Buffer Overflow Avoidance Techniques for Group Communication Protocols[&]

Raimundo J. de Araújo Macêdo[+], Paul D. Ezhilchelvan and Santosh K. Shrivastava[*]

[+]Universidade Federal da Bahia - UFBA
Laboratório de Sistemas Distribuídos - LaSiD
Campus de Ondina, 40.170-110
Salvador, Bahia, Brasil
email: macedo@ufba.br

[*]University of Newcastle upon Tyne
Department of Computing Science
Newcastle upon Tyne, NE1 7UR, UK
email: {santosh.shrivastava, paul.ezhilchelvan}@newcastle.ac.uk

Resumo

Mensagens transmitidas ou recebidas através protocolos de comunicação de grupo (*multicast*) têm que ser armazenadas localmente para possível retransmissão, até que certas condições de estabilidade sejam satisfeitas. Num sistema distribuído assíncrono, com tempos de transmissão e processamento de mensagens não limitados, o número de mensagens 'instáveis' pode crescer indefinidamente, causando estouro de buffers. Nessas circunstâncias, novas mensagens só poderão ser recebidas descartando-se mensagens 'instáveis', comprometendo assim a capacidade de retransmissão do protocolo. Portanto, existe a necessidade de um mecanismo efetivo para o controle de fluxo de mensagens. Este artigo trata do problema de controle de fluxo no contexto de um protocolo de ordem total, assíncrono e simétrico para comunicação em grupo. São apresentados três algoritmos de crescente eficiência e que possuem as propriedades de *safety* (estouros não ocorrem) e *liveness* (um processo transmissor não é indefinidamente impedido de transmitir novas mensagens). Provas de correção e resultados experimentais são também apresentados.

Abstract

Fault-Tolerant Group Communication Protocols require that transmitted or Received Messages be kept locally for possible retransmission until certain stability conditions are known to be satisfied within a group. In an asynchronous distributed system, with potentially unbounded transmission and message processing delays, the number of such 'unstable' messages may grow indefinitely, leading to the possibility of buffer overflows whereby new messages can be received only by discarding 'unstable' messages and thus sacrificing retransmission capabilities. Hence there is a compelling need for an effective flow control mechanism. This paper addresses the problem of flow control in the context of a asynchronous, symmetric, total order group communication protocol. It presents three, increasingly efficient flow control algorithms which are shown to be safe (no buffer overflows) as well as lively (a sender will not be indefinitely blocked from sending new messages). Proofs of correctness and experimental results are presented.

Key Words: group communication, multicast protocols, flow control, network protocols, fault tolerance.

[&] A less extensive version of this paper has appeared in the Proceedings of the IEEE Pacific Rim International Symposuim on Fault-Tolerant Systems, Newport Beach. California. USA. December/1995.

1. Introduction

Many fault-tolerant distributed applications can be structured as on or more groups or processes that co-operate 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 and process failures) have taken place. A multicast protocol that delivers message is 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 (i.e., Known to have been received by all members of a group). If we assume variable (potentially unbounded) transmission and message processing delays - which would be the case in asynchronous systems - then and number of such 'unstable' messages at processes may grow indefinitely, leading to the possibility of buffer overflows whereby new messages can only be received by discarding unstable messages and thus sacrificing retransmission capabilities. Hence there is a compelling need for an affective flow control mechanism.

This paper addresses the problem of flow control in the context of an asynchronous, total order group communication protocol called Newtop [Macedo93, Macedo94, Ezhilchelvan95, Macedo95a, Macedo95b,]. We have developed three 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 also guarantees liveness: a process that wishes to multicast a message, will be eventually permitted to carry out the multicast.

The version of Newtop considered in this paper is symmetric (i.e., a single, sequencer process is not relied on for message ordering), though the low control algorithms can be easily adapted for the asymmetric version. Newtop has a low message space overhead (the protocol related information contained in a multicast message is small) and this advantage is not compromised in the development of the flow control algorithms. Newtop is also fault-tolerant: ordering and liveness is preserved even if membership changes occur (due to failures such as process crashes and network partitions). In the next section, we present an overview of Newtop which is essential to understand the workings of the flow control algorithms. Section 3 presents 3 algorithms, each one containing an improvement over the previous algorithm; these algorithms are shown to be correct. In section 4, we describe the performance of the first algorithm in the experiments carried out over an early implementation of Newtop.

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 totally ordered delivery 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 is 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 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.

In order to explain the basic principles behind our flow control mechanism, we do not need to know the full 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).

---

* 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.

