MTP: An Atomic Multicast Transport Protocol

Alan O. Freier
Networking and Communications
Apple Computer, Inc.
Cupertino, CA 95014
freier@apple.com

Keith Marzullo *
Cornell University
Computer Science Dept.
Ithaca, NY 14853
marzullo@cs.cornell.edu

July 30, 1990

Abstract

This paper describes MTP: a reliable transport protocol that utilizes the multicast strategy of applicable lower layer network architectures. In addition to transporting data reliably and efficiently, MTP provides the client synchronization necessary for *agreement* on the receipt of data and the joining of the group of communicants.

Keywords: reliable transport, multicast, broadcast, atomic broadcast, agreement.

1 Introduction

A multicast transport is a virtual circuit connection among a set of communicating peer-level processes. As such, any multicast transport protocol has to satisfy somewhat conflicting goals. Being a *transport* protocol, it should supply quick and reliable delivery of large amounts of client data among the communicants. Yet, being a *multicast* protocol, it should be able to supply the ordering and agreement on the delivery of the data that is necessary for writing decentralized applications. Agreement on order and delivery can

*Keith Marzullo is supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, Contract N00140-87-C-8904. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

take time, thereby slowing the delivery of the data. Hence, most multicast protocols concentrate on a smaller set of goals; for example, [CP88] and [CW89] concentrate on fast delivery while [KTHB90] concentrates on the fast ordered delivery of relatively small messages.

MTP, the transport described in this paper, attempts to satisfy both of these goals. MTP is a full-duplex, flow-controlled, reliable multicast protocol in which the data is sequenced into (perhaps long) messages. Messages are sent within a process group called a *web*, where each message has a single sender and is received by all members of the web. The members of the web agree on the order of receipt of all messages and can agree on the delivery of the message even in the face of partitions[1].

MTP can be thought of as two protocols: a transport layer running underneath an ordering and agreement layer. The transport layer is a negative acknowledgement (or NAK) based protocol exploiting the high probability of successful message delivery that the local area networks of today provide [CLZ87]. Additionally, this transport utilizes the underlying data link and physical layer's capability to do multicast addressing. The ordering and agreement protocol uses a *sequencer site* [CM87,KTHB90] called the *master* that grants serialized *tokens* to producers.

The rest of this paper proceeds as follows. In Section 2, the class of applications for which MTP is meant is contrasted with those applications other atomic broadcast protocols support. The protocol is presented in Section 3. Suggestions for values of the protocol's parameters are derived in Section 4, and a discussion of MTP is given in Section 5.

2 Applications

MTP is designed to support applications that consist of a large number of processes, where the processes send large messages and where the application must be fault-tolerant (we consider *crash failures* of processes and *communication link failures* that can lead to partitioning). Examples of such applications include multimedia teleconferencing systems, multiscreen educational systems, and stock brokerage systems. In making this assumption, we intentionally exclude certain classes of applications that have been considered elsewhere; in particular, those structured as client-server systems with highly available services (*e.g.* [Sch90,MS88]).

[1] A *partition* is the separation of a network of processes into two or more disjoint sets that cannot communicate with each other.

One issue that MTP must address is the efficient handling of network partitioning. An argument can be made that transient partitioning is a very common failure in the kind of applications we are considering [Cri90]. Timeouts are used to detect both crash failures and communication failures. If a machine uses a timeout period that is too short, then it will appear to the machine that the network has temporarily partitioned. For CSMA/CD type data links, there is no upper bound on message delay (communication and operating system software can also increase the variance of this delay), so such transient partitions will be unavoidable. The application designer must balance the cost of recovery from partitioning against the penalty of using excessively long timeouts. Additionally, packets can be dropped due to temporary congestion at both routers and workstations, again creating transient partitions.

Our approach to tolerating partitions is to choose one process in the web to be a distinguished process called the *master*. Since an MTP web contains such a distinguished process, partitions can be treated in the same way as crash or timing failures. If the master process p_0 cannot communicate with a member process p_1, then p_0 assumes that p_1 has failed. If p_1 has instead partitioned away from p_0, p_1 will know that p_0 considers p_1 to have failed and behave accordingly. The vulnerability of a web to the failure of the master is a matter of concern, however. If the application is to be long-lived, care must be taken in choosing the machine that runs the master. In Section 5.2, we discuss some techniques for making a master more robust.

Most other atomic broadcast algorithms are structured in a very decentralized manner so the failure of any (usually size-bounded) subset of the processes will not cause the application to fail. Being fault-tolerant in this manner is very important for implementing highly-available services, but it means that the complex issue of tolerating partitions in a decentralized manner must be addressed [DGMS85][2].

A more detailed description on the issues and uses of reliable broadcast protocols can be found in [JB89].

---

[2] A notable example of an atomic broadcast protocol that does not have a decentralized structure is described in [KTHB90], although as presented, this protocol cannot tolerate partitioning.

3 Protocol

Section 3.1 describes the overall structure of an MTP web. In Section 3.2, the ordering and agreement protocol is described assuming an abstract transport protocol. In Section 3.3, the transport protocol is described, and in Section 3.4 the ordering and agreement protocol is extended to support the establishment of a web and the joining of a member.

3.1 Web Structure

An MTP web consists of a master process and a set of member processes. Member processes may join and leave the web, but the master process cannot, as the web is both instantiated and terminated by the master. All data is *reliably multicast*: that is, every process agrees on the order that a given message will be processed, and the transport guarantees that any given message is either accepted by all non-failed processes or not accepted by any non-failed processes.

There are four transport service access points (TSAPs) associated with a given web:

1. *Multicast transport addresses*: These are the addresses to which all messages targeted for the entire web are transmitted. Each consists of a multicast network service access point (NSAP) catenated with a unique transport connection identifier.

2. *Master's transport address*: This is the TSAP for the master process. This address is the destination of messages for the master process, such as requesting a token or leaving the web. This address is also the source of any message sent by the master process.

