Flow Control Schemes for a Fault-Tolerant Multicast Protocol

Raimundo Macêdo, Paul D. Ezhilchelvan and Santosh K Shrivastava
Department of Computing Science, University of Newcastle,
Newcastle upon Tyne, NE1 7RU, UK

Fault-tolerant multicast protocols require that transmitted/received messages be kept locally for possible retransmission until certain stability conditions are known to have been satisfied at all the members of the group. In an asynchronous distributed system, with potentially unbounded transmission and message processing delays, the number of such 'unstable' messages at processes may grow indefinitely, leading to the possibility of buffer overflows; hence the need for an effective flow control mechanism. This paper addresses the problem of flow control in an asynchronous, symmetric, total order group communication protocol. Two algorithms that guarantee that the number of unstable messages does not exceed the stated bound are developed. They are shown to be safe *(no buffer overflows) as well as* lively *(a sender will be eventually permitted to send). Proofs of correctness and experimental results are presented.*

1. Introduction

Many fault-tolerant distributed applications can be structured as one or more groups of processes that cooperate by multicasting messages to each other. The building of such applications is considerably simplified if the members of a group have a mutually consistent view of the order in which events (such as message delivery, process failures) have taken place. A multicast protocol that delivers messages in a causality preserving total order to all functioning members of a group is thus an important component of the underlying communication system. Design and development of fault-tolerant group communication protocols for distributed systems has been therefore an active area of research [e.g., Chang84, Peterson89, Melliar-Smith90, Birman91, Amir92a, Mishra93]. Fault-tolerant group communication protocols require that a transmitted/received message be kept locally for possible retransmission until that message is known to have become *stable* ie., known to have been received by all members of the group. If we assume variable (potentially unbounded) transmission and message processing delays - which would be the case in an asynchronous distributed system - then the number of such 'unstable' messages at processes may grow indefinitely, leading to the possibility of buffer overflows; hence the need for an effective flow control algorithm.

This paper addresses the problem of flow control in an asynchronous, total order group communication protocol. We have developed two algorithms that guarantee that the number of unstable messages in any given process does not exceed the stated bound, thus preventing buffer overflows. Our algorithms are *safe* (no buffer overflows) as well as *lively* (a process that wishes to multicast a message, will be eventually permitted to carry out the multicast).

Our flow control algorithms have been designed and implemented for a total order message delivery protocol called Newtop [Ezhilchelvan 94, Macedo94]. The version of Newtop to be discussed in this paper is *symmetric* (i.e., does not rely on a sequencer process for message ordering) and has low message space overhead (the protocol related information contained in a multicast message is small). The protocol is fault-tolerant: ordering and liveness is preserved even if membership changes occur (due to failures such as process crashes and network partitions).

2. An Overview of Newtop

2.1. Background and Assumptions

In a symmetric total order protocol, when a message is received at a node, its delivery to the local process(es) may have to be delayed, as there may be concurrent messages from other members of the group in transit. Once it is certain that all such messages have been received, then these messages can be delivered. In Newtop, we make use of logical clocks [Lamport78] for total ordering of messages. Messages are assigned block-numbers (logical clock values), such that messages with the same block-number imply that they are concurrent. The principles underlying Newtop can be easily understood, if each process is visualised to maintain an array of vectors (each vector of size equal to the cardinality of the group); each such vector, called the*Causal Block,* is used for recording information about messages received with the same block-number. Once a Causal Block is detected to be *complete* (there are no more distinct messages to be received with the same block-number), the corresponding messages can be delivered to the process in some fixed order. In order to ensure that Causal Blocks eventually complete, we employ a *time-silence* mechanism that ensures that a process transmits a message now and then (only a null message, if a given process has no other message to transmit), carefully avoiding a situation where all member processes of a group continuously send only null messages.

Consider now that a process multicasting a message crashes (or gets disconnected) in the middle of the multicast, such that only some of the members receive the message. When failures do occur, the time-silence mechanism alone will not be able to make Causal Blocks complete. To ensure block completion despite failures, Newtop (in common with other protocols, e.g., [Amir92b, Mishra93]) provides another liveness mechanism: each process is associated with a local *group-view* process that can execute a *membership protocol* with its counterparts to reach agreement on the membership of the group*. Thus, if the group-view process of P_i *suspects* a failure of some remote process (P_k) that does not seem to be responding, then

---

* Process crashes should be handled ideally by a fault tolerant protocol in the following manner: when a process does crash, all functioning processes must promptly observe that crash event and agree on the order of that event relative to other events in the system. In an asynchronous environment this is impossible to achieve: when processes are prone to failures, it is impossible to guarantee that all non-faulty processes will reach agreement in finite time [Fischer85]. This impossiblity stems from a process' inability to distinguish slow processes from crashed ones. Asynchronous protocols therefore need to circumvent this impossibility result by permitting processes to *suspect* process crashes and to reach agreement only among those processes which they do not suspect to have crashed or disconnected.

the group-view process of P_i initiates a membership agreement on P_k, the outcome of which is that either processes agree to eliminate P_k from the group, with an agreement on the set of messages sent by P_k, or P_k continues to be a member of the group and P_i is able to retrieve missing messages of P_k. Thus, even if a Causal Block is complete at a process, it is necessary to keep the messages with that block-number for a while (as it may be asked to supply some of those messages to others). Once it is known that a given block is complete at all the members of the group, only then can the messages with that block-number can be discarded after delivery (such messages are said to be stable). Our flow control mechanism is therefore based on piggybacking block completion and message stability information on multicast messages.