We now state the major assumptions: we consider a group named g that is initially formed with distributed processes $\{P_1, P_2, \ldots, P_n\}$; we assume that when the group g is formed every member process P_i, $1 \le i \le n$, has its membership view $V_i(g)$ set to the initial group membership: $V_i(g) = \{P_1, P_2, \ldots, P_n\}$; P_i communicates with other members only by multicasting to all processes in its membership view. We also assume the existence of a message transport layer that is capable of providing uncorrupted and sequenced message transmission service between member 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. Each P_i is assumed to have M, $M > 0$, message buffers available for storing sent and received messages until those messages are known to have become stable.

We will next describe the interactions that take place between a member process P_i and the Newtop processes in the host node of P_i when P_i sends, receives and delivers a message in group g. When P_i intends to send m in g, it prompts a Newtop process $send_i$ which does not respond to P_i immediately if there is no free buffer available for storing m. Within some finite time, however, the $send_i$ process is guaranteed to find buffer space for m; following this, it performs some Newtop operations on m which include incorporating some protocol-related information into m, stores m in the local buffer, and entrusts a copy of m with the transport layer process for transmission. Once m is entrusted with the transport layer process, the event of sending m by P_i, denoted as $send_i(m)$, will be said to have occurred. When the transport layer process picks up from the network a message m destined for P_i, it sends an interrupt to P_i which is intercepted by a Newtop process called *flow-control*$_i$. Note that the interrupt will not include any protocol-related information contained in m, and may at best indicate the sender and type of m. The process *flow-control*$_i$ responds to the interrupt by providing a buffer into which m will be copied over-writing any previous contents of that buffer. The message m will be said to be *incoming* while it is in possession of the transport layer process, and will be said to have *arrived* at P_i after it has been copied into the buffer provided by *flow-control*$_i$. It is easy to see that the response of *flow-control*$_i$ must be prompt and ideally be without having to process any information that accompanies the interrupt. The arrival of a new message m will cause another Newtop process called $receive_i$ to process the protocol-related information of m. Once that is done, m will be said to have been *received* by P_i and the event $receive_i(m)$ will be said to have occurred. A received m is subsequently delivered in total order to P_i by another Newtop process called $delivery_i$. When $delivery_i$ places m in the input queue of P_i, we will say the event $delivery_i(m)$, delivery of m to P_i, has occurred.

Note that as the processing within P_i's node is asynchronous, the time period that elapses between the arrival and the reception of m cannot be bounded and known. This means that the transport layer level acknowledgements can only indicate the arrival of m at a given destination member process, but not the reception by, nor the subsequent delivery to, that member process; therefore they cannot be used in developing algorithms for *flow-control*$_i$ which should be based only on Newtop level, message-reception acknowledgements.

In describing the flow control algorithms, we will denote (for notational simplicity) the Newtop processes working for P_i as P_i itself when the meaning is obvious; i.e., we will say P_i multicasts, receives, and delivers m when $send_i(m)$, $receive_i(m)$, and $delivery_i(m)$ occur respectively. Also, we will assume that a sent or received message never exceeds a known

bound on message size, and can be stored in any one of M local buffers. We will make a simplifying assumption that every member process P_i is "hungry" for input messages and instantly consumes a delivered message when message delivery to P_i is done by placing a reference to the buffer containing the message into the input queue of P_i. (If message delivery is done by copying the message into the input queue of P_i, we require an equivalent assumption that the input queue is never full.) This assumption spares us from having to deal with the case of stable messages which cannot be discarded because the local application process is slow in taking them in for processing. Later we state modifications necessary to realise this 'hungry processes' assumption. 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 the group membership and (therefore) processes' membership views do not change. 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 before $delivery_i(m)$: 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 before $send_i(m)$): Before P_i multicasts m, it increments BC_i by one, and assigns the incremented value to $m.b$; and,

CA2 (Counter Advance before $delivery_i(m)$): 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 '$\rightarrow$' 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, and P_j, $j \neq i$: $delivery_j(m) \rightarrow send_j(m'') \Rightarrow m.b < m''.b$; 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 and m', any P_i and P_j: $send_i(m) \rightarrow send_j(m') \Rightarrow m.b < m'.b$. 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 $P_j \in V_i(g)$: this field records the block number of the latest message received from P_j. Let D_i denote the minimum value in RV_i: $D_i = \min\{RV_i[j] \mid P_j \in V_i(g)\}$. 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 $P_j \in V_i(g)$ 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 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 when they become deliverable. A received message can be guaranteed to become deliverable, only if processes in $V_i(g)$ 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 foilows. 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 making its contribution towards the delivery of all m, $m.b \leq \beta$, by all $P_j \in V_i(g)$ (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: $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$) can be observed to be maintained, ie. successive multicasts from P_i will have increasing block-numbers.