3. *Join transport address*: This is the NSAP for the service[3] catenated with the predefined *join* transport connection identifier. This address is the destination of all requests to join the web.

4. *Member transport addresses*: These are the addresses of all the processes that are currently members of the web. Each consists of the member process NSAP catenated with a unique transport connection identifier. The source of any packet transmitted by a process, regardless of the packet's destination, is a member of this set.

[3]Determining this multicast NSAP for a given instantiation is not a function performed by MTP.

3.2 Sequencing Messages

The *agreement and ordering* layer of MTP ensures that all processes agree on which messages are accepted and in what order they are accepted. Let p_i be a member process and M_i be the sequence of messages that p_i has delivered to its client. The agreement layer ensures the following two properties:

AB–1 The sequence of messages that processes have delivered to the clients do not diverge; that is, for all processes p_i and p_j, M_i is a prefix of M_j or M_j is a prefix of M_i.

AB–2 There exists a connected subset of the nonfaulty (*i.e.* noncrashed) processes that make progress.

Figures 1, 2 and 3 shows pseudocode for the MTP agreement and ordering protocol. In these figures, the primitive **send** p x sends the message x to process p without blocking, the primitive **receive** p x is a CSP–like guard [Hoa78] that receives a matching message from process p and stores it into x, and **multicast** P x multicasts the message x to the processes in the set P without blocking. The predicate failed(s) represents a timeout; it will become true at some point after the processor that was issued the token for message number s has crashed or remained partitioned away from the master.

To send a message m to a web, a member process first requests one of a set of t tokens from the master of the web. This token contains:

- the message number to be assigned to m,
- the multicast transport addresses as discussed in Section 3.1,
- the status of the last t messages. Such messages can be *accepted*, *rejected*, or *pending*. Furthermore, the earliest of these t messages must either be *accepted* or be *rejected*.

The master sets the status of the last t messages using the following rule. Let m is one of these last t messages:

- if the master has seen the message m, then the status is *accepted* [4];

[4] The master has seen message m when it has received a data packet of message m containing an *end of message* indicator; see Section 3.3.

```
process Master
begin
    members: integer set := { ... };
    status (1.. t): Status := undefined, ..., undefined;
    t: constant integer; the number of tokens
    next: integer := 1;
    s: integer;
    m: Data;
    last (1.. t): Status;
    do receive Sender(i: 1..n) ["token_request"] and status(t − 1) ≠ pending →
            begin
            status(1.. t) := pending, status(1.. t − 1);
            next := next + 1;
            send Sender(i) ["token_grant", next, status, members]
            end
    [] receive Receiver(i: 1..n) ["data", s, m, last] →
            if next − s ≤ t and status(next − s) = pending
                then status(next − s) := accepted;
    [] failure(s) and next − s ≤ t
            and status(next − s) = pending → status(next − s) = rejected
    od
end
```

Figure 1: Agreement Protocol for Web Master

- if the master has not seen the message m but the sender of m is still operational and connected to the master (as determined by the master), then the status is *pending*;
- otherwise, the status is *rejected*.

An abbreviated proof of this protocol is presented in the Appendix. Informally, the specification is met because the behavior of the web is defined by the behavior of the master. In particular, a member process accepts a message m only if the master accepts m, and all messages are accepted in the order of their message sequence numbers; thus, **AB–1** is met. We define the connected subset of correct processes referred to in **AB–2** as those processes S that remain connected to the master. The master will accept messages sent by processes in S and possibly reject other messages, and the

```
process Sender(i: 1.. n)
begin
    last (1.. t): Status;
    members: integer set;
    s: integer;
    m: Data;
    do receive producer(i) [m] →
            begin
            send Master ["token_request"];
            receive Master ["token_grant", s, last, members];
            multicast {Master} ∪ Receiver(members) ["data", s, last, m]
            end
    od
end
```

Figure 2: Agreement Protocol for Web Producer

members of S will in turn accept and reject these same messages as other messages are sent[5].

Having obtained a token, s multicasts message m with the token for message m included in the header of the data packets that carry m. Processes learn the status of earlier messages by seeing such packets, and can accept and reject messages accordingly. This protocol can tolerate up to a sequence of t failures; if there are $t+1$ failures, then the master could send tokens to these processes which could then fail before any nonfaulty process sees any data sent with these $t+1$ tokens. The headers of these tokens carry information about the status of earlier messages, and since no other process received any data sent with the earliest token, the status of some message will never be propagated to the members of the web.

3.3 Sending Messages

The transport multicast layer of MTP is implemented using the multicast capability provided by the network layer (which in turn is provided by the data link and physical layers). For the purposes of this paper, we assume

[5]As written, a message can be acknowledged (and hence delivered) only when another message is sent. However, the master can send *empty packets*, defined in Section 3.3, in order to expedite the delivery of a message when subsequent messages are slow in being generated.

```
process Receiver(i: 1.. n)
begin
    data (1.. ): Data := empty .. ;
    status (1.. ): Status := pending .. ;
    nextIn, nextOut: integer := 1, 1;
    last (1.. t): Status;
    s: integer;
    m: Data;
    do receive Receiver(j: 1..n) ["data", s, last, m] →
            begin
            k: integer := 2;
            data(s − 1) := m;
            do k ≤ t → status(s − k) := last(k); k := k + 1 od;
            nextIn := max s, nextIn
            end
    [] receive consumer(i) and status(nextOut) = accepted →
            if data(nextOut) ≠ empty →
                send consumer(i) [data(nextOut)]; nextOut := nextOut + 1
            [] data(nextOut) = empty → rejoin
            fi
    [] status(nextOut) = rejected → nextOut := nextOut + 1
    [] status(nextOut) = pending and (nextIn − nextOut) > t → rejoin
    od
end
```

Figure 3: Agreement Protocol for Web Consumer

that a multicast to all of the processes in a web can be accomplished by performing multicasts to a small number of transport service access points (TSAPs)—no more than can be included in the data portion of an MTP packet. Network facilities similar to those described in [DC90] support this facility, but are not necessary for MTP to operate.