Fortunately, in order to explain the basic principles behind our flow control mechanism, we do not need to know the precise details of the membership service or the time-silence mechanism: it is sufficient to assume the existence of liveness mechanisms that ensure that a given Causal Block will eventually complete. This also means that to describe our flow control algorithms and establish their correctness, we can ignore the consequences of process failures altogether. In the description on Newtop to be given in the next section, we will therefore only briefly mention how the membership service and the time-silence mechanism actually work (fine details of these are covered in the cited Newtop papers).

We now state the major assumptions: we consider a group $g = \{P_1, P_2, \ldots, P_n\}$ and assume that every member process P_i, $1 \le i \le n$, knows the initial membership, and member processes communicate with other members only by multicasting to the membership of the group. We assume the existence of a message transport layer permitting best effort uncorrupted and sequenced message transmission between a sender and destination processes, if the processes are alive and the destination processes are not partitioned (either due to physical network failure or message congestion) from the sender; no assumption about message transmission time is made. We assume that all processes have identical, fixed amount of local buffer spaces available for storing sent and received messages. For the sake of simplicity in exposition, we will assume that a sent or received message does not exceed a known bound on message size, and can be stored in any one of the local buffers.

In describing the flow control algorithms, we initially make the (simplifying) assumption that the application processes are "hungry" for input messages: a message delivered by the protocol is quickly consumed; modifications required to remove this assumption are stated later. In this paper, we do not consider the following issues: a process being a member of more than one group, a recovered or new process joining a group and processes forming a (new) group.

2.2. Newtop Algorithm

We first consider a static, failure-free environment where membership changes do not occur. In Newtop, each member process P_i maintains a logical clock called the *Block Counter* denoted as BC_i. BC_i is an integer variable and its value can only increase monotonically. When *g* is created BC_i of every P_i is initialised to -1. Before P_i multicasts a message *m*, it advances BC_i by 1. The contents of incremented BC_i is assigned to *m* as its *block number* in the message field *m.b*. As BC_i is advanced by 1 for every multicast, consecutively multicast messages will have increasing block-numbers.

BC_i may also be modified by P_i during *delivery*$_i$*(m)*: before delivering a multicast message *m*, P_i sets BC_i to be *m.b*, if the current value of BC_i is less than *m.b*. Thus, the two events under which BC_i may be advanced are:

CA1 (Counter Advance during send$_i$*(m))*: Before P_i multicasts *m,* it increments BC_i by one, and assigns the incremented value to *m.b*; and,

CA2 (Counter Advance during delivery$_i$*(m))*: Before P_i delivers *m,* it sets $BC_i = \max\{BC_i, m.b\}$.

Note that block-numbers of member processes are advanced in a manner similar to Lamport's logical clocks [Lamport78], with the difference that *delivery* in place of *receive* events advance BC. We can state the three following properties possessed by block-numbers of multicast messages (where '→' is used to indicate the 'happened before' relationship [Lamport78]).

pr1: $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$;

pr2: for any m, $P_j \in m.g$: $delivery_j(m) \rightarrow send_j(m'') \Rightarrow m.b < m''.b$; and,

pr3: for all m', m'': $m'.b = m''.b \Rightarrow m'$ and m'' are concurrent.

Properties pr1 and pr2 follow directly from CA1 and CA2, respectively. Together they imply that for any distinct m, m': $send(m) \rightarrow send(m') \Rightarrow m.b < m'.b$. Property pr3 states that two distinct messages multicast with same block-number are necessarily concurrent. These messages must have been multicast by distinct processes, as CA1 forbids two *send* events to occur in a given process with the same value of BC.

Each P_i maintains a vector called the *Receive Vector*, denoted as RV_i. This vector has one integer field for every $P_j \in g$; this field records the block number of the latest message received from P_j. Let D_i denote the minimum value in RV_i: $D_i = \min\{RV_i[j] \mid P_j \in g\}$. Recall that messages from a given process are sent with increasing numbers and are received in FIFO order (transport layer assumption). Therefore, $D_i \leq BC_j$ for all $P_j \in g$ and P_i is guaranteed not to receive any new m such that $m.b \leq D_i$. So P_i can 'safely' deliver all received m, $m.b \leq D_i$. The message delivery conditions for P_i in g_x are stated below:

safe1: a received *m*, is deliverable if $m.b \leq D_i$;

safe2: deliverable messages are delivered in the non-decreasing order of their block-numbers; a fixed pre-determined delivery order is imposed on deliverable messages of equal block-number.

The two *safety conditions* ensure that the received messages are delivered in total order provided they become deliverable. A received message can be guaranteed to become deliverable, only if processes in g remain *lively* by sending messages so that D_i increases with time. Newtop provides each process with a simple mechanism, called the *time-silence,* that enables a process to remain lively by sending *null messages* during those periods when the process is required to be lively but is not generating computational messages.

2.2.1. Time-silence Mechanism

The time-silence mechanism of P_i, *timesilence*$_i$, works as follows. A process P_i maintains in the interger variables $SENT_i$ and $RECD_i$, the largest block number of the messages P_i has sent and received at any given time, respectively. Whenever P_i receives a multicast message with block-number β, it checks whether $SENT_i \geq \beta$. If $SENT_i < \beta$, then a timeout, denoted as *timeout(β)*, for some predetermined time period is set. This timeout period indicates the duration within which P_i is expected to multicast a message with a block-number β or larger - thus contributing to the delivery of all *m*, $m.b \geq \beta$, by all member processes of g (including itself). When *timeout(β)* expires, P_i multicasts a null message *m* with the block-number $m.b = RECD_i$ if $SENT_i < \beta$ is still true; if $SENT_i \geq \beta$, the expiry of *timeout(β)* is ignored. Note that by multicasting a null message with block-number $m.b = RECD_i$, P_i contributes to the delivery of all m, $m.b \leq RECD_i$. Stated below is the third possibility *CA3* by which the BC_i is advanced due to $timesilence_i$.