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 $RECD_i$ ever becomes larger than $SENT_i$; 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) $SENT_i$.

2.3. Group Membership Service

Suppose that D_i of P_i is less than $SENT_i$. D_i will not increase if P_i cannot receive any message from some P_k, $P_k \in V_i(g)$, because P_k has either crashed or got disconnected from P_i. Since the repair time can be unbounded, the progress of D_i can be made to resume only by excluding P_k from $V_i(g)$ and thereby excluding P_k in the computation of D_i: $D_i = \min\{RV_i[j] \mid P_j \in V_i(g)\}$. In other words, the liveness of message delivery requires that every P_i updates its $V_i(g)$ taking failures into account; P_i employs a Newtop process *group-view*$_i$ to meet this requirement. Recalling that when g is created, each functioning P_i gets installed $V_i(g) = \{P_1, P_2, \ldots P_n\}$, the *group-view*$_i$ 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 $\geq SENT_i$ does not arrive within a predetermined timeout period, the crash/disconnection of P_k is suspected and an agreement protocol over the membership of P_k is initiated.

The outcome of this agreement protocol 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 missing messages from anyone who has it and to accept them as "late" messages. 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 $V_i(g)$. Below, we mention some guarantees provided by the group membership service which are relevant to development of the flow control mechanisms:

G1 *group-view*$_i$ ensures 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; and,

G2 any two mutually unsuspecting processes will have identical membership views for identical values of their respective D.

A realistic, non-trivial implementation of the group membership service will require the following from the flow control algorithms:

(i) *flow-control*$_i$ must permit P_i to multicast a (null or non-null) message before P_i can be falsely suspected by processes in $V_i(g)$; and,

(ii) *flow-control*$_i$ must not discard a sent or arrived message before the message is received by all $P_j \in V_i(g)$.

In what follows, we will describe our flow control algorithms which are shown to meet the above requirements by satisfying the following liveness and safety conditions:

liveness: P_i will not be indefinitely blocked when it intends to multicast a (null or non-null) message; a precise bound on this blocking cannot be obtained, but a reasonably accurate estimate can be obtained and should be used in fixing the timeout for suspecting failures. (The more accurately the bound is estimated, the less will be the number of false suspicions.)

safety: *flow-control*$_i$ does not allocate or free a message buffer for storing an incoming message until the message contained in the buffer is received by all $P_j \in V_i(g)$.

In describing the flow-control algorithms we will ignore the effect of failures in group communication, with the knowledge that *group-view*$_i$ of each functioning P_i ensures in the presence of failures that D_i increases in value to become equal to $SENT_i$ whenever $SENT_i$ becomes larger than D_i.

3. Flow Control in Newtop

3.1 Basic Concepts

Recall that each process is assumed to have M message buffers for running Newtop for group g created with the intial membership $\{P_1, P_2, \dots, P_n\}$. $M \geq N * n$, where N is some positive integer that is known to all processes in group g. 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 $P_j \in V_i(g)$. A message with a stable block number will be termed *stable message*.

Once P_i knows that a message has become stable, it is assured that every process in $V_i(g)$ has received that message. So, the safest way to reuse a message buffer will be to wait until the message contained in that buffer is known to have become 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 buffers available at remote 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 $m.\Sigma$ respectively. P_i maintains three vectors called the *Complete Vect. r*, 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 $P_j \in V_i(g)$; the fields $CV_i[j]$, $SV_i[j]$ and $SSV_i[j]$, $j \neq i$, respectively record the values $m.D$, $m.S$ and $m.\Sigma$ 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 $\min\{XV_i[j] \mid \text{all } P_j \in V_i(g)\}$ and $\max\{XV_i[j] \mid \text{all } P_j \in V_i(g)\}$ 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

P_i sends m only if the following three conditions hold:

fl1.1: $m.b$ - N is super-stable, that is, $\Sigma_i \geq m.b - N$;

fl1.2: $m.b$ - N + 1 is stable, that is, $S_i \geq m.b - N + 1$; and,

fl1.3: $m.b$ - 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 and is used for storing incoming messages.

