might be a new message sent at the end of the sender's interval, which may take as long as δ to reach the receiver, our assertion follows. Moreover, connection setup must be delayed for some similar time after rebooting a previously crashed node. Finally, we argue that the protocol is not really simple due to the somewhat complicated link transit time protocols needed.

Another fully engineered transport protocol relating to our SSCMP is *VMTP* described in [Che86], which employs a timer-based connection management scheme very similar to the one used in Delta-t. However, it exploits properties following from introducing T-stable addressing for duplication detection purposes also. In its original version, VMTP relied on a time-to-live field incorporated in each message; in its revised version (cf. [Che89]), however, it employs end-to-end timestamps in order to enforce maximum packet lifetimes. Hence, it requires synchronized clocks and also some sort of stable storage in order to generate T-stable identifiers valid even across node crashes.

Similar to Delta-t, VMPT may fail if the clock synchronization condition is violated since packet lifetime enforcement solely depends on that condition. For each active connection, there

[15] Actually, it is possible to replace some of the per-connection-timers by a single one for all connections, e.g., the sender and receiver connection record release timers.

are again two timers for the sender (connection record release and retransmissions/probes) and one for the receiver. Restricting VMTP to the bare essentials, the connection record size should be quite small since there are only the usual entity identifiers and a single transaction identifier (corresponding to a sequence number) to be stored. Adopting the analysis of VMTPs connection release times given in [Che86] to our framework, we find that, for the receiver's T_R, $T_{\text{reliable}} = \delta$ and also $T_{\text{retrans}} + T_{\text{packet}} \approx \delta$ so that $T_R > 2\delta + T_{\text{probe}}$, where $T_{\text{probe}} > 0$ denotes the time used for (periodic) verification of the validity of the T-stable entity identifiers of the parties involved in the connection. Note also that the sender's connection record should reasonably be released not earlier than $T_R + 2T_{\text{packet}}$ after the last (initial) transmission. Moreover, if no stable storage is used (in case of stable identifiers, cf. [Che86]), connection establishment must be delayed for some time following the reboot of a previously crashed node. Finally, we think that VMTP is not really simple since there is some overhead involved in establishing T-stable identifiers.

A particularily carefully engineered and versatile "next generation" transport protocol is *XTP* described in [SDW92]. However, as far as timer-based connection management is concerned, it does not provide novel ideas but relies on the mechanisms of Delta-t. Nevertheless, we should note that XTP actually uses a certain mixture of timer-based and handshake-based mechanisms, supporting a quick graceful close and hence speeding up connection release time. Note that the usual disadvantage of timer-based protocols with respect to handshake-based ones is the fact that a connection record is usually released later; nevertheless, even completely handshake-based protocols like TCP require some non-zero connection release time, cf. [Wat81]. Anyway, we do not want to characterize such techniques in this paper.

The *CMSC* protocol described in [BF93] follows the approach underlying the revised VMTP ([Che89]) to remove the dependence of the protocol from the underlying network. More specifically, an expiration time (instead of a time-to-live field) is added to each message, making the maximum packet lifetime enforcement purely end-to-end by means of synchronized clocks. Actual message transmission is governed by an ordinary sliding window protocol. Although CMSC is not a fully engineered protocol in the sense of Delta-t, VMTP, or XTP, it deals with a connection oriented interface in some detail.

Reviewing the properties of CMSC, the first thing to note is that it may fail in case of clock synchronization violations for the same reasons as Delta-t and VMTP. Moreover, since CMSC employs expiration time and sequence numbers in an "orthogonal" manner (and not in an integrated way as we do in SSCMP), a rather complicated timer-management —involving five "logical" timers per connection (which may, however, be implemented by three physical ones)— is necessary to make sequence number reuse safe. The connection record is also slighly larger as the one of other protocols since explicit lifetimes are to be maintained. As far as the connection record release times are concerned, adopting the analysis of [BF93] to our framework shows that $LT = \delta$ so that the receiver's connection record must be retained for as much as 2δ: The controlling timer T_{r1} is set approximately to the most recently received message's expiration time (= time of transmission + LT = the sender's connection record release time) plus LT, that is, to $2LT$ in the worst case. Using stable storage, CMSC allows immediate resumption following an (fail-stop) endsystem failure. Finally, we argue that CMSC is not at all simple; at least, it takes several pages of pseudo-code for its algorithmic description.

