Efficient At-Most-Once Messages Based on Synchronized Clocks

BARBARA LISKOV, LIUBA SHRIRA, and JOHN WROCLAWSKI
Massachusetts Institute of Technology

This paper describes a new at-most-once message passing protocol that provides guaranteed detection of duplicate messages even when the receiver has no state stored for the sender. It also discusses how to use at-most-once messages to implement higher-level primitives such as at-most-once remote procedure calls and sequenced bytestream protocols. Our performance measurements indicate that at-most-once RPCs can be provided at the same cost as less desirable forms of RPCs that do not guarantee at-most-once execution. Our method is based on the assumption that clocks throughout the system are loosely synchronized. Modern clock synchronization protocols provide good bounds on clock skew with high probability; our method depends on the bound for performance but not for correctness.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks, network communications, packet networks, store and forward networks*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: At-most-once message passing, message-passing protocols, remote procedure calls, synchronized clocks

1. INTRODUCTION

In this paper we describe a new way of providing at-most-once delivery of messages. Our method allows messages to be sent without prior communication (e.g., to set up a connection), yet it provides an absolute guarantee that duplicate messages will be detected. It is based on loosely synchronized clocks; because it depends on synchronized clocks, we refer to it as the *synchronized clock message protocol*, or SCMP for short. SCMP can easily tolerate the clock skews provided by existing clock synchronization protocols [4]; these skews are typically less than 100 milliseconds, even in a wide area

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988 and in part by the National Science Foundation under grant DCR-8822158.
Authors' address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass. 02139.

network. If the rare event of unsynchronized clocks does occur, the protocol continues to work correctly, although there may be a degradation of performance.

SCMP is the first message-passing protocol to make use of synchronized clocks to provide at-most-once delivery. It is interesting because it provides at-most-once delivery at low cost. In addition, it can be used as a building block for constructing higher-level protocols with desirable performance characteristics. In particular, it can be used to implement at-most-once RPCs efficiently, even in the case where a client communicates only occasionally with each of many servers.

The paper is organized as follows. In Section 2 we describe SCMP and show how it can be used to guarantee at-most-once delivery, discuss our clock requirements and protocol parameter tradeoffs, and compare our technique to the Delta-t message passing protocol [11]. In Section 3 we discuss how SCMP can be used to provide higher-level primitives. The section describes an implementation of at-most-once RPCs based on the SunRPC library [10], compares the performance of this implementation with zero-or-more and at-most-once RPCs already available in the SunRPC library, and compares our RPC method with that of Birrell and Nelson [1]. We conclude with a summary of what we have accomplished.

2. AT-MOST-ONCE MESSAGE DELIVERY

Implementing at-most-once semantics is typically done by having each message receiver maintain a table containing information about senders. When a message arrives, if there is information about the sender in the table it is used to determine whether or not the message is a duplicate. If there is no information, there are two choices: either accept the message or reject it. If the message is accepted, there is a chance of accepting a duplicate. This chance can be made arbitrarily small by keeping information about senders long enough. However, it is difficult to determine how long to keep this information in the presence of sender retransmission and networks with probabilistic delay.

The alternative of rejecting the message is safe; it guarantees that no duplicates will ever be accepted. However, it gives rise to a problem. When a message is sent, we want to be reasonably certain that the receiver will accept it. Therefore we need to know that the receiver has information about the sender in its table. If it is unlikely to have such information, e.g., because this is the first time the sender has communicated with it in a while, then it is necessary to set up the information before sending the message. This can be done by means of a handshake in which a pair of messages is exchanged between the sender and receiver in advance of the at-most-once message. If the sender then sends many messages over the connection established by the handshake, the cost of the handshake is amortized across all of them. If there are only a few messages, the overhead is high relative to useful work. In the worst case, the sender transmits only one message. Yet this case may be quite common; it corresponds to a client that performs a single operation at each of many servers.

The handshake between the sender and receiver could be avoided if there were a way for a receiver to be sure a message was new in the absence of information about the sender. Our scheme allows this to be done by using time. The idea is that the receiver remembers all "recent" communications. If a message from a particular sender is "recent", the receiver will be able to compare it with the stored information and decide accurately whether the message is a duplicate. If the message from the sender is "old", it will be tagged as a duplicate even though it may not be, but this case is very unlikely. Thus the system may occasionally mismark a nonduplicate but it will never erroneously accept a duplicate.

For such a scheme to work, receivers need to know whether a message is "recent". Our scheme accomplishes this by means of loosely synchronized clocks. All nodes have clocks that with very high probability differ by no more than some skew ϵ. When a node sends an "at-most-once" message, it timestamps the message with the current time of its clock. When the message arrives at the receiver, it is considered recent if its timestamp is later than the receiver's local time minus a period ρ; otherwise it is old. The parameter ρ is chosen to minimize the probability of erroneously discarding late messages as duplicates; it is much larger than ϵ. The characteristics of ρ are discussed further in Section 2.4.

The remainder of this section discusses our protocol in more detail. As will be discussed in Section 3, there may be other messages in the system as well, e.g., unreliable datagrams. We do not consider such messages here; in the remainder of this section, "message" will always mean an "at-most-once" message.