Since *fl1.1* is true, a sent*m* is assured of the availability of a free buffer at every recipient. The rationale behind *fl1.2* and *fl1.3* is as follows. By applying condition *fl1.2*, it is guaranteed that a received message m, $m.b = \beta$, 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 *fl1.3*). 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 *fl1.3* can always be relied upon to become true. The minimum size of a Block Matrix is therefore three. The proof of correctness is provided to show that the three conditions together implement flow control properly for $N \geq 3$, i.e., they guarantee that there is no buffer overflow and that a sender never blocks indefinitely. First, we give an example to illustrate the workings of the algorithm.

Fig. 1 illustrates the construction of Causal Blocks for a process P_i in a particular context of group communication; N is assumed to be 8. In fig. 1(a), blocks 0, 1, and 2 are super-stable, block 3 is stable and block 4 is complete. (Note: by rule fb1, the messages represented by the super-stable and stable blocks will be discarded by P_i.) Given this initial situation, let us suppose that P_i intends to multicast a few, say more than fourteen, messages and see how P_i can perform these multicasts without causing buffer overflow. Since the current values of Σ_i, S_i and D_i are 2, 3, and 4 respectively, P_i can send messages with block-numbers $m.b$, $5 \leq m.b \leq 10$ without being blocked by rules *fl1.1-fl1.3*, but will be blocked from sending a message with block-number larger than 10. Let us suppose that P_i multicasts six messages with $m.b$, $5 \leq m.b \leq 10$, and the figure 1(b) depicts the current situation.

Let us suppose that every other P_j in the group initially has Σ_j, S_j and D_j as 2, 3, and 4 respectively, and has no non-null message to multicast. Each P_j will be soon prompted by *timesilence*$_j$ to multicast a null *msg* with $msg.b = 10$, $msg.\Sigma = 2$, $msg.S = 3$ and $msg.D = 4$. This will cause in each process the completion of all causal blocks upto 10, the super-stabilisation of block 3 and stabilisation of block 4; this in turn allows P_i to multicast its next (non-null) message m with $m.b = 11$, $m.\Sigma = 3$, $m.S = 4$ and $m.D = 10$ (see fig. 1(c)). Prompted by *timesilence*$_j$, every other P_j will eventually multicast a null *msg* with $msg.b = 11$, $msg.\Sigma = 3$, $msg.S = 4$ and $msg.D = 10$ - thus causing the completion of causal block for 11, the super-stabilisation of 4 and stabilisation of all blocks upto 10. P_i can now send its next message with $m.b = 12$, $m.\Sigma = 4$, $m.S = 10$ and $m.D = 11$. At the completion of causal block for 12, P_i will have $\Sigma_i = 10$, $S_i = 11$ and $D_i = 12$; it can now send six more messages with $m.b$, $13 \leq m.b \leq 18$, without blocking. This situation (depicted in figure 1(d) with blocks 0 to 7 omitted) marks the completion of one cycle (of N blocks), starting from the initial situation shown in 1(a).

Observe that no process has more than N, N= 8, unstable messages of P_i at any given time. (This observation can be extended to indicate that if all processes are lively as P_i in the above example, no process will have more than N* n, $N^* n \leq M$, unstable messages at any given time.) This observation is proved in lemma 1.1; for an intutive explanation, consider the situation of figure 1 (c): P_i has $\Sigma_i = 3$, $S_i = 4$ and $D_i = 10$, and has just multicast a message with $m.b = 11$. When P_i is making the multicast, a given P_j may not have the 4th block stable (yet) as it may not have yet received all messages with $m.b = 10$. So, for a brief period following the arrival of P_i's message with $m.b = 11$, P_j will have eight unstable messages of P_i with $m.b$, $4 \leq m.b \leq 11$; once the arrived message is "seen" by the Newtop process *receive*$_j$, S_j will raise to 4 and all messages with $m.b = 4$ will be discarded. Thus, the above example illustrates that even if every other process in the group has no (non-null) message to multicast, a single process can multicast any number of messages without causing overflow and without being blocked permanently.