CA3 (Counter Advance due to $timesilence_i$*):* Before P_i multicasts a null message *m*, it sets $m.b = RECD_i$ and $BC_i = m.b$.

Note that for a multicast message, block-numbers are computed using different algorithms depending on whether the message to be multicast is null or non-null. Despite this difference, property pr1 (which is: $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$) is maintained, ie. successive multicasts from P_i will have increasing block-numbers (the proof is given in [Macedo94]).

Null messages contain only protocol related information (such as block-number, destination group identifiers etc.). They are distinct and distinguishable from application-related messages, which will be called non-null messages, where distinction is required. A null message, upon being received, is treated by Newtop exactly like a non-null message except that when a null message is due for delivery, it is not supplied for processing.

From the point of view of flow control and buffer management, null and non-null messages are treated alike. Hence P_i should send a null message only if it can speed up the delivery of a non-null message. From the description of the time-silence mechanism, it can be seen that if P_i multicasts a null message then some other process in the group must have multicast a non-null message with the same block number. This implies that null messages are multicast only to enable the delivery of non-null messages in finite time.

Despite the judicial use of null messages, the time-silence mechanism can increase the message overhead of Newtop. However, the time-silence mechanism or some equivalent mechanism (such as periodic exchange of 'I am alive' or 'synchronise' messages by processes) is essential for ensuring the liveness of any *symmetric* total order protocol (see [Mostefaoui93], [Mishra93]). More importantly, it is essential for failure detection/suspicion without which a (synchronous or asynchronous) membership service cannot be built.

Finally, we observe that the timesilence mechanism of a given P_i ensures that $SENT_i = RECD_i$ becomes true in finite time if $SENT_i$ ever becomes smaller than $RECD_i$; timesilence mechanisms of all other P_j ensure (in the absence of failures) that D_i continually increases with $SENT_i$ such that if $SENT_i$ does not increase for a sufficiently long period of time then D_i may eventually become equal to (but never larger than) $SENT_i$.

2.2.2. Group Membership Service

When P_i multicasts (or delivers) a message *m* it actually does so only to (or from) those processes which it *views* as functioning members of g. Let V_i represent the current view of P_i. When g is initially formed, each functioning P_i installs an initial view = $\{P_1, P_2, ... P_n\}$. The *group-view* process of every P_i suspects failures in the following manner: it expects a message from a member process P_k, if the block-number of the last message received from P_k is less than $SENT_i$; if a message with block-number ≥ $SENT_i$ does not arrive within a predetermined timeout period, the crash/disconnection of P_k is suspected and a protocol for membership agreement on P_k is initiated.

The outcome of this execution is that all functioning and connected processes agree either to eliminate P_k from their group membership views, with an agreement on the last message sent by P_k, or to drop the suspicion on P_k and to retrieve the message from anyone who has it and to accept the "late" message of P_k. Therefore, once a process receives a message, it keeps it as long as necessary (in case it may be asked to supply a copy). Suppose that the elimination of P_k is agreed and $LAST_k$ is the block-number of the agreed last message from P_k. The *group-view* process of every P_i will do the following: when D_i becomes equal to $LAST_k$, it will remove P_k from the membership view, V_i, of P_i and P_k is eliminated from the Receive Vector RV_i. Below, we mention some features of the group membership service that are relevant to development of the flow control mechanisms:

(i) *group-view* processes of unsuspecting member processes ensure in the presence of failures that D_i continually increases with $SENT_i$ such that D_i becomes equal to, but never larger than, $SENT_i$, if $SENT_i$ does not increase for a sufficiently long period of time;

(ii) a functioning P_i is always present in its own membership view; and,

(iii) any two mutually unsuspecting processes will have identical membership views for identical values of their respective D.

3. Flow Control in Newtop

3.1. Basic Concepts

Each process is assumed to have M message buffers for running Newtop for group g = $\{P_1, P_2, \ldots , P_n\}$. $M \geq N * |g|$, and N is a positive integer that is known to all processes in group g. It will be shown that the minimum value for N is 3 and 2 for the first and second algorithms respectively. The set M of buffers can be visualised as composed of N slots of buffers, each slot capable of holding $|g|$ number of messages. Note that for any block number β, a process will need space for storing $|g|$ number of messages, so our flow control algorithms use message slots as the unit of buffer space allocation and deallocation. We assume the existence of a hash table whose addressing function takes a message block-number and generates a slot address in the interval [0, N-1].

As stated before, it is necessary to ensure that a process can always retrieve a missing message from another functioning and connected member process. This in turn means that we require a mechanism that enables a process to safely discard a received message. To develop such a mechanism, we will first define the concept of *stability*:

Definition: A block number β is said to be *stable* in P_i, if P_i knows that $D_j \geq \beta$ for all P_j in its current membership view V_i. A message with a stable block number will be termed *stable message*.

Once a message becomes stable at a P_i, then it is assured that every process in the current view of P_i will have received that message. So, a received message is discarded, and the freed space made available for reuse only after the message becomes stable. We next define a stronger stability condition, called *super-stability*:

Definition: β is said to be *super-stable* in P_i, if P_i knows that β is stable in all P_j in its current membership view V_i.

Once P_i knows that β is a super-stable block, then, it also knows that all the members in its current membership view V_i have discarded the messages with block numbers upto and including β. Thus super-stability information can be used for deducing free space at member processes.