The following table provides a summary:

Protocol	Clock fault	Timer	CR.size	CR.release	Resumption	Complexity
Delta-t	not tolerated	3	small	2δ	not immediate	medium
VMTP	not tolerated	3	small	$> 2\delta$	immediate	medium
XTP	not tolerated	3	small	$(<)2\delta$	not immediate	medium
CMSC	not tolerated	3(5)	> small	2δ	immediate	high
SSCMP	tolerated	3	small	2δ	immediate	low

Our SSCMP requires synchronized clocks and some non-volatile memory, which does not seem to be too severe a restriction. As far as correctness is concerned, it surpasses (the "core" of) any of the other protocols since it guarantees at-most-once delivery even in the case of clock synchronization failures. Note that the violation of the clock synchronization condition is a rather likely event, in particular in systems employing probabilistic algorithms (like NTP), but also in deterministic ones, where at least the possibility of faulty clocks exists. Unlike protocols like Delta-t, for example, SSCMP also tolerates any (very) late message (arriving when the connection record has long been released), and its correctness does not depend on bounds on the transmission rate since there are no implicit timing constraints involved (as in sequence number wrapping/reuse or T-stability in other protocols).

Moreover, SSCMP is purely end-to-end in that it does not require support from the underlying network, allows immediate resumption after reboot, and accepts messages generated during a long lasting receiver crash. Last but not least, its striking simplicity should be judged in view of the fact that packet losses in modern networks are mainly caused by receiver overruns, i.e., performance problems caused by complicated protocols, and not by transmission errors.

6 Conclusions and Further Research

In the previous sections, we introduced and rigorously analyzed a novel timer-based connection management protocol providing reliable ordered at-most-once delivery of messages. Extending the (unsequenced) SCMP algorithm of [LSW91], our SSCMP (Sequenced Synchronized Clock Message Protocol) surpasses (the "core" of) existing protocols like Delta-t, VMTP, XTP, and CMSC as far as correctness and also simplicity is concerned. More specifically, SSCMP provides sequenced at-most-once delivery even in the case of clocks being (totally) out of synch, which is no unlikely event in practice.

SSCMP requires synchronized clocks and some non-volatile memory, but does not need statically preallocated memory for storing connection records, allows multiple outstanding messages for optimal performance, needs no support from the underlying network, and requires only a very simple management of (retransmission) timers. Our actual implementation(s) of SSCMP show that it is simple, efficient, and easily extensible. Some "engineering-extensions" we are currently working on are:

- Providing some suitable rate control scheme as in XTP or VMTP, for example.
- Dealing with the problem of mapping sender/receiver identifiers to connection records in real-time.
- Incorporating connection duration agreement into SSCMP, allowing for both connection-specific (which is rather trivial, cf. CMSC and XTP) and, most importantly, time-variant values of ρ. This also includes measures for a quick graceful close based on handshaking.

- Bidirectional communication via piggybacked acknowledment techniques.
- Providing a *cancel* function for cancelling previously *put* but not yet *getack*'ed messages.
- Dealing with timestamps that may wrap around.

The final version of SSCMP will eventually be used for the basic communication subsystem underlying a high-performance remote system call facility of the distributed real-time operating system mentioned in Section 1. It is designed to run on an Ethernet-coupled system of VMEbus-based 68030 CPUs, with a (limited) amount of non-volatile memory and high-precision synchronized clocks available.

In addition to the engineering issues mentioned above, we are currently working on the following theoretical research topics:

- The proof techniques applied in Section 4 are of course adequate for the simple SSCMP variant discussed in this paper. However, adding the abovementioned features, the proofs get more and more involved. Thus, we are trying to adopt the formal proof technique of [Lam93] for our purposes.
- It is important to explore *systematically* the performance (not correctness!) penalties associated with the various exceptional situations provided for in our system model, and to conceive measures for improvement — a problem that has been somewhat neglected in our and the previous work mentioned. A question of particular importance is how long an exceptional situation may affect the execution of the protocol after it has ceased to exist; remember the distinction between the time an *exceptional condition* and an *exceptional situation* are actually lasting, cf. Footnote 4 in Section 2.