The transport layer treats a message as an uninterpreted sequence of bytes terminated by an *end of message* marker. The transport layer fragments a message into a sequence of packets. Each packet carries a *sequence number* of the form (m, p) where m is the message number and p is the packet number in this message, starting at zero. For example, if message 5 were broken into 3 packets, then the packets would be sequenced as $(5, 0), (5, 1)$

and $(5,2)$ (of which the last would carry an *end of message* marker), and the next packet would be sequenced as $(6,0)$.

There are three parameters that control the flow of data in the transport layer. They are:

- **heartbeat**: A base unit of time, in milliseconds.
- **window**: The maximum number of data packets a producer can send during any heartbeat.
- **retention**: The maximum number of heartbeats a producer must buffer packets for possible retransmissions.

Data is transmitted in a burst of packets such that no more than the current window of data packets will be sent during a single heartbeat. Every packet transmitted (including control packets) always contains the latest heartbeat, window and retention information along with the statuses of the previous t messages and the next message sequence number. If full packets are not available[6], *empty packets* will be transmitted instead (defined below). The only data packets that will be transmitted containing less than the maximum capacity will be those that mark a client state transition.

A *empty packet* is a control packet that is multicast into the web at regular intervals whenever the producer owning a token cannot transmit client data. Empty packets are sent to maintain synchronization and to advertise the maximum sequence number of the producer. Empty packets provide the opportunity for consuming processes to detect and request retransmission of missed data as well as identifying the owner of a transmit token.

If a producer receives a NAK from a consumer requesting the retransmission of one or more packets, those packets will be multicast to the entire web or to a selected subset of the multicast TSAPs. All retransmitted packets will contain the original client information and sequence number. However, the retransmitted packets will contain updated parameter information (the heartbeat, window and retention). As no more than than a full window of data messages can be sent during one heartbeat, retransmitted packets have priority over new packets during the next heartbeat.

The producer is obligated to retransmit a packet upon request for at least *retention* heartbeats after its original transmission (even after the message has been completely sent). If the producer receives a NAK from a consumer

[6]The resource being flow controlled is a packet carrying client data. Consequently, full packets provide the greatest efficiency.

process requesting the retransmission of a packet that is no longer available, the producer sends a *nak deny* to the source of the request. If the consumer cannot recover from the loss of this packet, then the consumer rejoins the web to resynchronize.

Figure 4 shows a space-time diagram of a process transmitting into a web assuming no NAKs, and Figure 5 illustrates data transmission and NAK processing.

3.4 Consistency and Joining the Web

A process p_i may become unrecoverably inconsistent with the master of the web for several reasons. The most likely reason is that p_i has partitioned away long enough from the master so that p_i missed learning the status of a message. A less likely scenario is that some process p_j transmits a message that is received by the master but not by p_i, and p_j crashes before p_i can ask for retransmission of the missed packets. In any case, when a process finds itself inconsistent with the master, it can resynchronize itself by rejoining the web.

As described in Section 3.1, the master of a web constructs the master transport address by catenating the NSAP with a locally generated unique transport connection identifier. A process that wishes to join or rejoin the web will send a *join request* message to the join transport address, and the master will answer with a *join response* carrying a source of the master transport address. Note that a rejoining process can determine whether the web is the same session with which it became inconsistent by comparing the previous and new transport connection identifiers it obtained in the *join confirm* messages.

In general, a process that repeatedly receives no *join confirm* cannot elect itself the master. Another process may follow the same reasoning in another partition, and then if the partition were to end, there would be two inconsistent webs with undesirable properties; for example, a third joining process would nondeterministically join one of the two existing webs. Any "merging" of such inconsistent webs would have to be done outside of MTP, as the semantics of such a merge would depend on the application. A better method for master selection would be for a process to know *a priori* if it were the master or not. Doing so would both guarantee that there exists only one active web with a given NSAP and would allow the master to be located on a machine that is known to be reliably available.

Having joined a web, a process p must be informed which message it should first accept. If p does not need to be given any state in order to process the next message, then the master can immediately reply to the *join request* message with a *join confirm* message containing the sequence number s of the next token the master will hand out. Then, the joining process p need only start receiving messages with sequence numbers greater than or equal to s. However, for some applications p would need to be initialized with the state of the web after all message before s have been accepted or rejected. In this case, having received a *join request*, the master will stop granting token requests and will delay sending a *join confirm* message to p until all message before s have been accepted or rejected. Then, the master can respond with the *join confirm*, p's state can be initialized (either by having the master send the state or through a protocol outside of MTP), and the master can resume granting tokens.

Figure 6 shows a space-time diagram illustrating the sequence of messages during a join with a transfer of state from the master.

4 Parameter Values

The values of *heartbeat*, *window* and *retention* can be adjusted by the transport to reflect the capability of the members, the type of application being supported and the network topology. In general, the producers will try to drive these numbers towards a higher performance level, and the consumers will try to drive these numbers towards a higher reliability level. By doing so, both are trying to optimize the quality of service.

Producers can try to improve the performance by reducing the heartbeat interval and by increasing the window size. This will have the effect of increasing the resources committed to the transport at any time. To level the resource commitment, the producer may also reduce the retention. In the worst case, a producer must commit enough storage to hold *window size* $\times$ *retention* maximum-size packets for *heartbeat* $\times$ *retention* milliseconds.

Consumers must rely on their clients to consume the data occupying the resources of the transport. The consumer transport implementation must monitor the level of committed resources in order to ensure that resources are not overcommitted. Since MTP is a NAK-based protocol, the consumer is required to inform a producer if a change in parameters is required. A consumer must be capable of committing at least t times the memory committed by a producer.

For more reliable operation, a consumer would try to extend the heartbeat interval and increase retention. This has the effect of increasing the resources needed to support the transport. To counteract this, the consumer could reduce the window.