2.1 The Model and Assumptions

We are interested in a distributed collection of *nodes* connected by a *network*. All nodes can communicate with one another by sending messages on the network. Although the network might be a local area net, we are concerned here with the more general case of a geographically distributed net such as the Internet. Nodes may fail by crashing. The network may lose or duplicate messages, or deliver them late or out of order; in addition it may partition so that some nodes are unable to send messages to some other nodes temporarily. As is usual in distributed systems, we assume failures are not Byzantine [7]: we assume the nodes are fail-stop processors and the network will deliver only uncorrupted messages.

A program running on the network consists of a collection of *modules*, each of which resides entirely on a single node. Some modules are *servers* and others are *clients* (some modules are both servers and clients). Clients send messages to servers to request some service; servers accept such messages, carry out the request, and sometimes return a response in a message to the client. The exact form of a module is not of interest to us. In some systems, a module is limited to doing one thing at a time; other systems support concurrency within a module. Our mechanism supports both kinds of systems.

Some servers are *resilient*: they survive failures of their node. Resilience requires access to a non-volatile storage medium. The storage need not be located at the server's node; instead it could be provided over the network by a stable storage service [2].

Every node has a clock. As mentioned, we assume that the nodes' clocks are loosely synchronized with some skew ϵ; nodes ensure this by carrying out a *clock synchronization protocol*. At least one practical clock synchronization protocol exists [4]. It synchronizes clocks of nodes on a geographically distributed network so that clocks are guaranteed with very high probability to have a skew of less than a hundred milliseconds. The protocol does this at low cost and low overhead. Synchronized clocks are useful for other purposes than ours, e.g., for authentication [9] and for capabilities that expire. Therefore, our protocol is merely another client of a service used by many parts of a system. Our method is tolerant of all clock failures, although performance may suffer if clocks get out of synch.

2.2 The SCMP Protocol

Every module G has a *current time*, *G.time*; this is read from the clock belonging to its node. Every message m contains a timestamp, *m.ts*; this is G.time of the sending module at the time m is created. Even though a particular message may be duplicated either by the network or by the software that carries out a higher-level protocol, all instances of the message will contain the same m.ts.

Each message also contains a *connection identifier*, *m.conn*. However, as opposed to other connection-based systems, this connection identifier is selected by the client without consultation with the server. If a client has only one outstanding message to a server at a time, its unique module identifier can serve as the connection identifier. If it has many unrelated outstanding messages, it should have a separate connection for each; this could happen, for example, in a client that runs multiple concurrent threads. Thus, in general, a connection identifier is a pair ⟨module id, uid⟩ where the uid is unique relative to the module identifier of the client; the size of the uid field depends on how many outstanding messages a system allows between a client and a server. Note that a client can freely reuse connection ids; distinct ones are needed only when concurrent communication with the same server is occurring.

The connection id and the timestamp in the message together constitute a unique message id, provided that the timestamps of all messages sent on that connection are distinct. Timestamps should have fairly fine granularity (e.g., one millisecond) so that clients can send messages frequently. If the clock at a client has coarser granularity than the timestamp granularity, the client can maintain a counter for use as the low bits of the timestamp when necessary, e.g., if it attempts to send multiple messages within one tick of its clock.

Each server maintains a *connection table*, *G.CT*. This is a mapping from connection ids to connection information. For the discussion in this section, the only connection information of interest is the timestamp of the last

Initialization.
When a server G is first created, its connection table is empty and G upper is set to the minimum of zero and G time - ρ.

Processing of Message m by Server G.
1. If there is an entry for m.conn in G.CT then
   a. if m.ts ≤ G.CT[m conn] ts, flag the message as a possible duplicate
   b else accept the message and set G.CT[m.conn] ts to m.ts

2 Otherwise,
   a. if m.ts ≤ G.upper, flag the message as a possible duplicate
   b. else accept the message and set G.CT[m conn] ts to m ts.

Garbage Collection
Periodically, the server G removes entries from G CT. Only entries with timestamps ≤ G time - ρ are removed. Then G upper is set to the maximum of its former value and the timestamps of the removed entries

Fig. 1. Processing at a nonresilient server.

message accepted on that connection. Not all connections have an entry in G.CT. G is free to remove an entry for connection C from its connection table provided G.CT[C].ts ≤ G.time − ρ. Here, ρ is the interval mentioned above during which we retain connection information; see Section 2.4 for further discussion.

A server also maintains an upper bound, *G.upper*, on the timestamps that have been removed from the table. Since only old timestamps are removed from the table, G.upper ≤ G.time − ρ.

Figure 1 describes the processing at servers that do not survive crashes (resilient servers are described below). The algorithm works by determining a per-connection bound that distinguishes "new" from "old", or potentially duplicate, messages, and comparing the timestamp of the newly arrived message with that bound. If the messages's connection has an entry in the table G.CT, the bound is the timestamp of the most recent previously accepted message. If there is no table entry, the global bound G.upper is used. G.upper is an appropriate bound because if there is no information for the connection in G.CT, this means the last message on the connection (if any) contained a timestamp t ≤ G.upper. Therefore, if a message arrives whose timestamp is later than this, it must be a new message. Since G.upper ≤ G.time − ρ, we rule out (with high probability) the chance of incorrectly flagging a message as a duplicate, provided ρ is large enough.