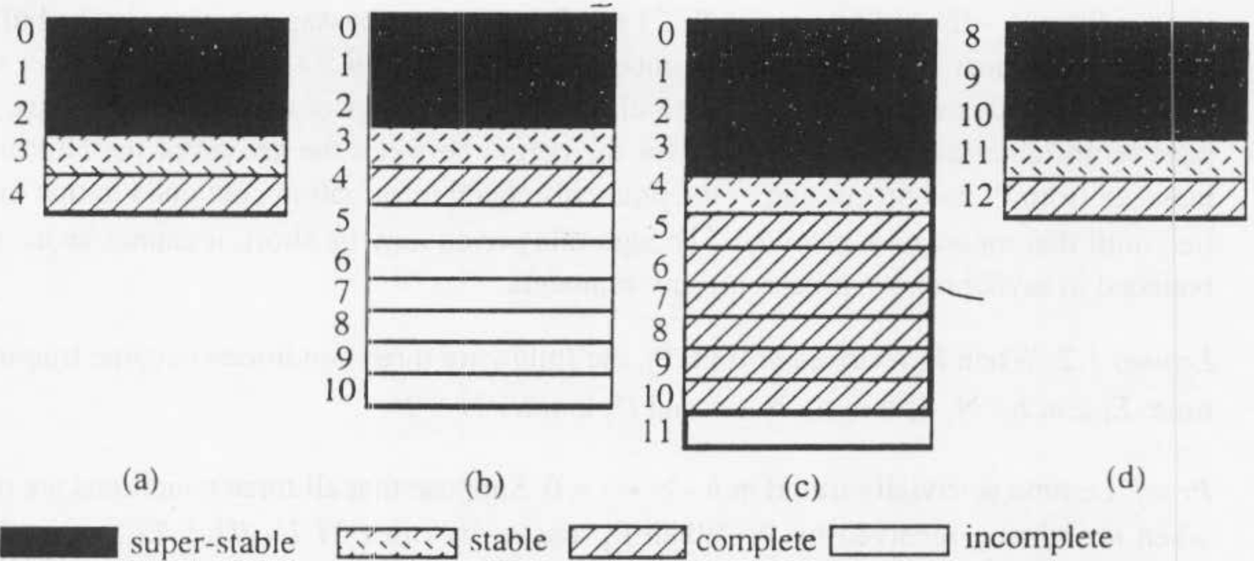

Fig. 1: Causal Blocks indicating the workings of Algorithm 1 with N = 8.

Correctness proof

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

Lemma 1.1: At any given time, the message buffers of P_j will have no more than N unstable messages from P_i.

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 *fl1.1*, 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*, $m.b \geq \beta - N$, in P_j's buffer must be considered stable by P_j. Hence the lemma.

Corollary 1.1: When P_i multicasts *m*, every P_j, $P_j \in V_i(g)$ and $j \neq i$, will have a free buffer to receive *m*.

Proof: By *fl1.1*, 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 unstable messages of P_i; by lemma 1.1, it can have at most N unstable messages from any other process. Since $M \geq N^*n$, *m* will have a free buffer in the host node of P_j.

Remarks: Every message that is being multicast is guaranteed by the rule *fl1.1* a free buffer at all (functioning and connected) destination processes. This guarantee extracts a price: a process will have, for most of the time, at most (N-1) unstable messages per remote process and will have to store N unstable messages only for a brief period of time. This may be an under-utilisation of M. $M \geq N * n$, message buffers which can accommodate at most N unstable messages per process. When the Newtop process *receive*$_j$ of P_j examines and records the protocol-related information contained in a newly arrived message *m* from P_i, S_j becomes at

least $m.S \geq m.b$ - (N-1) (due to rule *fl1.2*) and P_i's unstable messages present in the buffers of P_j will be at most N-1 with block-numbers in the range: $[m.b$ - (N-2), $m.b]$. (Also see the observation made in the example used to illustrate the workings of the algorithm.) Thus, P_j will have N unstable messages of P_i only for the period between the arrival of the Nth unstable message from P_i and processing of the protocol-related information contained in that message (ie., until that message is received). Though this period may be short, it cannot be accurately bounded in asynchronous processing environments.

Lemma 1.2: 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 *fl1.1 - fl1.3*, $\Sigma_j \geq m.b - N$, $S_j \geq m.b - N + 1$ 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 $SENT_i \geq m.b - N + 2$. (See G1 in section 2.3.) Hence the lemma.

Corollary 1.2: P_i does not block indefinitely, when the message to be multicast is null.

Proof: If P_i is prompted by *timesilence*$_i$ to send a null m, then it must have received a message m', $m'.b = m.b$. By lemma 1.2, P_i will be able to send m in finite time.