In order to make these parameters more concrete, consider MTP running on a collection of 1-MIP workstations with local industry-standard disks, communicating over a IEEE 802.3 local area network. The heartbeat is approximately the transport time constant. Assuming that the transport can be modeled as a closed loop function, reaction to feedback into the transport should settle out in three time constants. In a transport that is constrained to a single network, the dominant cause of processing delay will most likely be the page fault resolution time. The time to service a page fault is overwhelmingly the disk access time, and for the current industry-standard disks, around 40 milliseconds is the average worst-case access time. In the worst case, this time could double in order to reclaim a dirty page. Allowing for additional overhead and scheduling delays, two times the worst case page fault resolution time should be a suitable minimum transport time constant, which is 160 milliseconds.

The window is the number of packets that can be consumed during one heartbeat. For IEEE 802.3 local area networks, the transmit time per packet is 1.2 milliseconds for a full packet of 1500 bytes. The processing time on a 1-MIP machine running Unix should be around 5 milliseconds for a full packet (where 2.5 – 3 ms of this is incurred by the operating system). Assuming that the data for the packet originated from a disk backing store and that disk service overhead is comparable to network service overhead, the resulting overhead is 11.2 milliseconds per packet, corresponding to a bandwidth of 1 Mbit/sec. During a heartbeat of 160 milliseconds 14 packets can be sent, so the maximum window would be approximately 14 packets per heartbeat.

At worst, each producer could consume 10 percent of the available network bandwidth, so MTP will not be limited by the network bandwidth. Each producer consumes about 80 percent of the consumer's processing time, so having more than one producer outstanding could saturate a consumer. However, to a point, having multiple tokens allows some producers to acquire a token shortly before it is required (presumably overlapping the transmission of an earlier message) without locking out another producer. Additionally, increasing t decreases the average message delivery time (until thrashing becomes a problem). Since the peak resource requirement scales linearly with t, a reasonable value of t would probably be two or three.

Reducing retention may introduce instability because a consumer will have less opportunity to react to missing data. Data can be missed for a variety of reasons. If constrained to the local net, the data lost due to corruption should be around one packet in 50,000[7]. Four orders of magnitude more packets are lost at receiving stations, including packet switch routers, than over physical links. The losses are usually the result of congestion and resource starvation at lower layers due to the processing of (nearly) back to back packets. One can only require that a receiving station be capable of receiving some number of back to back packets successfully, and that number must be at least greater than the window size. The probability of success can be made as high as needed by providing the receiver the opportunity to observe the data multiple times.

At worst, the receiving station detects packet loss using timers. Such timers might have a granularity of more than two orders of magnitude greater than the maximum packet transmit time. As such, the worst case is much worse than detecting data loss due to gaps in sequence numbers. When the loss is detected, the response (a NAK) is transmitted and should be received at the producing process in less than two heartbeats after the data it references was transmitted. Again, it is the detection time that dominates, not the transmission of the NAK. NAKs are also subject to loss, but the probability of delivery can be made close to one by retransmitting. In order to be able to respond to a second NAK, the minimum retention is three.

The resources committed to a transport using the above assumptions are buffers sufficient for 126 packets of 1500 bytes each, and each buffer will be committed for at least 480 milliseconds.

The parameters would be very different for a web that spans an internetwork of several LANs, and could be adjusted to accommodate the properties of the network. For example, if a producer is separated from a set of consumers by a router and the router drops a packet due to congestion then all of the consumers will simultaneously send NAKs, further aggravating the congestion. To avoid this burst of NAKs, the master could have previously set the web's retention to $f + 3$ for some positive value of f. Each NAKing consumer would then dally for some number of heartbeats between 0 and f before NAKing a missed packet. Not only would this dallying reduce the number of simultaneous NAKs by a factor of f, but most processes would probably receive the retransmission without sending a NAK.

[7]Telephone links (between routers, for example) are capable of exhibiting similar corruption rates.

5 Discussion

5.1 Number of Tokens

In Section 4, it was argued that a reasonable number of tokens would be around two or three. It isn't clear what the number of tokens should be when a web spans a larger collection of networks. On one hand, having more tokens allows more processors to pre-allocate tokens, thereby overlapping the longer round-trip message time with (hopefully) other processing. On the other hand, the maximum number of buffers increases with the number of tokens, and processors distant from the master are more likely to partition away from the master, thereby increasing the number of failures.

One can allow the master to find a balance by varying the number of tokens. This is done by logically splitting t into the two values t_{max}, which is the maximum number of tokens that can be outstanding and is the number of message statuses carried in a header, and t_{cur}, which is the *current* maximum number of tokens that can be outstanding and need be known only by the master. The number of failures that can be tolerated is determined by t_{max} (see the discussion in the Appendix). The master could then vary t_{cur} between 1 and t_{max} depending on the web performance.

5.2 Resiliency Against Failure of the Master

The main vulnerability of MTP is that the failure of the master can cause the web to fail. For some applications (*e.g.*, a stock brokerage system), such a failure could be intolerable. In this case, it would become desirable to replicate the web master. Replicating the master for high tolerance to processor failure can be done without changing MTP, but having a replicated master would be noticed by the members as an increase in the response time to a token request (and less importantly, to a join request).

All the master replicas $p_0^1, p_0^2, \ldots p_0^k$ would reside on an unpartitionable network (for example, a single local area network), guaranteeing that if a member p_1 is connected with p_0^1 and member p_2 is connected with p_0^2, then p_1 is connected with p_0^2 and p_2 is connected with p_0^1. The web's master TSAP would be a multicast address for these replicated masters.

The masters would choose one amongst themselves to be the *coordinator*, with the rest being *cohorts* [BJ87]. Any replica receiving a request would atomically broadcasts the request to all the master replicas before the coordinator would respond. Similarly, when the coordinator decides that a message becomes *accepted*, the coordinator would first atomically broadcasts this fact to all the master replicas[8]. If the coordinator were then to fail, one cohort would become the new coordinator. This new coordinator would reject all messages that it considered *pending* and start responding to master requests.