For servers that survive crashes, we need a way to determine whether a message that arrives after crash recovery is a duplicate of a message that arrived before the crash. We could accomplish this by keeping connection information in stable storage [3] but this would be very inefficient, since each change to the table would have to be recorded immediately on stable storage.[1]

[1]We consider here only crashes in which volatile memory is lost. If some volatile memory survives, the connection table could be kept there, and the server crash would have no effect on our algorithm. We conjecture that wholesale discarding of information in volatile memory after a crash will become less common in the future as interest in fault-tolerant systems grows.

Initialization.
When a server G is first created, its connection table is empty and G upper is set to the minimum of zero and G time - ρ. G time + β is written to stable storage and G latest is set to this value

Processing of message m by server G
1 If m.ts > G latest, refuse the message since it is too early

2 If there is an entry for m conn in G CT then
a. if m.ts ≤ G.CT[m conn].ts, flag the message as a possible duplicate
b. else accept the message and set G CT[m conn].ts to m ts

3 Otherwise,
a. if m ts ≤ G upper, flag the message as a possible duplicate
b. else accept the message and set G.CT[m.conn] ts to m ts

Maintaining Latest
Periodically G writes G time + β to stable storage and then sets G latest to this value

Garbage Collection
Periodically, the server G removes entries from G CT Only entries with timestamps ≤ G time - ρ are removed Then G upper is set to the maximum of its former value and the timestamps of the removed entries

Crash Recovery
Initialize G upper to be the value of G.latest on stable storage Establish a new G latest for use in accepting further messages The connection table is empty

Fig 2. Processing at a resilient server.

If the connection table does not survive the crash, we need a way of effectively reinitializing it after a crash. We do this by using stable storage, but we need write only a little information once in a while.

Our approach is to establish an estimate of the timestamp of the latest message that may have been received before the crash. The idea is that before the crash no message whose timestamp was greater than the estimate was accepted. Therefore, any message after the crash whose timestamp is greater than the estimate is not a duplicate and can be accepted. Messages with timestamps less than or equal to the estimate might be duplicates, so they must be flagged as such. Thus our plan is the following:

(1) Before the crash we must ensure that all accepted messages have timestamps less than the estimate that will be established should a crash occur. This means we must enforce an upper bound on the timestamps of accepted messages. We will refer to this bound as *G.latest*.
(2) After a crash we establish the estimate and use it to initialize G.upper.

The full algorithm carried out by a resilient server is in Figure 2. In step 1 of message processing, we need not discard a message that arrives too early; instead we can just delay processing of such a message.

We establish G.latest as follows: Periodically a server writes G.time + β to stable storage; G.latest is the most recent value written to stable storage. β is some increment that ensures that we are unlikely to reject (or delay)

messages as arriving too early; it would be based on the clock skew ϵ, the time required to write to stable storage, and the frequency of writing. Stable storage need only be written infrequently, for example, once every few seconds in the background; so this work will not cause much of a drain on the server. For many persistent servers, G.latest can simply be written to stable storage as part of the records that are being written there anyway to record information about the server's persistent state.

The following alternative algorithm for maintaining G.latest is also possible. During normal processing, the server treats messages as too early if m.ts > G.time + ϵ, rather than keeping a separate value for G.latest. When restarting after a crash, the server refuses all messages for a period of ϵ, then sets G.upper to G.time and begins normal processing. This alternative eliminates the need for stable storage, but it requires that clocks be monotonic, and it will erroneously reject a large number of messages even after a long crash.

2.3 Correctness

The following brief argument should convince the reader that SCMP properly detects duplicates. Assume H sends a message m on connection C and assume G receives multiple copies of m. We need to show that at most one copy is accepted. We know G.CT[C] contained m.ts when the first copy of m was accepted. By examination of the steps of the protocol, we can see that a later copy of m will be flagged as a duplicate provided the value in G.CT[C] is greater than or equal to m.ts, or, if G.CT[C] does not exist, that G.upper is greater than or equal to m.ts. These properties follow from the following invariants:

Invariant I1: For any connection C, the value in G.CT[C] is monotonically increasing.

Invariant I2: At time t, G.upper is greater than or equal to the timestamp m.ts of any message m that entered and left G.CT before t.

Invariant I1 holds because SCMP modifies a connection table entry G.CT[C] either by overriding the previous value in the protocol step 2b, where the timestamp is guaranteed to increase because of the test in step 2a, or by writing into an empty entry in step 3b, where the increase is guaranteed provided invariant I2 holds. Invariant I2 holds initially and is preserved by a garbage collection step. It is also preserved across crashes. By step 1, we know that any message m that enters G.CT before the crash has a timestamp m.ts that is less than or equal to G.latest at the time m is accepted. Because of the way G.latest is computed and the way G.upper is initialized after a crash, we know that

$$\mathrm{G}.latest_{\mathrm{precrash}} \leq \mathrm{G}.upper_{\mathrm{postcrash}}$$

for all values of G.latest that existed before the crash. Therefore, invariant I2 is preserved by the crash recovery step.