We describe how stability and super-stability information is computed and disseminated. P_i maintains integer variables S_i and Σ_i , the largest block number that it knows to be stable and super-stable, respectively. (S_i and Σ_i are initialised to -1.) It piggybacks the values of D_i, S_i and Σ_i onto every message it multicasts, in the message fields *m.D*, *m.S* and *m.Σ* respectively. P_i maintains three vectors called the *Complete Vector*, the *Stable Vector*, and the *SuperStable Vector* denoted respectively as CV_i, SV_i and SSV_i. These vectors, like the *Receive Vector* RV_i, have one integer field for every $P_j \in V_i$; the fields $CV_i[j]$, $SV_i[j]$ and $SSV_i[j]$, $j \neq i$, respectively record the values *m.D*, *m.S* and *m.Σ* of the latest message *m* received from P_j. For a given vector XV_i, we will use XV_{mn} and XV_{mx} to denote $\min\{XV_i[j] \mid \text{all } P_j \in V_i\}$ and $\max\{XV_i[j] \mid \text{all } P_j \in V_i\}$ respectively. Whenever the contents of RV_i change (due to sending or receiving of a message), P_i computes $CV_i[i]$, $SV_i[i]$ and $SSV_i[i]$ as RV_{mn}, CV_{mn} and SV_{mn} respectively; it then computes S_i and Σ_i as SV_{mx} and SSV_{mx} respectively.

3.2. Algorithm 1

A process multicasts a message *m* only if the following three conditions hold:

fl1: *m.b* - N is super-stable, that is, $\Sigma_i \geq m.b - N$;

fl2: *m.b* - N + 1 is stable, that is, $S_i \geq m.b - N + 1$; and,

fl3: *m.b* - N + 2 is complete, that is, $D_i \geq m.b - N + 2$.

A process P_i generates free buffers by the following rule:

fb1: Any buffer that contains a stable *m* is considered to be free.

If *fl1* is true, *m* can be sent with the assurance that the slot addressed by *m.b* - N will be free at every recipient. The rationale behind *fl2* and *fl3* is as follows. By applying condition *fl2*, it is guaranteed that a received message *m*, *m.b = β*, will bring the information that at least block number β - N + 1 is stable at the sender. Since this condition will be followed by all processes, it is guaranteed that when block number β completes at P_i, block β - N + 1 will be super-stable, and block β + 2 - N is stable (thanks to *fl3*). Thus completion of a block, β, ensures that the first two conditions are true for the next block (β + 1). Fortunately, the liveness mechanisms of Newtop ensure *fl3* can always be relied upon to become true. The minimum size of a Block Matrix is therefore three. The proof of correctness (that the three conditions together implement flow control properly for N ≥ 3, i.e., buffer pool never overflows and a sender never blocks indefinitely) is given below after an illustrative example.

Fig. 1 shows the buffer pool of a process, say P_i, composed of eight buffer slots. In fig. 1(a), blocks 0, 1, and 2 are super-stable, block 3 is stable and block 4 is complete. The other blocks (from 5 to 10) have not yet been created (thus, no message space allocated). Given this situation, P_i will be able to send messages with block-numbers 5 to 10 (fig. 2(b)), but will be blocked for the sending of a message with block-number 11. For this case, the message can only be sent once block 3 become super-stable, block 4 becomes stable, and block 5 becomes complete.

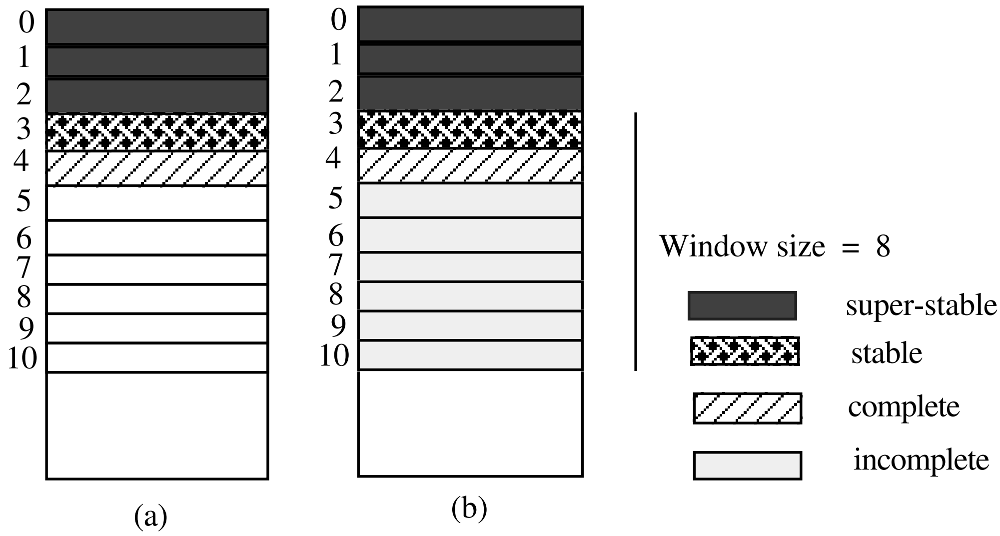

Fig. 1: **Super-stable, stable, complete and unstable messages.**

Correctness proof

We now prove the correctness of the flow control algorithm, for any $N \geq 3$.

Lemma 1: At any given time, the message buffers of P_j will have no more than N messages from P_i which are considered by P_j as unstable.

Proof: The proof is by contradiction. Suppose that P_j has N+1 unstable messages of P_i. Order these messages according to their block-numbers. If β is the largest block number, then the smallest must be at least β-N since P_i can multicast no more than one message with the same block-number. By fl1, when P_i multicast the message with block-number β, it must have had $\Sigma_i \geq \beta$ - N; that means $S_j \geq \beta$ - N. Hence the message *m, m* $\geq \beta$ - N, in P_j's buffer must be considered by P_j as stable. Hence the lemma.