5.3 Web Membership

One issue we have not discussed in this paper is how a process can determine the current membership of a web. Knowing this information can be very useful; for example, if all the processes agree on the current web membership, then each can agree *a priori* on how work should be partitioned amongst themselves. The *group membership problem* is essentially that of having the web members agree on when a process joins the web and when a process leaves the web (either by failing, by partitioning away, or under its own volition) [Cri88,Ric90]. The difficulty with the group membership problem is that it really cannot be "solved"; since a process can fail without notifying any other process, a member of a web cannot be sure whether or not another process is currently a member. The best that can be done is to have the web members agree on the membership of the web, and accept the fact that there may be members that have crashed, and that there may be processes that, due to the asynchronism in the system, have been excluded from the web even though they have not crashed or partitioned away[9].

Group membership protocols operate by having processes monitor each other. If a process p' decides that another process p has failed, then p' uses some reliable broadcast protocol to disseminate this information to the other web members [Ric90]. A common method of detecting whether a process p has failed or not is to use low-level "alive" messages: other processes periodically expect such messages from p (perhaps as the result of periodic

[8] As stated in this paper, the only time a member learns the status of a message is when it receives a token or a data message from another member. So, if the coordinator were to notify the cohorts before granting a token, then the cohorts would be consistent. However, in the actual protocol the master may send periodic empty packets to expedite the delivery of messages. If this empty packet advertises a new status, then the coordinator must inform the cohorts

[9] Web members must be careful in the deductions they make from the purported group membership. For example, even if a process p was a member of a web through the delivery of some message m, other web members cannot assume that p actually processed any message ordered before m unless p specifically acknowledged this fact. To do otherwise would be assuming a solution exists to the *coordinated attack* problem, which is unsolvable [Gra79].

requests), and assume that p has failed if such messages cease to arrive. Once all web members agree that p has failed (even if it has not), the new web membership is defined.

Since MTP is a NAK-based protocol, there is no defined low-level "alive" protocol. A web membership protocol, however can be implemented on top of MTP as part of the application protocol. Each web member maintains a set that contains the current web membership. When a process p joins a web, p multicasts this fact to the web, and all web members (including p) add p to their membership set when they receive this message. Similarly, if a process p' decides, for any reason, that another process p has failed, then p' multicasts this fact to the web. If p' is still a member of the web when this message is delivered, then each process (including p') removes p from its membership set when it receive this message.

Such membership information is of interest to the master. As discussed in Section 3.1, the master includes a list of multicast TSAPs in a *token grant* message. This list of TSAPs covers the membership of the web as known by the master, which as currently presented may not be the same as the membership set described above. The solution in MTP is to allow a producer and receiver to execute with the master. These processes can exchange membership changes each observes–the master seeing token losses and the receiver seeing member-observed failures. By doing so, the master can remove a multicast TSAP from its list when all processes reached via that TSAP have left the web, and the producer can multicast the removal of a member process when that member loses a token.

5.4 Conclusions

MTP is a multicast transport that supports the strong conditions of agreement on delivery, agreement on order and agreement on web membership. An implementation of MTP is currently under way.

Acknowledgements Susie Armstrong is one of the architects of MTP, and her careful readings of this paper have greatly increased its clarity. Fred Schneider aided in formulating the proofs in the appendix, and Aleta Ricciardi helped clarify the issued discussed in the group membership section. Additionally, the authors would like to thank Ken Birman, Navin Budhiraja, Tushar Chandra and Patrick Stephenson for their comments on the specifications presented here and on early drafts of this paper.

Appendix: Specification and proof

This appendix presents a specification and a proof of the ordering and agreement protocol. In interest of brevity, the proof is somewhat informal and incomplete; in particular, several simple lemmas are stated and used without proof.

Let p_0 be the master process and p_1 through p_n be the member processes. The sequence of messages that p_i has delivered to its client is denoted as M_i, and we write $M_i \sim M_j$ to mean that M_i is a prefix of M_j or M_j is a prefix of M_i. Similarly, we will denote by A_i the messages that p_i has marked as *accepted* and R_i the messages that p_i has marked as *rejected*. Both R_0 and A_0 are defined, but as there is no client of the master, M_0 is not defined. The *sequence number* of a message sent with the statement **multicast** ... ["data", s, last, m] is $\mathsf{s} - 1$, which we will denote as $m.seq$ [10]. We will write $m_1 < m_2$ as shorthand for $m_1.seq < m_2.seq \wedge m_1 \in A_0 \wedge m_2 \in A_0$.

The subset of processes that are not faulty are denoted as $\mathcal{C}$. The state predicate $conn(p_i, p_j)$ is true when p_i and p_j are connected, the state predicate $send(m, p_i)$ is true when p_i sends message m, the state predicates $produce(i)$ and $consume(i)$ are true when the client on p_i requests a message to be sent and requests data respectively, and the state function $\mathcal{S}$ is a subset of the processes $p_1, p_2, \ldots p_n$.

The specification consists of two properties. The first is a *safety property*, which specifies that "bad" states do not occur, while the second is a *liveness property*, which specifies that "good" states will eventually occur.