Clock synchronization is used in the protocol to establish a system-wide notion of "recent", which is used to delete connection information from the table, and as a consequence to flag incoming messages as possible duplicates. The correctness of the protocol does not depend on clock synchronization but its performance does. If clocks are synchronized, only messages that are very late will actually be flagged as duplicates; otherwise, recent messages may be flagged in this way. If G's clock is slow, G's messages are more likely to be flagged as duplicates by other modules; also G may refuse (or delay accepting) other modules' messages (since they are "too early") and G may maintain connection information that is out of date. If G's clock is fast, it is more likely to flag messages from other modules as duplicates and its messages may be refused or delayed as "too early". Thus, in either case no duplicate messages are accepted, but performance suffers because there is extra flagging of duplicate messages and some out-of-date connection information may be kept in connection tables.

The correctness of the protocol also does not depend on the monotonicity of clocks (in the scheme in which G.latest is kept on stable storage). If a node's clock runs backwards, the only effect is that messages are likely to be flagged as duplicates. A resilient server must be careful to ensure that the value of G.upper after a crash is greater than any G.latest before the crash, but this is easy to do. The server merely ensures that the value of G.latest on stable storage is monotonic. The only time monotonicity might be violated is if stable storage fails. But in this case, the server will have failed catastrophically since it has lost its persistent state.

2.4 The Parameter ρ

This section discusses the engineering tradeoffs that arise in setting the value of ρ. The parameter ρ determines the time during which a message will be considered recent, and consequently the length of time information must be retained in the receiver's connection table. Selection of a value for ρ requires a compromise between the performance penalties associated with overly large connection tables and the probability of erroneously marking messages as duplicates.

The size of a receiver's connection table is determined by the number of recently active senders (which cause entries to be added to the table) and the value of ρ (which determines when entries may be removed from the table). Often, ρ can be made quite large with no noticeable penalty. When resources are limited, it is desirable to limit ρ to keep table memory usage, paging overhead, and search time low. This is particularly true when higher-level protocols are also storing information in the connection table, leading to larger entries. However, making ρ too small will reduce performance. We consider here the determination of an appropriate minimum value for ρ.

We assume that the user of SCMP (e.g., a higher-level protocol such as RPC) requires that messages from the sender be delivered to the receiver with a probability P_{req}. We further assume a network that, when handed a datagram, will deliver at least one copy of it with probability P_{net} in time δ or less. If the raw network does not provide adequately reliable service, the

higher-level protocol must implement an error recovery strategy involving retransmission of failed datagrams.[2]

The SCMP algorithm puts a bound on the time available to convey a sender's message to the recipient. If the message has not been delivered successfully by $T_{transmit} + \rho - \epsilon$, it might be flagged as a duplicate by the receiver. The minimum value for ρ is the time necessary to deliver at least one copy of the sender's message with probability of P_{req}, plus an adjustment for clock skew. This value is determined by the characteristics of the network and the error recovery algorithm of the higher-level protocol.

The determination of ρ is particularly simple when $P_{net} \geq P_{req}$. Here the sender will never need to retransmit a message. ρ can be based entirely on δ, the delay of a single datagram in the network, and on ϵ, and the simple bound $\rho \geq \epsilon + \delta$ holds.

When $P_{net} < P_{req}$, the sender may have to retransmit the message to achieve the desired reliability. If the sender might need to retransmit the message up to N times, a bound on ρ is given by

$$\rho \geq \epsilon + \delta + RT(N)$$

where RT(N) is the time from when the message was first sent to the time the Nth retransmission is sent.

A point to notice is that the server has control of the value of ρ and in fact can use different ρ's for different classes of clients. This classification can be performed using hints from the clients themselves. Clients enclose a suggested value for ρ in each message. The server sorts clients into classes based on the client's suggested value for ρ, and uses an appropriate actual value for each of these classes. Alternatively, if a server is able to classify clients based on the characteristics of the underlying communication path (e.g., using information in the client ID), it could use larger ρ's for clients on slower or less reliable paths and smaller ρ's for clients on better paths. Note that in either case the server needs to maintain a different G.upper for each category of clients. Otherwise, removal of table entries for clients which are using a small ρ may raise G.upper excessively, leading to increased rejection of messages from clients which are using a larger ρ. These strategies grant clients with higher delivery requirements, or on low quality or long delay paths, the extra time needed to reliably transmit a message, while minimizing the unnecessary use of resources by clients with lesser requirements.

The above analysis considers only transient failures in the network. Some protocols support recovery from long-term network partition, recipient node failure, and other statistically unlikely events by allowing the sender to retransmit indefinitely, until communication is reestablished. This approach does not always work with SCMP-based protocols, because after a time retransmitted messages may become too old and will not be accepted by the receiver. (Note that we cannot solve the problem by giving the message a later timestamp since this can cause it to be accepted when it is actually a

[2]Note that this analysis does not depend on the *method* of triggering retransmission, e.g., negative acknowledgment or timeout.

duplicate.) When such a message is rejected, a higher-level failure recovery mechanism must be used. We are explicitly trading simplicity in the common case (no handshaking on normal connection startup) for somewhat more complexity in the unusual case of a long-term failure.

In fact, however, the algorithm performs well in the case of a receiver crash. We would like to accept as many messages as possible after the restart, even those older than G.time $- \rho$. The reason for this is that ρ is based only on the expected delay in delivering messages in the absence of a long-term failure. Things are different when the receiver has crashed; the sender might keep trying to send the message for a period much longer than ρ.

If the crash duration is very brief, many of these messages will have timestamps less than t_{cr}, the time at which the crash occurred. Here storing G.latest on stable storage is not much help, since G.latest will usually be greater than or equal to t_{cr}. We would be able to accept such messages only by having the missing connection table information.