Lemma 1.3: 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 1.2. 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 *fl1.1 - fl1.3* 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. Let *last*$_i$ denote the last message multicast by P_i. Since m is non-null, *last*$_i.b$ must be $m.b$ -1 according to CA1. Let $m.b - 1 > 0$. After receiving *last*$_i$, every P_j, $P_j \in V_i(g)$, can send a message with block-number = *last*$_i.b$ (see lemma 1.2); even if P_j has no non-null message to send, the *timesilence*$_j$ will prompt P_j to send a null message. Thus, every P_j, $P_j \in V_i(g)$, eventually sends a message *msg*, $msg.b \geq last_i.b = m.b - 1$. Every *msg* that P_i receives with $msg.b \geq m.b - 1$ will indicate that $msg.\Sigma \geq m.b - 1 - N$, $msg.S \geq (m.b - 1) - (N-1)$ and $msg.D \geq (m.b - 1) - (N-2)$. So when $D_i \geq m.b - 1$ becomes true, $\Sigma_i \geq m.b - N$ and $S_i \geq m.b - (N-1)$ will become true - thus permitting another message to be sent with block-number $m.b$. Hence the lemma.

Theorem 1: The algorithm 1 ensures that a buffer will not be used for storing an incoming message until the message contained in the buffer is received by all $P_j \in V_i(g)$ (*safety*), and that P_i will not be indefinitely blocked when it intends to multicast a (null or non-null) message (*liveness*).

Proof: Follows from Lemmas 1.1 - 1.3.

3.3. Algorithm 2

The remarks that follow corollary 1 observe that the message buffers in a given process are not being used optimally by the previous algorithm. The second algorithm improves the usage of buffer spaces by removing the rule *fl1.1*. Consequently, it does not guarantee a free buffer for every incoming message; if a free buffer is not available for storing an incoming message, the "oldest" unstable message will be discarded to make space for the incoming 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

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

fl2.2: $m.b$ - N + 1 is complete, that is, $D_i \geq m.b - N + 1$.

A process P_i generates free buffers by the following rule:

fb2: Any buffer that contains a stable message is considered to be free and is preferred for storing an incoming message. When there are no free buffers, an incoming message is received using the buffer that contains a message with the smallest block-number; if there are more than one message with the smallest block-number, then one of them is randomly chosen.

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 the oldest unstable message is discarded to generate a free buffer space for storing an incoming message, then the discarded message, though unstable locally, is stable in some other process and therefore it must have been delivered by all functioning member processes and need no longer be preserved for retransmission.

Note that the oldest unstable message is discarded only when there is a pending incoming message waiting to be received.

Lemma 2.1: If P_i discards an unstable 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: That the unstable μ is evicted for receiving the incoming *m*, indicates that each one of P_i's M buffers contains an unstable message. Note that M ≥ N * n. Considering the messages in the buffers of P_i, two cases arise: (i) M = N * n and there are exactly N messages from each of the n processes in the group g (including P_i); and, (ii) there are more than N messages from some processes, say P_k.

Consider case (i). Let the incoming m was sent by P_j and let μ_j represent the oldest message of P_j that is held in the buffers of P_i (i.e., μ_j has the smallest block-number among all P_j's messages held in the buffers of P_i). The rule fb2 confirms that the message buffers of P_i have no message with block-number smaller than $\mu.b$. So, $\mu_j.b \geq \mu.b$. Recall that no process sends more than one message with the same block-number. So, even if each one of P_j's N messages in P_i's buffers had been multicast with sequentially increasing block-numbers, $\mu_j.b = m.b$ -N; this means $m.b - N \geq \mu_j.b \geq \mu.b$. By fl2.1, P_j must have had $S_j \geq m.b$ -N before it multicast m. So, $S_j \geq \mu.b$ and the lemma is true for k=j.

Let us take the case (ii). Let μ_k and m' be P_k's messages in the buffers of P_i with the smallest and the largest block-numbers respectively. Obviously $\mu_k.b \geq \mu.b$. Even if each one of P_k's more than N messages in P_i's buffers had been multicast with sequentially increasing block-numbers, $\mu_k.b \leq m'.b$ -N; this means $m'.b - N \geq \mu_k.b \geq \mu.b$. By fl2.1, P_k must have had $S_k \geq m'.b$ -N before it multicast m'. Hence the lemma.

Lemma 2.2 When P_i receives m from P_j, the following become true in finite time: $S_i \geq m.b$ - N and $D_i \geq m.b - N + 1$.

Proof-Sketch: Lemma 2.1 indicates that when P_i discards an unstable μ to make space for an incoming message, it must have $D_i \geq \mu.b$ and therefore it must have delivered μ; this means that no unstable message μ gets discarded before the protocol process of P_i have "taken notice of" all the protocol related information contained in μ. Therefore this lemma can be shown to be true by adopting similar arguments used in the proof of Lemma 1.2.

Lemma 2.3: P_i does not block indefinitely, when the message m to be multicast is non-null, provided N >1.