AB–1 The sequence of messages delivered to the clients do not diverge:

$$\Box\ (\forall p_i, p_j\colon M_i \sim M_j)$$

AB–2 There exists a connected subset $\mathcal{S}$ of the correct processes $\mathcal{C}$ that make progress:

[10] Formally, any reference to m is actually a reference to $m.seq$. The values of R_i and A_i for $i > 0$ are state functions whose values are defined by the array Producer.status and Producer.data: if there exists a state in which process p_i has status$[k]$ = *accepted* and data$[k] \neq$ *empty*, then in that state $m\colon m.seq = k\colon m \in A_i$, and if there exists a state in which process p_i has status$[k]$ = *rejected*, then in that state $m\colon m.seq = k\colon m \in R_i$. We can then define $m \in M_i$ as $m \in A_i \wedge \mathrm{nextOut} > m.seq$. Similarly, the values of R_0 and A_0 are defined by the array Master.status; if in some state status$[k]$ = *accepted*, then henceforth $m\colon m.seq = \mathrm{next} - k\colon m \in A_0$, and if in some state status$[k]$ = *rejected* then henceforth $m\colon m.seq = \mathrm{next} - k\colon m \in R_0$.

$$\Box\ (\forall m, p_i, p_j\colon p_i, p_j \in \mathcal{C}\colon send(m, p_i) \wedge \Box\ (p_i, p_j \in \mathcal{S}) \Rightarrow \\ \Diamond\ m \in M_j)$$

Our assumptions are:

- All $M_i, R_i,$ and A_i are initially empty;
- the master never fails: $p_0 \in \mathcal{C}$;
- $conn(p_i, p_j)$ is an equivalence relation (*i.e.*, it is symmetric and transitive);
- unbounded fairness is followed in the selection of enabled guards *i.e.*, a guard that remains true will eventually be selected;
- clients on correct processors always continue to send messages and consume messages:

$$\Box\ \forall p_i\colon p_i \in \mathcal{C}\colon \Diamond\ produce(i) \wedge \Diamond\ consume(i)$$

Additionally, we will assume without proof that the protocol satisfies the following three lemmas:

1. The delivery of a message is monotonic:

$$L_1\colon \Box\ (\forall m, p_i\colon (m \in M_i) \Rightarrow \Box\ (m \in M_i))$$

2. A process cannot both accept and reject the same message:

$$L_2\colon \Box\ (\forall p_i\colon (m \in A_i) \Rightarrow (m \notin R_i))$$

3. Clients receive messages in message sequence number order:

$$L_3\colon \Box\ (\forall m_1, p_i\colon m_1 \in M_i \Rightarrow \\ (\forall m_2\colon m_2.seq < m_1.seq\colon m_2 \in M_i \vee m_2 \in R_i))$$

Showing Safety One can show a program satisfies a safety property E by finding a property I such that the initial conditions $Init$ imply I, I implies $\Box\, I$, and I implies E. For I, we will use the conjunct of the two predicates I_1 and I_2:

$$I_1\colon \forall m, p_i\colon (m \in R_i) \Rightarrow (m \in R_0)$$

$$I_2\colon \forall m_1, m_2, p_i\colon (m_1 < m_2 \wedge m_1 \notin M_i) \Rightarrow m_2 \notin M_i$$

Initially, all M_i are empty, making the antecedents of I_1 and I_2 both false; thus, $Init \Rightarrow I$. To show I_1 and I_2 implies **AB–1**, note that together they state that for all p_i, M_i is a prefix of M_0. Since M_i is a prefix of M_0 and M_j is a prefix of M_0, at least one of (M_i, M_j) is a prefix of the other, meaning $M_i \sim M_j$.

We now prove $I_1 \Rightarrow \Box\, I_1$. By L_1, I_1 can become false only if a member p_i rejects a message m before p_0 rejects m. For p_i to reject m, it received a *data* message from a member process p_j containing values of s and *last* such that *last*$(s - m.seq) =$ *rejected*. To send such a data message, p_j must have received from p_0 a *token grant* message containing the same values of s and *last*. By the definition of R_0, $m \in R_0$. Thus, $I_1 \Rightarrow \Box\, I_1$.

We now prove $I_2 \Rightarrow \Box\, I_2$. I_2 can become false only if the expression $m_1, m_2, p_i\colon (m_1 < m_2 \wedge m_1 \notin M_i) \wedge m_2 \in M_i$ becomes true for some messages m_1 and m_2 and member process p_i; that is, p_i delivers a message m_2 to its client but has not (yet) delivered m_1 to its client, where m_1 and m_2 have both been accepted by p_0 and $m_1.seq < m_2.seq$. By L_3, we know that $m_1 \in M_i \vee m_1 \in R_i$, and since by assumption $m_1 \notin M_i$, we know that $m_1 \in R_i$. By I_1, we know that $m_1 \in R_0$, but since $m_1 < m_2$ we know that $m_1 \in A_0$. This is a contradiction (it violates L_2), so $I_2 \Rightarrow \Box\, I_2$.

Showing Liveness To show liveness, we will first assume that for the master $\Box(\mathrm{t} > \mathrm{next})$ which implies $\Box(\mathrm{status}(\mathrm{t}-1) = \textit{null})$. We will then show the effects when t is assumed to have a more reasonable value.

Property **AB–2** is expressed in terms of the set of processes $\mathcal{S}$; we will define this set as $p_i \in \mathcal{S} \equiv conn(p_0, p_i)$. Rewriting, we get

$$\begin{aligned} I_3\colon \Box\ (&\forall m, p_i, p_j\colon p_i, p_j \in \mathcal{C}\colon send(m, p_i) \wedge \\ &\Box\ (conn(p_0, p_i) \wedge conn(p_0, p_j)) \Rightarrow \Diamond\ m \in M_j) \end{aligned}$$

To show I_3, we will need the following five liveness properties, of which I_4, I_5 and I_8 imply I_3:

I_4: $\Box\ (\forall m, p_i\colon p_i \in \mathcal{C}\colon send(m, p_i) \wedge \Box\ (conn(p_0, p_i)) \Rightarrow \Diamond\ m \in A_0)$

I_5: $\Box\ (\forall m, p_j\colon p_j \in \mathcal{C}\colon m \in A_0 \wedge \Box\ (conn(p_0, p_j)) \Rightarrow \Diamond\ m \in A_j)$

I_6: $\Box\ (\forall m, p_j\colon p_j \in \mathcal{C}\colon m \in R_0 \wedge \Box\ (conn(p_0, p_j)) \Rightarrow \Diamond\ m \in R_j)$