Corollary 1: When P_i multicasts *m*, every (functioning and connected) member process P_j will have a free buffer to receive *m*.

Proof: By fl1, when P_i multicasts *m*, it must have had $\Sigma_i \geq \beta$ - N; that means $S_j \geq \beta$ - N. So P_j has no more than N-1 messages of P_i which it regards as unstable; by lemma 2, it can have at most N unstable messages from any other process. Since $M \geq N^*|g|$, *m* will have a free buffer.

Lemma 2: When P_i receives *m* from P_j, the following three conditions become true in finite time: $\Sigma_i \geq m.b$ - N, $S_i \geq m.b$ - N + 1 and $D_i \geq m.b$ - N + 2.

Proof: Lemma is trivially true if $m.b$ - N + 2 < 0. Suppose that all three conditions are not true when *m* is being received by P_i. When P_j sent *m*, by rules *fl1 - fl3*, $\Sigma_j \geq m.b$ - N, $S_j \geq m.b$ - N + 1 and $D_j \geq m.b$ - N + 2. Therefore, $m.\Sigma \geq m.b$ - N, $m.S \geq m.b$ - N + 1 and $m.D \geq m.b$ - N + 2. So, the first two conditions become true as soon as P_i receives *m* and recomputes its Σ_i, S_i and D_i. $D_j \geq m.b$ - N + 2 implies that P_j has received from P_i a message with block-number larger than or equal to $m.b$ - N + 2. (If P_j has not received such a message from P_i, then that must mean that P_j has removed P_i from its membership view before $D_j \geq m.b$ - N + 2 could become true - in which case P_j would not be transmitting m to P_i.) So $SENT_i \geq m.b$ - N + 2. If $D_i < m.b$ - N + 2, then P_i has not received some messages that were received by P_j. If these messages arrive, by corollary 1, P_i will have free buffers to receive them. If they do not arrive due to failures, the group-view process of P_i ensure that D_i increases to $SENT_i \geq m.b$ - N + 2. Hence the lemma.

Corollary 2: P_i does not block indefinitely, when the message to be multicast is null.

Proof: If P_i is prompted by timesilencei to send a null m, then it must have received a message m', $m'.b = m.b$. By lemma 2, P_i will be able to send m in finite time.

Lemma 3: P_i does not block indefinitely, when the message m to be multicast is non-null, provided N >2.

Proof: Suppose that P_i has received m', $m'.b \geq m.b$. The lemma is true by lemma 2. So we will assume that P_i has received no m', $m'.b \geq m.b$. If P_i has not already sent any message at all, then the non-null message m will have to be sent with $m.b = 0$ and the rules fl1 - fl3 are trivially met since N >2 and Σ_i, S_i, and D_i are all initialised to -1. Suppose that P_i has previously multicast messages. Since m is non-null, the block-number of the last message multicast by P_i must be $m.b$ -1 according to CA1. Let $\beta = m.b - 1 > 0$. The *group-view* process of P_i ensures that D_i reaches the value of β in finite time. Every m that P_i receives with $m.b \geq \beta$ will indicate that $m.\Sigma \geq \beta$ - N, $m.S \geq \beta$ - N + 1 and $m.D \geq \beta$ - N + 2. So when $D_i \geq \beta$ becomes true, $\Sigma_i \geq \beta$ - N + 1 and $S_i \geq \beta$ - N + 2 will become true - thus permitting another message to be sent with block-number $\beta + 1 = m.b$. Hence the lemma.

Theorem 1: The algorithm 1 avoids buffer overflows (safe) and ensures that processes are permitted to send new multicast messages after finite blocking.

Proof: Follows from Lemmas 1 -3.

3.3. Algorithm 2

We now introduce a variation to the previous algorithm. This algorithm does not guarantee a free buffer for an incoming message; instead, it discards the "oldest" message stored in the buffers to make space for the newly received message. A process P_i is not required to maintain Σ_i and SSV_i; all other vectors, D_i and S_i are maintained as in the first algorithm. The flow control rules are as follows:

A process P_i can multicast a message m, only if

fl1': $m.b$ - N is stable, that is, $S_i \geq m.b$ - N; and,

fl2': $m.b$ - N + 1 is complete, that is, $D_i \geq m.b$ - N + 1.

A process P_i generates free buffers by these following rules:

fb1': Initially, all buffers are assumed to contain a (non-existing) message with block-number = -1.

fb2': To store a newly received message, the buffer that contains the message with the smallest block-number is used; if there are two distinct messages with the smallest block-number, then one of them can be randomly chosen to be discarded.

The second algorithm guarantees a buffer for every incoming message. What remains to be shown is that the procedure for generating space for new messages is safe. The next lemma shows that if a message is discarded to generate a free buffer, then that message must have been delivered by all functioning member processes and therefore that message need no longer be stored for retransmission.

Lemma 4: If P_i discards μ to make space for an incoming m, then there exists a member process, say P_k, that has $S_k \geq \mu.b$.

Proof: if $S_i \geq \mu.b$, then the lemma is true. Say, $S_i < \mu.b$. Suppose that $m.b \geq \mu.b$ +N. Then (by fl1') the sender of m, say P_j, must have had $S_j \geq \mu.b$ before it multicast m. Suppose that $m.b < \mu.b$ +N. The rule fb2' confirms that the message buffers of P_i have no message with block-number smaller than $\mu.b$. No process sends more than one message with the same block-number. Since M ≥ N * |g| and N is identical for all processes in g, there must exist a message m' in the message buffers of P_i, such that $m'.b \geq \mu.b$ +N. By fl1', the sender of m', say P_k, must have had $S_k \geq \mu.b$ before it multicast m'. Hence the lemma.