Typically, however, crashes last a long time relative to the length of time senders remain interested in messages. Therefore a much more likely case is that all messages of interest have timestamps substantially greater than t_{cr}. For such crashes, our scheme of reinitializing G.upper from the G.latest stored on stable storage periodically will allow us to accept virtually all messages of interest. In particular, even in the worst case we will accept all messages sent after $t_{cr} + \beta$. (Recall that G.time $+ \beta$ is written to stable storage periodically.)

To ensure that the algorithm works properly immediately after a restart, it is necessary to avoid removing entries from the table until sufficient time has passed to process all pending messages. Otherwise, if a message with a recent timestamp is removed, G.upper will be adjusted upwards, and messages with older timestamps will no longer be accepted.

We can reduce the number of unnecessary message rejections caused by a crash by increasing the frequency with which we write to stable storage and the amount of information we write. We are again encountering here the common tradeoff between optimizing for the normal case and for recovery. We chose a method that has essentially no impact on normal case behavior, since writing to disk every few seconds is not much of a drain, and could even be combined with other information that the server must write. Alternatively, every change to the connection table could be recorded on stable storage; this would ensure that a crash causes no erroneous rejections, but would slow down normal processing substantially.

2.5 Comparison with Delta-t

The Delta-t protocol [11, 12] also implements at-most-once messages without requiring connection setup. Our work is closely related to Delta-t; the key difference is that our method does not require the network to enforce a maximum lifetime on messages.

Delta-t works by allowing the transmitter of a message to specify a bound on the lifetime of transmitted messages. Each message initially contains this

bound as one of its fields. Routing nodes within the message's path decrement this field by the amount of time they spend processing the message. If the result is negative, the message is discarded. Otherwise, the message is forwarded with the decremented bound. End nodes, as well as routing nodes, must participate in the packet lifetime bounding algorithm, due to queueing delays at both endpoints and message retransmission at the sender. Note that the algorithm must account for the time a message spends transiting data links, as well as the time spent in queues at the various nodes.

Servers maintain information about a received message until the remaining lifetime of the message has expired. The server need not check for duplicates after this point because they are assumed to be discarded by the switching nodes. Note that Delta-t relies on a property closely related to clock synchronization, since it requires that the clocks of all switching nodes run at the same rate.

Both Delta-t and our protocol work by enforcing a maximum message lifetime, thus limiting the time connection state information must be stored at the receiver. Our protocol enforces this lifetime with an algorithm in which all work is done at the end nodes, while Delta-t relies on both the end nodes and the network to enforce message lifetimes using a per-hop algorithm.

We believe our protocol is preferable to Delta-t for several reasons:

(1) Our protocol is purely end-to-end, while Delta-t requires cooperation from the network. We can make do with very simple switching nodes. The switching nodes in Delta-t are more complicated since they must bound the maximum packet lifetime. In particular, nodes in a network where the links have unpredictable delays must carry out a separate and somewhat complex link-transit-time protocol [8].

(2) Our protocol provides better performance than Delta-t. The reason for this is that the mechanism used to maintain state between the client and server is an inexpensive out-of-band clock synchronization algorithm, rather than an in-band maximum packet lifetime algorithm involving all nodes in the net.

(3) Our protocol is fail-safe. The Delta-t protocol will fail if clock rates at the different nodes vary by too much or the link transit time of a message is underestimated, since then there is a chance of a duplicate arriving at a server after information about the earlier copy of the message has been discarded.

3. HIGHER-LEVEL PROTOCOLS

In this section, we discuss how SCMP can be used to implement higher-level protocols. The SCMP protocol can be viewed as a filter that receives messages from a network and passes them up to a higher-level tagged as either "new" or "duplicate". Higher-level protocols use new messages to initiate actions such as establishing a connection in TCP or starting an RPC call. They may either discard duplicate messages or use them to initiate a recovery action, such as retransmitting a result or sending an acknowledgment.

As mentioned earlier, our at-most-once messages are not intended to replace other low-level communication primitives. Instead, we assume that there are also unreliable datagrams or some similar primitive. Higher-level communication primitives are implemented out of a combination of datagrams and at-most-once messages. Typically, at-most-once messages are used to *start* the protocol associated with a higher-level primitive; the remainder of the protocol makes use of datagrams. Thus the RPC protocol discussed below uses an at-most-once message for the (first part of the) call; the reply and control messages are sent using UDP [5]. Similarly, we could implement connection-based protocols like TCP [6] by having the first message from the client to the server be an at-most-once message; all other messages would be datagrams. This would allow us to piggyback the connection setup on the first message to be exchanged over the connection. The client could choose the connection id, and all messages on the connection would contain this id. In addition, messages on the connection would be distinguished by sequence number in the usual way. When initializing a streaming protocol in this fashion, the client may wish to send several messages in the sequence before receiving the initial reply from the server. It is important in this case that the SCMP algorithm be applied only to the first message in the sequence. The reason for this is that SCMP is sensitive to message reordering; if two SCMP messages sent on the same connection are received in reverse order, acceptance of the second message will cause the first message to be rejected as a duplicate when it arrives.