I_7: $\Box\ (\forall m\colon \Diamond\ (m \in A_0 \vee m \in D_0))$

I_8: $\Box\ (\forall m, p_j\colon p_j \in \mathcal{C}\colon m \in A_j \wedge \Box\ (conn(p_0, p_j)) \Rightarrow \Diamond\ m \in M_j)$

For brevity, only an informal proof for I_5 will be shown. If there are no p_j that satisfy I_5, then the lemma is vacuously true, so we will assume that there is at least one such p_j, say p_k. By assumption, the producer on p_k will eventually request a message to be sent, and by finite progress p_k will eventually send a message to p_0 requesting a token. By fairness and connectivity, p_0 will eventually select the guard (status(t$-$1) = *null*). By the definition of A_0, $m \in A_0 \Rightarrow$ status(next $- m.seq$) = *accepted*, which is passed back to p_k in the *token grant* message (again by fairness and connectivity).

By finite progress, p_k will send a message containing the value of last, and since connectivity is an equivalence, any p_j is connected to p_k, and will therefore receive this message. Then, by finite progress p_j will eventually set $m \in A_j$, and the lemma holds.

The effect of letting t to be smaller than the maximum sequence number is that a nonfaulty process p_j that is connected to p_0 may not satisfy **AB–2**; in particular, I_5 and I_6 may not hold. A sequence of $f \geq t$ token requests by processes that appear to fail after having been granted a token will generate a sequence of f rejected messages. However, when p_j receives message m, it only sets the status for messages m': $m'.seq \geq$m.seq - t, so there will be some message whose status will remain *pending*. Eventually, nextIn $-$ nextOut will be greater than t and nextOut will point to the *pending* message, forcing p_j to rejoin. Thus, the algorithm is live only if there are no sequences of rejected messages with a length of $f \geq t$.

References

[BJ87] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 123–138. ACM SIGOPS, 1987.

[CLZ87] D. Clark, M. Lambert, and L. Zhang. NETBLT: A high throughput transport protocol. In *Proceedings of ACM SIGCOMM '87 Workshop*, pages 353–359, 1987.

[CM87] J. Chang and M. Maxemchuck. Atomic broadcast. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1987.

[CP88] J. Crowcroft and K. Paliwoda. A multicast transport protocol. In *Proceedings of SIGCOMM '88*, pages 247–256. ACM, August 1988.

[Cri88] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. In *Proceedings of the 18th International Conference on Fault-Tolerant Computing.* IEEE TCOS, 1988.

[Cri90] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 33(8), August 1990.

[CW89] David Cheriton and Carey Williamson. VMTP as the transport layer for high-performance distributed systems. *IEEE Communications Magazine*, pages 37–44, June 1989.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[Gra79] J. N. Gray. *Notes on Database Operating Systems.* Springer-Verlag, Munich, 1979.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[JB89] Thomas Joseph and Kenneth Birman. *Reliable Broadcast Protocols*, pages 294–318. ACM Press, New York, 1989.

[KTHB90] M. Franz Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1990.

[MS88] Keith Marzullo and Frank Schmuck. Supplying high availability with a standard network file system. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 447–455. IEEE Computer Society, June 1988.

[Ric90] Aleta Ricciardi. A formalism for fault-tolerant applications in asynchronous systems. In *Fourth SIGOPS European Workshop*, September 1990.

[Sch90] Fred B. Schneider. The state machine approach: A tutorial. *Computing Surveys*, 22(3), September 1990.