Lemma 5: A functioning P_i is not infinitely blocked from sending its (null or non-null) messages.

Proof-Sketch: Lemma 4 indicates that a received message is discarded only after it is delivered, and all the protocol related information contained in it have been "taken notice of". A block-number that is super-stable (in P_i) is necessarily stable (in P_i); similarly, A block-number that is stable (in P_i) is necessarily complete (in P_i). Hence, the set of conditions {fl1, fl2, fl3} of the previous algorithm is strictly stronger than {fl1', fl2', fl3} which in turn is strictly stronger than of {fl1', fl2'} this algorithm. Therefore this lemma can be seen to be true by lemmas 2 and 3.

Theorem 2: The algorithm 2 avoids buffer overflows (safe) and ensures that processes are permitted to send new multicast messages after finite blocking.

Proof: Follows from Lemmas 4 and 5.

4. Experimental Results

We have performed experiments to evaluate the effect of the flow control mechanism on the performance of Newtop. In the experiments we have a process group with six members distributed over three workstations; algorithm one was implemented. We consider the case when only one process is sending messages. In this circumstance, the time-silence mechanism of the inactive processes is responsible for the completion of blocks. The time-silence timeout for the experiments was fixed to 50 msecs. The graphs depicted in figs. 2 and 3 are for the sender process. For each run, 1000 messages of 32 bytes each were transmitted. We varied the inter-message transmission time (denoted in the graphs as transmission time) from 400 msecs to 6 msecs and calculated the maximum number of unstable blocks as well as the average delivery delays.

Figs. 2 and 3 graphically show the data collected during the experiments when the flow control mechanism was switched off and on (with N fixed to 50) respectively. It is interesting to note that when the flow control was switched on, besides limiting the number of unstable blocks to 50, the average delivery delay was also reduced for inter-message transmission times of less than and equal to 50 msecs. The explanation for this is that when the transmission rate is higher, a larger number of messages are transmitted without stability information (i.e. Σ, S and D) being updated at the sender.

This causes the number of incomplete blocks to grow quickly and thus increases the average delay overhead for message delivery.

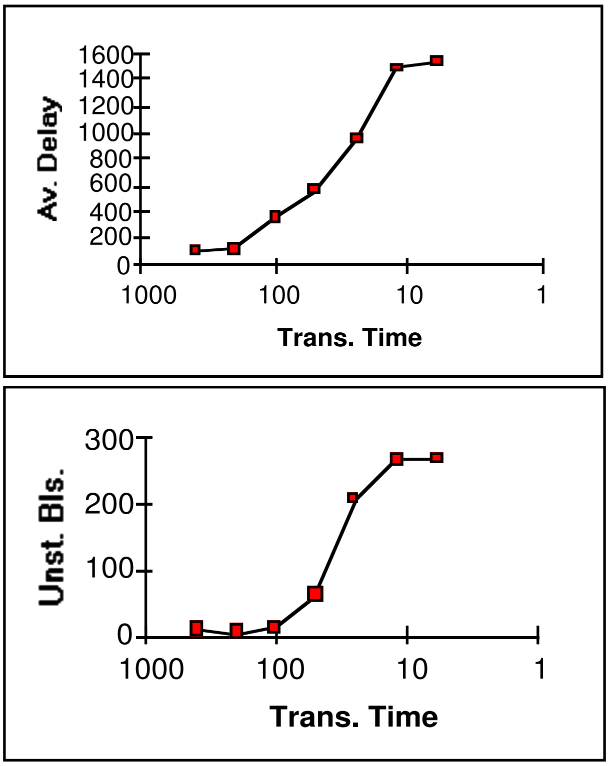

Fig. 2: 1-sender and 5-receivers with flow control switched off.

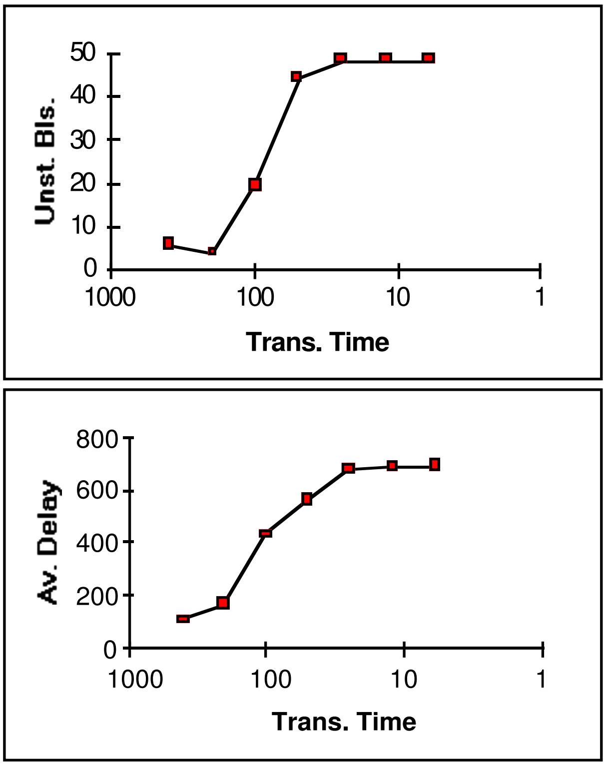

Fig. 3: 1-sender and 5-receivers with flow control switched on.