The use of SCMP is most interesting when the client uses a connection very little, since in this case we gain the most from avoiding extra messages for connection setup. RPC is an example of this kind of communication, especially in the case where clients call servers only occasionally. Below we present an implementation of at-most-once RPC using SCMP and then compare our implementation with that of Birrell and Nelson [1]. At-most-once semantics for RPCs means that a call is guaranteed to be executed at most once even when failures occur such as a crash of the receiving module; it is desirable because it provides proper semantics even when calls are not idempotent.

3.1 An Example: At-Most-Once RPC Implemenation

We have added an SCMP-based at-most-once RPC implementation to the widely used SunRPC library [10], which currently supports UDP- and TCP-based RPC protocols. Sun's UDP-based protocol provides zero-or-more semantics, which gives only weak guarantees about how many times a call is executed: even when a call terminates normally, it may have been executed more than once. Sun's TCP-based protocol provides at-most-once call execution when there are no crashes; in the case of a crash, however, a call can be run more than once. Our protocol provides at-most-once semantics even across crashes. Measurements indicate that we can provide at-most-once semantics at about the same cost as the UDP-based SunRPC, and with significantly better performance than the TCP-based SunRPC in the case of clients calling servers occasionally.

Our at-most-once RPC provides reliable delivery: a client can depend on the protocol to deliver the call message provided the server is accessible to the client, and similarly the protocol guarantees to deliver the reply. It is implemented by using an at-most-once message for the call message, and UDP datagrams for the reply and control messages. This protocol is optimized for the common case of mostly-reliable networks and heavily loaded servers.[3] Since SunRPC modules are single-threaded, we use a connection identifier consisting simply of a unique identifier for the client; the client uid and timestamp provide the unique call id. We support only calls that fit in a single UDP message. However, these messages can contain up to 64K bytes.

The client makes an RPC by sending a CALL message to the server. When the call has been executed, results are returned to the client in a REPLY message. When there are no failures, only these two messages are needed for an RPC that does not take long for a server to execute, although the client may optionally send a REPLY-ACK message to inform the server that it has received the reply. To ensure reliability, clients retransmit CALL messages on a periodic basis; the server responds with an ACK message if it receives a duplicate call, or a REPLY if the results have already been computed. (If the client learns from such an ack that the server has received the CALL message, subsequent retransmissions send a truncated message that contains only the connection and call ids.) If several retransmissions have occurred without a response from the server, the client times out the call.

The server maintains state for each active connection in a connection table. Each table entry e contains:

—The server state *e.state* for this connection, one of IDLE, COMPUTING or REPLYING.

—A message timestamp *e.msg_timestamp*, which records the timestamp in the most recently received CALL message.

—If the state of the connection is REPLYING, the server's reply to the most recently processed call and a reply timestamp *e.reply_time*, which records the time the reply was first transmitted.

Processing on the server side is described by Figure 3. The server starts performing a call when a CALL message tagged as new by the SCMP algorithm is received from a client. While the server is performing the call, a duplicate CALL message triggers the transmission of an ACK message to the client. When the server completes the call, it sends the results to the client in a REPLY message, and saves them in its connection table. If the server receives a duplicate CALL message at this point, it retransmits the reply. If a new CALL message is received, the server discards any stored reply and begins processing the new call. If a REPLY-ACK message is received, the stored reply is discarded, and the connection returns to the IDLE state. Periodically the server connection table is garbage collected. COMPUTING

[3]In other situations, different choices would be better than ours. The result would be a family of RPC protocols.

```
Initialization and Crash Recovery
    (Re)Initialize SCMP algorithm as in Figure 2.
    The connection table CT is empty

Process Incoming CALL Messages
    1 Apply SCMP algorithm to message, tagging it as NEW or DUPLICATE
        If message is NEW and there is no connection table entry,
        one is created by SCMP

    2 If message is NEW.
        discard stored reply, if any.
        set server connection state e state to COMPUTING
        set e.msg_timestamp to the timestamp in the message
        begin user-level computation of results

    3 Otherwise (message is a DUPLICATE).
        if no connection table entry, ignore message
        if state is COMPUTING, send ACK
        if state is REPLYING, retransmit stored reply

Process completion of user-level computation
    Transmit REPLY message to client.
    Set state to REPLYING
    Store results in appropriate connection table entry e.
    Store current time G time in e reply_time

Process Incoming REPLY-ACK messages
    Delete stored reply information, if any
    Set state to IDLE

Garbage collection
    Examine each entry e in the connection table
    1 If state is COMPUTING, do nothing.

    2 If state is IDLE or REPLYING,
        discard the entry if e reply_time < G time - ρ'
        Set G upper to the maximum of its former value and
        e msg_timestamp of the removed entry
```

Fig. 3. Processing of RPCs at the server.

connection entries are never deleted. IDLE and REPLYING connection entries may be deleted whenever the reply timestamp e.reply_time is older than G.time $- \rho'$; entries are deleted when the table is almost full. Here ρ' is chosen to be the maximum of two values, the period ρ specified by the SCMP algorithm, and a constant κ selected to ensure a high probability of the client receiving the reply if it is still interested in the result. We use a value of five minutes for both ρ and κ, based on the characteristics of our client retransmission algorithm and the nominal value of two minutes for δ in the internet protocol suite. Note that G.upper is set using the client's call time, not the reply time, since our only concern is recognizing duplicates of the call. Since we do not remove connection table entries unless the table is almost full, we retain them for a long period after recovering from a crash. This provides us with a sufficient period to accept any pending messages as discussed in Section 2.4.