Proof-Sketch: The lemma can be shown to be true by adopting similar arguments used in the proof of Lemma 1.3.

Theorem 2: The algorithm 2 ensures that a buffer will not be used for storing an incoming message until the message contained in the buffer is received by all $P_j \in V_i(g)$ (*safety*), and that P_i will not be indefinitely blocked when it intends to multicast a message (*liveness*).

Proof: Follows from Lemmas 2.1, 2.2 and 2.3.

3.4. Algorithm 3

Suppose that P_i is prompted to multicast a null message with block-number, say β, and that β-1 is larger than the current value of $SENT_i$; after multicasting the null message, P_i will no longer multicast a message with $m.b$, $SENT_i < m.b \leq \beta$. As per algorithm 2 (also algorithm 1), the buffers that would have been used if P_i had multicast messages with $m.b$ in the range $SENT_i < m.b < \beta$, will remain unused. This is because the rule fl2.1 (also rules fl1.1 and fl1.2 in algorithm 1) assumes (the worst case) that P_i will *always* have N unstable and consecutively block-numbered messages stored in its buffers, and hence requires that a new m must not be sent until the message with block-number $m.b$- N gets stable and discarded. So, the 'holes'

created by null message multicasts lead to an under-utilisation of buffers; this situation is avoided in algorithm 3 where fl2.1 is replaced by a simple rule that the total number of sent messages that remain unstable must be less than N. (Note: the rules fl2.2 and fb2 are unchanged.)

To illustrate the usefulness of this optimisation, let us consider the workings of algorithm 2 in the situation depicted by figure 1(c): let us regard super-stable blocks as merely stable blocks and take N=7 (instead of 8). The scenario of figure 1(c) is as follows: P_i is the only process in the group that has non-null messages to multicast; every other P_j, $P_j \in V_i(g)$ and $j \neq i$, had multicast a null message *msg* with $msg.b = 10$, $msg.S = 3$ and $msg.D = 4$, and had not sent any message with block-number $m.b$ in the range, $5 \leq m.b \leq 9$; P_i has $S_i = 4$, $D_i = 10$, and has just multicast a message with block-number = 11. Suppose now that a given P_j generates six non-null messages to multicast. As per algorithm 2, P_j can send the first message with block-number = 11 after the 10th block completes which will happen when P_j receives a non-null m, $m.b = 10$, from P_i and null *msg*, $msg.b = 10$, from other processes; it cannot however multicast the remaining five messages with block-number $m.b$ in the range, $12 \leq m.b \leq 16$ until the 11th block completes which will cause S_j to become 10 and satisfy the rule fl2.1. This waiting for the completion of the 11th block is forced despite the fact that P_j has sent no message in the blocks numbered from 5 to 9 (inclusive) which are complete but not stable. As per the algorithm 3, P_j has sent (at this point in time) only two unstable messages (with block-number 10 and 11) and therefore can send the remaining five messages without having to wait for the completion of the 11th block; these five messages fill the holes created by P_j's multicasting of the null message *msg* with $msg.b = 10$. Thus, the algorithm 3 is effective in utilising holes when the need arises. The flow control rules of algorithm 3 are as follows:

A process P_i can multicast a message m, only if

fl3.1: less than N of the messages it had sent, are unstable; and,

fl3.2: $m.b - N + 1$ is complete, that is, $D_i \geq m.b - N + 1$.

P_i generates free buffers as in fb2 of algorithm 2.

Lemma 3.1: If P_i discards an unstable 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: That the unstable μ is evicted for receiving the incoming m, indicates that each one of P_i's M buffers contains an unstable message. Note that $M \geq N * n$. As in lemma 2.1, two cases arise: (i) $M = N * n$ and there are exacly N messages from each of the n processes in the group g (including P_i); and, (ii) there are more than N messages from some processes, say P_k.

Consider case (i). Let the incoming m was sent by P_j and let μ_j represent the oldest message of P_j that is held in the buffers of P_i (i.e., μ_j has the smallest block-number among all P_j's messages held in the buffers of P_i). The rule fb2 confirms that the message buffers of P_i have no message with block-number smaller than $\mu.b$. So, $\mu_j.b \geq \mu.b$. Note that P_j has sent (N-1) messages between sending and μ_j and m. Therefore, by fl3.1, when m was sent by P_j, μ_j must have been stable in P_j. So, $S_j \geq \mu_j.b \geq \mu.b$ and the lemma is true for k=j.