5. Enhancements

We will first remove the assumption of hungry application processes that quickly consume a delivered message. If the message delivery queue that connects the protocol delivery process to application processes is full, then a deliverable message cannot be delivered and the message delivery must wait for a delivered message to be consumed. To handle such delays, we will define Δ_i for process P_i as the maximum block-number of *delivered messages*. Note that $\Delta_i \leq D_i$. Using Δ_i instead of D_i in determining stable block-numbers will ensure that the sending of new messages is blocked sufficiently long to accommodate the delivery delay caused by slow application processes. So, $SV_i[i]$ will now be set to Δ_i and the message fields *m.D* of outgoing messages to Δ_i.

We next optimise the algorithms so that message buffers are effectively used in all situations. With the algorithms presented earlier, the message buffers available to a given process are more efficiently used only when all member processes are equally lively and null messages are seldom multicast; they will be under-used whenever the block-numbers of consecutive multicasts of any given process differ by more than one. This can happen when a process multicasts a null message to 'catch up' with other livelier processes. Consider a process P_i that has $SENT_i = 7$ and $RECD_i = 10$, because some other member process(es) had generated and multicast more messages. If P_i cannot generate three messages within some finite time interval, then it will be prompted by *timesilence*$_i$ to multicast a null message with block-number = 10. This will leave two "holes" in the buffer slots of every member process including P_i: the two buffers that would have been occupied by P_i's messages with block-numbers 8 and 9, had these messages been multicast. The algorithms do not permit any process to multicast extra messages (excess to the amount allowed by rules fl1-fl3 or fl'-fl2') even after it is known that P_i would never send messages for block-numbers 8 and 9. These holes therefore will remain unused until they disappear when the block-numbers 8 and 9 become stable, respectively.

Our algorithms can be modified so that the holes created in the buffer slots are identified and filled - thus leading to an efficient use of buffers even when processes are not equally lively. In this paper, we will only briefly present the necessary modifications (omitting the proofs of correctness). The principles are best explained in terms of causal blocks. Consider a Complete Causal Block say for β that is not full. In the above example, the Causal Block for 8, upon becoming complete, will not be full as it does not have a message from P_i. We now pack the complete Causal Block for β with messages $m.b > \beta$ and consider the full and complete Causal Block as the Causal Block for β_{mx}, β_{mx} = maximum of all *m.b* that were used to fill this causal block. Note that there will be at most one full and complete Causal Block for any given β and that block is now treated as a unit of allotment and disposal for buffer management. (When every process multicasts one message for every β then every complete Causal Block will be full; so, the unmodified algorithms are efficient in those situations.)

A process P_i uses a function called *available*(S_i, D_i, β) in order to compute the number of buffers available for messages with an incomplete block-number $m.b = \beta > D_i$. The computation is done as follows: since messages with $m.b \geq S_i$ need no longer be in the buffer, count the number of messages with unstable and complete block-numbers *m.b*, $S_i < m.b \leq D_i$, as *unstable_count*. Assume (the worst case) that the incomplete Causal Blocks will be full when they complete; so, the incomplete Causal Blocks will require *incomplete_count* = $(\beta - D_i - 1)*n$ buffers where $n = |g|$. So, (M - *unstable_count* - *incomplete_count*) will give the number of buffers available for messages with block-number = $\beta > D_i$.

At any given time, let $d_1, d_2, \ldots d_{2n}$, represent 2n messages that have been discarded *most recently* by a given process, such that $d_1.b \le d_2.b \ldots \le d_{2n}.b$. If there are only 2n-k, $0 < k \le 2n$, messages that have been discarded so far, then it will be assumed that $d_1.b = d_2.b \ldots = d_k.b = -1$. Also, let $O_1, O_2, \ldots O_n$ represent n messages that are present in the buffers and have the smallest block-numbers, such that $O_1.b \le O_2.b \ldots \le O_n.b$. If there are only n-k, $0 < k \le n$, messages in the buffers, then it will be assumed that $O_1.b = O_2.b \ldots = O_k.b = -1$.

With the rules for generating free buffers as before, the new rules for sending messages are stated below:

Algorithm 1:

A process P_i can multicast a message *m*, only if

FL0 available(S_i, D_i, β) $\ge$ n;

FL1: $d_{2n}.b$ is super-stable, that is, $\Sigma_i \ge d_{2n}.b$;

FL2: $d_n.b$ is stable, that is, $S_i \ge d_n.b$; and,

FL3: $O_n.b$ is stable, that is, $D_i \ge O_n.b$.

Since stable messages explicitly discarded in the first algorithm (see rule fb1), $d_1.b \le d_2.b \ldots \le d_{2n}.b \le S_i$. Hence FL2 above is trivially true and is presented here to help observe the similarities in rules FL1-FL3 and fl1-fl3.

Algorithm 2:

A process P_i can multicast a message *m*, only if

FL0 available(S_i, D_i, β) $\ge$ n;

FL1': $d_n.b$ is stable, that is, $S_i \ge d_n.b$; and,

FL2': $O_n.b$ is stable, that is, $D_i \ge O_n.b$.

6. Related Work

Flow control algorithms for unicast communication protocols (such as TCP/IP) have been extensively explored. Usually, these algorithms utilise the notion of *sliding windows*. In a sliding window scheme, transmitted messages are sequentially numbered before being sent to the destination. At any given time, the sender has a "window" whose size indicates the maximum number of messages it can transmit without waiting for acknowledgements from the sender. The window size is determined by the receiver following a request from the sender. If, for example, the window size is k+1, then the sender can send any message numbered from n to n + k and thus the window extends from n to n + k. As the receiver acknowledges the receipt of the sent messages, say w messages, the window slides forward to become [n + w, n + k + w]. The sequence numbers of all messages that were sent but not yet acknowledged, will be represented within the sender's window. Such messages are unstable and are kept until their reception is acknowledged by the sender. If the window is full when the sender wants to transmit a message, the sender blocks until the window slides forward.