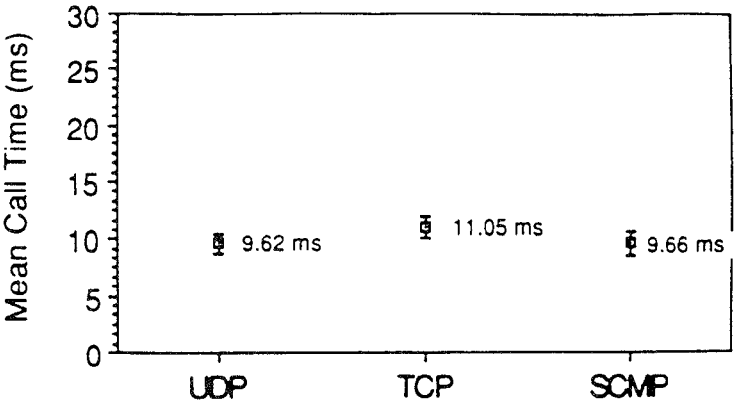

Fig. 4. Performance when a client makes multiple calls to a server.

We compared the performance of our RPC protocol with both UDP and TCP-based protocols by running two experiments. Our measurements were obtained by making null calls between server and client modules running on DEC MicroVax 3 processors under Berkeley Unix 4.3. The nodes were connected by an Ethernet. Our first experiment compared the performance of the three RPC mechanisms while performing a sequence of 1000 calls from a single client to a single server. The results are presented in Figure 4, which shows results obtained from three different runs for each protocol. The data were obtained at times when the load on the network was low. As can be seen, SCMP-based RPC performs comparably to UDP-based RPC, and better than TCP-based RPC. This experiment shows that our protocol does not impose any significant overhead cost over UDP in the case where a client makes many calls to the same server.

Our second experiment compared the performance of the protocols while performing calls from a thousand different clients to the same server; in this experiment each client made exactly one call to the server. This experiment simulates the situation of interest: a system of many clients and servers, where clients talk to servers only occasionally. The results are shown in Figure 5. In this case we can see that the cost of our protocol remains similar to the UDP-based RPC, but the cost of the TCP-based RPC grows dramatically because of the need to establish a new connection for each call.

3.2 Discussion of Birrell and Nelson's RPCs

The RPC implementation of Birrell and Nelson [1] is a simple mechanism that (almost) supports at-most-once semantics. In this section we describe their method and discuss its behavior in the presence of late messages and failures. We also discuss how SCMP can be used with their method to provide truly at-most-once semantics.

A call message in the Birrell and Nelson mechanism contains a client uid, a sequence number, and an "incarnation number" for the server. The client uid and sequence number together constitute the unique call id; each

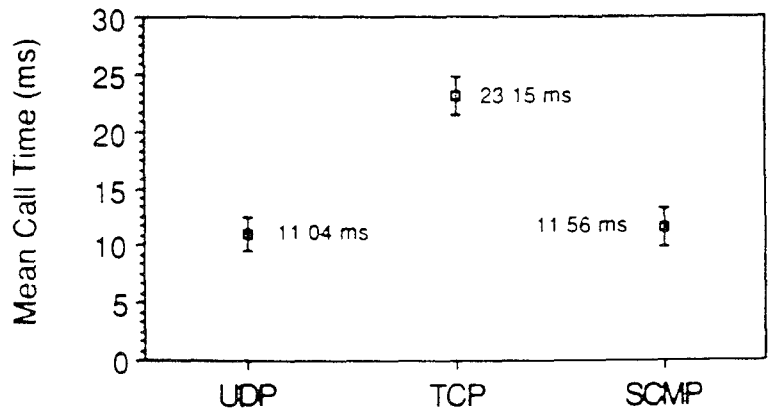

Fig. 5. Performance when clients make occasional calls to a server.

succeeding call must have a larger sequence number. When the server has performed a call, it sends a reply message to the client. The client must then send an acknowledgment message for the reply to the server. The next call can serve as this ack; if there is no next call, an explicit ack is needed.

A server maintains a connection table that stores the sequence number of the most recent call from a client, and also the reply message for the last call until it is ack'd. When an explicit ack arrives, the connection becomes "idle". Information about an idle connection is retained for some period of time σ chosen to be substantially larger than the maximum expected delay in the network. Provided a client never sends a call message after it has sent an ack for that call, it is highly unlikely that a duplicate call message will be received after the connection information is discarded.

When a call message arrives at a server, the following occurs:[4]

(1) The incarnation number in the message is checked against the server's number; if they do not match, the message is rejected.
(2) If the server's connection table contains an entry for the client, the message's sequence number is compared with that in the table. If the message's number is greater than that in the table, it is accepted and its number stored in the table; otherwise it is rejected.
(3) If there is no entry in the table, the message is accepted. An entry is created for the client and the message's sequence number is stored in it.