  For example, consider the case of a sender's clock $S.time$ jumping forward considerably (without being detected) and a message m being sent subsequently, which is not an unlikely exceptional situation in practice. Even after clock synchronization (normal operation) is achieved again, the exceptional condition is still lasting: The sender is blocked by our clock monotonicity enforcement until time t with $S.time(t) > m.st$. The only thing we can guarantee at first sight is that SSCMP recovers within finite time (after all, expiration must eventually occur), but that is of course too weak a statement for practical purposes.

A forthcoming paper will be devoted to those issues.

Acknowledgments

We are grateful to Wolfgang Kastner for reading an earlier version of the manuscript and some stimulating discussions on the subject. A long list of comments and suggestions of several anonymous referees have been of great help in improving the overall appearance of the final paper.

References

[Bel76] D. Belsnes. *Single-Message Communication*, IEEE Transactions on Communications, COM-24(2), February 1976, p. 190–194.

[BF93] E. W. Biersack, D. C. Feldmeier. *A timer-based connection management protocol with synchronized clocks and its verification*, Computer Networks and ISDN Systems, 25, 1993, p. 1303–1319.

[Che86] D. R. Cheriton. *VMTP: A Transport Protocol for the Next Generation of Communication Systems*, Proc. SIGCOMM '86, 1986, p. 406–415.

[Che89] D. R. Cheriton. SIRPENT^{TM}: *A High-Performance Internetworking Approach*, Proc. SIGCOMM '89, Austin, Texas, September 1989, p. 158–169.

[CW89] D. R. Cheriton, C. L. Williamson. *VMTP as the Transport Layer for High-Performance Distributed Systems*, IEEE Communications Magazine, June 1989, p. 37–44.

[Lam93] B. W. Lampson. *Reliable Messages and Connection Establishment*, in: S. Mullender. *Distributed Systems*, 2nd ed., Addison-Wesley, 1993, p. 251–281.

[Lis93] B. Liskov. *Practical uses of synchronized clocks in distributed systems*, Distributed Computing, 6, 1993, p. 211–219.

[LSW91] B. Liskov, L. Shrira, J. Wroclawski. *Efficient At-Most-Once Message Based on Synchronized Clocks*, ACM Transactions on Computer Systems, 9(2), May 1991, p. 125–142.

[Mil91] D. L. Mills. *Internet Time Synchronization: The Network Time Protocol*, IEEE Transactions on Communications, 39(10), October 1991, p. 1482–1493.

[Pos81] J. Postel. *DoD Standard Transmission Control Protocol*, DARPA-Internet RFC 793, 1981.

[RSB90] P. Ramanathan, K. G. Shin, R. W. Butler. *Fault-Tolerant Clock Synchronization in Distributed Systems*, IEEE Computer, 23(10), October 1990, p. 33–42.

[Sch94] U. Schmid. *Synchronized UTC for Distributed Real-Time Systems*, to appear in Proceedings IFAC Workshop on Real-Time Programming WRTP '94, Lake Reichenau/Germany, 1994.

[Sha91] A. U. Shankar. *Modular Design Principles for Protocols with an Application to the Transport Layer*, Proceedings of the IEEE, 79(12), December 1991, p. 1687–1707.

[Slo83] L. Sloan. *Mechanisms that Enforce Bounds on Packet Lifetimes*, ACM Transactions on Computer Systems, 1(4), November 1983, p. 311–330.

[SDW92] W. T. Strayer, B. J. Dempsey, A. C. Weaver. *XTP — The Xpress Transfer Protocol*, Addison Wesley, 1992.

[SWL90] B. Simons, L. Lundelius-Welch, N. Lynch. *An Overview of Clock Synchronization*, B. Simons, A. Spector, editors: Fault-Tolerant Distributed Computing, Lecture Notes on Computer Science 448, 1990, p. 84–96.

[Tan81] A. S. Tanenbaum. *Computer Networks*, Prentice Hall, 1981.