Flow control schemes for fault-tolerant multicast protocols have not received much attention in the literature. As discussed earlier, in fault-tolerant group communication protocols, not only sent messages must be kept for possible retransmission but also any received messages. This makes flow control a significant and challenging problem to be solved. The importance of flow control as an essential requirement for the correct working of a fault-tolerant multicast protocol has been pointed out in [Amir92a]; this paper briefly outlines a flow control scheme implemented for the Transis system. To the best of our knowledge, [Cristian93] is one other paper that explicitly describes flow control schemes for the *pinwheel* multicast protocols. We will briefly compare our algorithms with theirs.

In Transis, unstable messages are kept in buffers and the flow control algorithm is designed to minimise the chances of the buffer overflow. Each process computes the size of the sliding window that consists of unstable messages. Sending of new multicasts are delayed depending on the window size: the smaller the window size, the smaller will be the delay. When the window size exceeds a maximal size, sending of messages is blocked. This will reduce the chances of buffer overflows in other processes. Group membership service is relied on to unblock the flow and to retrieve any messages lost due to 'buffer-spills'. The flow control algorithms of Newtop, in contrast, guarantee that buffer overflows never occur. This is achieved by forcing processes to observe certain completion and stability requirements before sending new messages and thereby deducing *accurate* information about message stability conditions.

Two flow control algorithms are proposed in [Cristian93]. The first algorithm, like ours, guarantees buffer overflow never occurs. It is considerably simpler than ours and this is due to certain features of the pinwheel multicast protocol: each process acts as a coordinator in a predefined ranking order. Hence if P_j is known to be acting as the coordinator then all messages it sent and ordered in the previous round are implicitly stable. The second algorithm tolerates buffer overflows which are permitted to occur in a controlled manner: when there is no free buffer, a process carefully chooses a received message and discards it to accommodate the incoming message. (Unlike in our second algorithm, the discarded message need not even be delivered, let alone be stable.) Messages discarded in this manner are considered to have been lost in transmission (an omission failure) and they are retrieved using mechanisms designed for retrieving lost messages. It is not clear from the paper, how and whether the simplicity of the first scheme and the robustness of the second will be preserved in the presence of failures. Our experience with the implementation of fault tolerant multicast protocols shows that recovery from a (real or supposed) message loss is usually costly and can lead to further message losses.

7. Concluding Remarks

Flow control is an important aspect of protocol design. However, it has not received much attention in the literature for fault-tolerant multicast protocols. We have presented two flow control algorithms for a multicast protocol that ensures that a sender process does not cause buffers to overflow from a known limit at any of the functioning destination processes. This is achieved by piggybacking block completion and block stability information (two integer values) on top of normal messages. This permits a process to compute whether it is safe to transmit a message.

Acknowledgements

This work has been supported in part by grants from the UK MOD and the Science and Engineering Research Council (Grant no. GR/H1078), ESPRIT basic research project 6360 (BROADCAST) and CNPq/Brazil (grant no. 200811/89.4).

References

[Amir92a] Amir, Y., et al, "Transis: A Communication Sub-system for High Availability", Digest of Papers, FTCS-22, Boston, July 1992, pp. 76-84.

[Amir92b] Amir, Y., Dolev, D., Kramer, S., and Malki, D., "Membership Algorithm for Multicast Communication Groups", Proc. of 6th Intl. Workshop on Dist. Algorithms, pp 292-312, November 1992.

[Birman91] Birman, K., Schiper, A. and Stephenson, P.,"Lightweight Causal and Atomic Group Multicast", ACM Transactions On Computer Systems, Vol. 9, No. 3, August 1991, pp. 272-314.

[Chang84] Chang, J. and Maxemchuk, N. F., "Reliable Broadcast Protocols", ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 251-273.

[Cristian93] Cristian, F. and S. Mishra, " The Pinwheel Asynchronous Atomic Broadcast Protocols", Technical Report 93-331, Department of Computer Science & Engineering, University of California, San Diego, USA.

[Ezhilchelvan94] Ezhilchelvan, P. E., Macedo, R.A. and Shrivastava, S. K., "Newtop: a Fault-Tolerant Group Communication Protocol", BROADCAST Project deliverable report, Vol. 2, October 1994; available from Dept. of Computing Science, University of Newcastle upon Tyne, UK.

[Fischer85] Fischer, M., Lynch N., and Paterson, M., "Impossibility of Distributed Consensus with One Faulty Process", J. ACM, 32, April 1985, pp 374-382.

[Lamport78] Lamport, L., "Time, clocks, and ordering of events in a distributed system", Commun. of ACM, 21, 7, July 1978, pp. 558-565.

[Macedo94] Macedo, R. A., "Fault-tolerant Group communication Protocols for asynchronous systems", PhD Thesis, Oct. 1994, Dept. of Computing Science, University of Newcastle upon Tyne, UK.

[Melliar-Smith90] Melliar-Smith, P. M., Moser L.E., and Agarwala, V., "Broadcast Protocols For Distributed Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990, pp. 17-25.

[Mishra93] Mishra, S., Peterson L., and Schlichting, R., "Consul: a Communications Substrate for Fault-Tolerant Distributed Programs", Distributed Systems Engineering, 1 (1993), pp. 87-103.

[Mostefaoui93] Mostefaoui, A., Raynal, M., "Causal Multicasts in Overlapping Groups: Towards a Low Cost Approach", Research Report, IRISA Campus de Beaulieu -35042 RENNES, France.

[Peterson89] Peterson, L. L., Bucholz, N. C. and Schlichting, R., "Preserving and Using Context Information in Interprocess Communication", ACM Transactions on Computer Systems, 7 (3), August 1989, pp. 217-246.