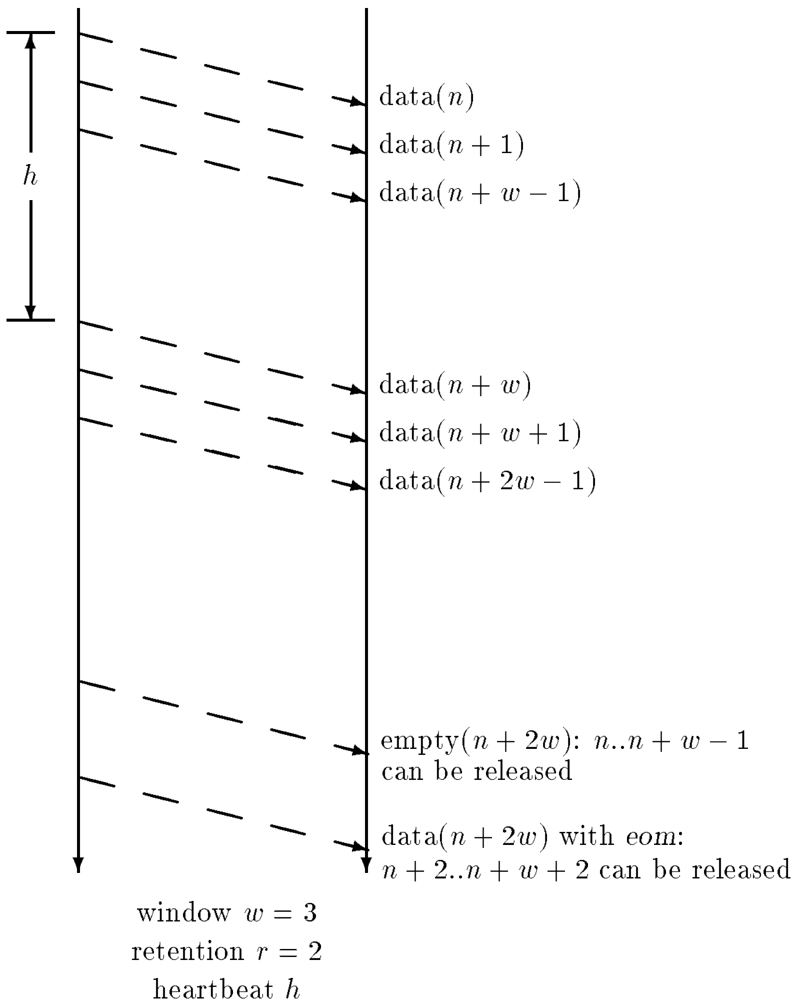

Figure 4: Normal Data Transmission

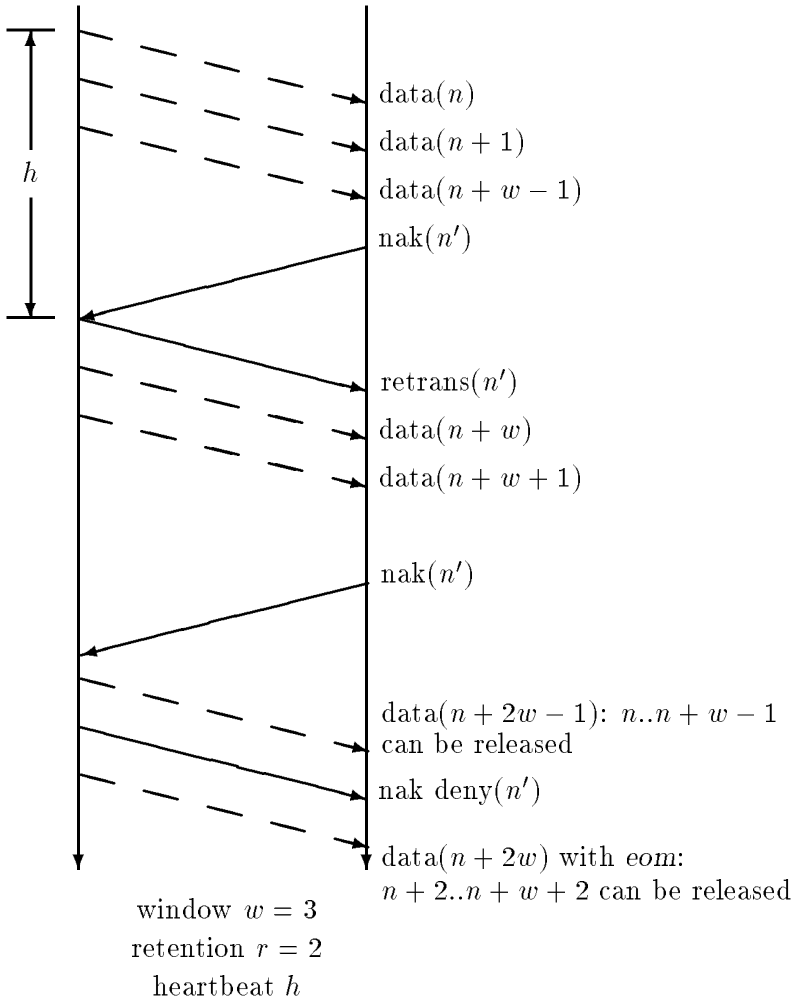

Figure 5: NAKs and Retransmission

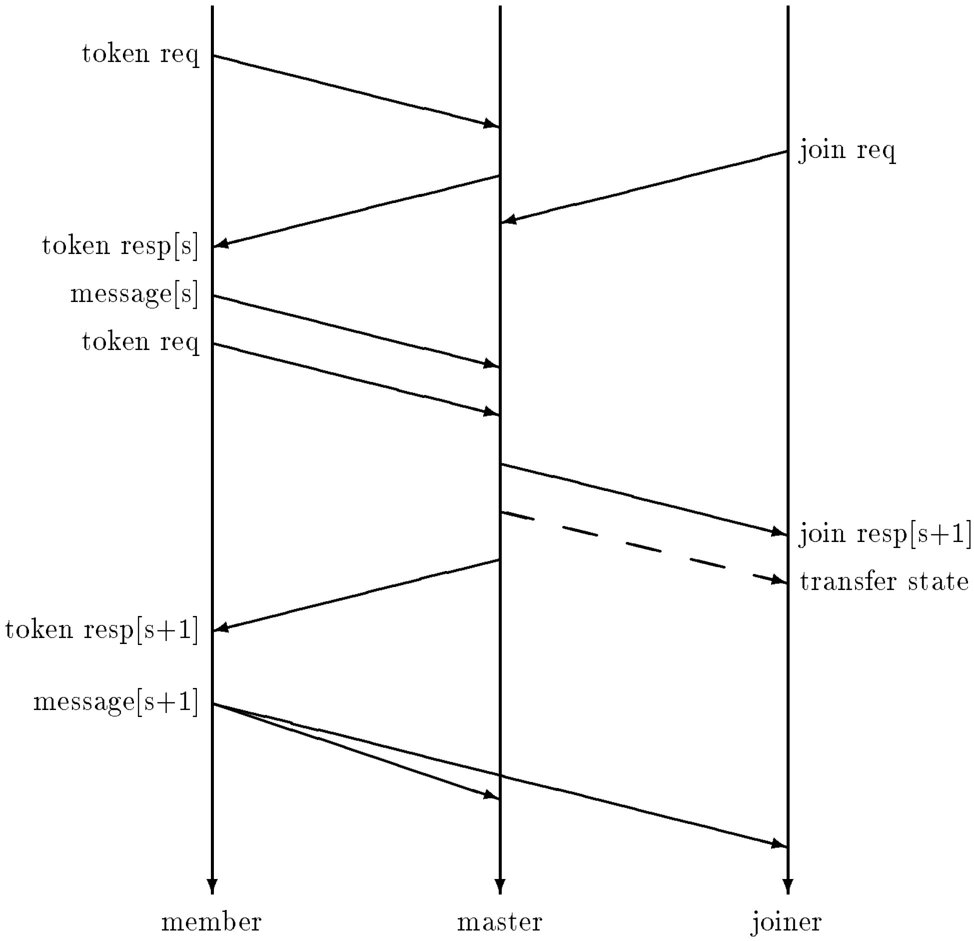

Figure 6: Joining and State Transfer