The incarnation number in the message will match that at the server provided the server has not crashed (and recovered) since the client obtained the incarnation number (by "binding" to the server). In this case, the client's message will be accepted unless information in the table indicates that it is

[4]This description is simplified For example, we ignore the techniques used to generate client retransmissions.

old. When the incarnation numbers match, the protocol does not accept duplicates *provided* the network never delivers late messages; otherwise, the message accepted in step 3 may be a duplicate. As mentioned, for some networks (e.g., local area networks) there can be no late messages; however, for wide area networks, late messages are possible, although rare.

Our protocol does not use incarnation counts and does not require the initial handshake needed by the Birrell and Nelson protocol to obtain an incarnation number before communicating with a server. A handshake is sometimes needed anyway as part of a higher-level binding step, but our protocol avoids this overhead when binding is not required, as in the case of well-known servers or messages forwarded through a well-known address.

Thus, in the absence of crashes, both protocols have problems with late messages. However, they fail in different ways: our protocol fails by rejecting a possible nonduplicate, while theirs fails by accepting a possible duplicate.

Now we consider the case of crashes. The Birrell and Nelson protocol guarantees that crashes will not lead to duplicate messages being accepted. However, it does so at the cost of a handshake: the client must exchange messages to obtain the correct incarnation id before sending the call. Our mechanism also does not accept duplicates across crashes, but does not require the handshake. Both systems will reject nonduplicates. However, we will reject fewer messages: only those messages that were sent at approximately the time of the crash are rejected. By contrast, Birrell and Nelson will reject all messages with the previous incarnation id, both old and recent.

An alternative to our implementation is to use the SCMP algorithm within the Birrell and Nelson method to provide fail safe behavior. This is accomplished by timestamping their call and ack messages; the timestamp would be included in these messages as a separate field. Each time the client sends a call or ack, it puts its current time in the message; thus retransmitted call messages for the same RPC would contain different timestamps. When an ack is received, its timestamp is stored in the connection table entry. The entry can be removed, and G.upper updated, after a time period approximately equal to the maximum expected delay in the network. Call messages on connections without an entry in the table are handled according to our algorithm. No duplicates will be accepted with this method provided the timestamps in call messages for an RPC are always less than those of acks for that RPC or call messages for later RPCs. This approach still requires incarnation numbers, but has the advantage over our approach that clients can retry calls indefinitely until a server responds, which would be helpful, for example, in the case of a long-lasting network partition.

4. CONCLUSIONS

This paper has shown how to implement at-most-once message delivery without connection setup by using loosely-synchronized clocks. The method is fail safe: it never delivers a duplicate message. It may incorrectly reject a nonduplicate, but this is highly unlikely, even when there are node crashes. The protocol is superior to the Delta-t protocol because it is a purely end-to-end method that does not require support from the routing network.

We also discussed how at-most-once messages can be used to implement high-level communication primitives, and described such an implementation for at-most-once RPCs based on the SunRPC library. Our performance data indicate that we can provide at-most-once RPCs at the same cost as zero-or-more RPCs. Our method outperforms at-most-once RPCs based on protocols such as TCP that use extra communication to establish connections, especially in the case where clients talk to servers only occasionally.

Our method relies on clock synchronization. We believe that systems of the future will provide synchronized clocks since they are useful for many different reasons; our protocol is just another client of this service. Existing clock synchronization protocols guarantee with very high probability a clock skew of less than a hundred milliseconds even in a wide area network, and they do so at low cost. Since the synchronization guarantee is probabilistic, it is better not to rely on it for correctness, however. The correctness of our scheme does not depend on clocks being synchronized. We do depend on synchronization for good performance; this is a comfortable assumption because of the high probabilities.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the useful comments made by many, including David Gifford, Karen Sollins, David Tennenhouse, and the referees.

REFERENCES

1. Birrell, A. D., and Nelson B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
2. Daniels, D. S., Spector, A. Z., and Thompson, D. S. Distributed logging for transaction processing. In *ACM Special Interest Group on Management of Data 1987 Annual Conference.* ACM SIGMOD (San Francisco, May 1987), 82–96.
3. Lampson, B. W., and Sturgis, H. E. Crash recovery in a distributed data storage system. Tech. Rep., Xerox Research Center, Palo Alto, Calif., 1979.
4. Mills, D. L. Network time protocol (Version 1) specification and implementation. DARPA-Internet Rep. RFC 1059. 1988.
5. Postel, J. User datagram protocol. DARPA-Internet RFC 768. 1980.
6. Postel, J. DoD standard transmission control protocol. DARPA-Internet RFC 793. 1981.
7. Schlichting, R. D., and Schneider, F. B. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (1983), 222–238.
8. Sloan, L. Mechanisms that enforce bounds on packet lifetimes. *ACM Trans. Comput. Syst. 1*, 4 (Nov. 1983), 311–330.
9. Steiner, J. G., Neuman, C., and Schiller, J. I. Kerberos: An authentication service for open network systems. Tech. Rep. Project Athena, MIT, Cambridge, Mass., Mar. 1988.
10. Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification, Version 2. DARPA-Internet RFC 1057. 1988.
11. Watson, R. W. Timer-based mechanisms in reliable transport protocol connection management. *Comput. Networks 5*, 1 (Feb. 1981), 47–56.
12. Watson, R. W. The Delta-T transport protocol: Features and experience. In *Proceedings of the 14th Conference on Local Computer Networks.* Oct. 1989, IEEE Society Press, 399–407.

Received March 1990; revised February 1991; accepted February 1991