Let us take the case (ii). Let μ_k and m' be P_k's messages in the buffers of P_i with the smallest and the largest block-numbers respectively. Obviously $\mu_k.b \geq \mu.b$. Note that P_k has sent more than (N-1) messages between sending and μ_k and m'. By fl3.1, when m' was sent by P_k, μ_k must have been stable in P_k. So, $S_k \geq \mu_k.b \geq \mu.b$. Hence the lemma.

Lemma 3.2 If P_i receives m from P_j, $j \neq i$, then it can (eventually) send m', $m'.b \geq m.b$, provided $N > 1$.

Proof: When P_j sent m, by rule *fl3.2*, $D_j \geq m.b$ - (N-1). This implies that P_j has received from P_i and from every other process a message with block-number larger than or equal to $m.b$ - (N-1). So $SENT_i \geq m.b$ - (N-1). If $D_i < m.b$ - (N-1), then P_i has not received some messages that were received by P_j. When these messages do arrive, by lemma 3.1, P_i will have free buffers to receive them and will not discard them before the protocol-related information in them are recorded. So, $D_i \geq m.b$ - (N-1) eventually becomes true. Thus the rule fl3.2 is met for P_i to multicast m', $m'.b = m.b$.

That $D_k \geq m.b$ - (N-1) becomes true for every P_k (including P_i) in $V_i(g)$, implies that every P_k has $S_k \geq m.b$ - 2(N-1) because every message *msg* that P_k received with $msg.b \geq m.b$ - (N-1) will have $msg.D \geq m.b$ - 2(N-1).(See rule fl3.2.) This means that every P_k can send m' with $m'.b \geq m.b$ - (N-1) + 1, even if P_k had sent one message for each block-number in the range [$m.b$ - 2(N-1)+1, $m.b$ - (N-1)]. Since P_j has sent m, every P_k will be eventually prompted by the *timesilence*$_k$ to send a null m', $m'.b=m.b$ if there is no non-null message to multicast; thus every P_k is guaranteed to send m', $m'.b=m.b$ if it had not already sent any *msg'* in the block-number range: $m.b$ - (N-1) + 1 $\leq msg'.b < m.b$. In other words, every P_k is guaranteed to have $D_k \geq m.b$ - (N-1)+1 within some finite time and if P_i had not sent any *msg'* in the block-number range: $m.b$ - (N-1) + 1 $\leq msg'.b < m.b$, then the lemma is true straightaway. So, to summarise, $D_k \geq m.b$ - (N-1) becomes true for every P_k (including P_i) implies (i) every P_k has $D_k \geq m.b$ - (N-1)+1 within some finite time and (ii) P_i can m', $m'.b=m.b$, if it had not sent any any *msg'* in the block-number range: $m.b$ - (N-1) + 1 $\leq msg'.b < m.b$.

By repeating the above arguments we can show: $D_k \geq m.b$ - (N-1)+1 becomes true for every P_k implies (i) every P_k has $D_k \geq m.b$ - (N-1)+2 within some finite time and (ii) P_i can send m', $m.b \leq m'.b \leq m.b+1$, if it had not sent any *msg'* in the block-number range: $m.b$ - (N-1) + 2 $\leq msg'.b < m.b$. Repeating this line of arguments will lead to: $D_k \geq m.b$ - (N-1) + (N-2) = $m.b$ -1 becomes true for every P_k implies that P_i can send m', $m.b \leq m'.b \leq m.b$+(N-2). Hence the lemma.

Lemma 2.3: P_i does not block indefinitely, when a message m to be multicast is non-null, provided $N > 1$.

Proof: Suppose that P_i has received m', $m'.b \geq m.b$. The lemma is true by lemma 3.2. So we will assume that P_i has received no m', $m'.b \geq m.b$. If P_i has not already sent anv message at all, then the non-null message m will have to be sent with $m.b$ = 0 and the rules *fl3.1* and *fl3.2* are trivially met since $N > 1$, and S_i and D_i are all initialised to -1. Suppose that P_i has previously multicast messages. Let $last_i$ denote the last message multicast by P_i. Since m is non-null, $last_i.b$ must be $m.b$ -1 according to CA1. Let $m.b$ - 1 > 0. After receiving $last_i$, every P_j, $P_j \in V_i(g)$, can send a message with block-number = $last_i.b$ (see lemma 3.2); even if P_j

has no non-null message to send, the *timesilence*$_j$ will prompt P_j to send a null message. Thus, every P_j, $P_j \in V_i(g)$, eventually sends a message *msg*, $msg.b \geq last_i.b = m.b - 1$. Every *msg* that P_i receives with $msg.b \geq m.b - 1$ will indicate that $msg.D \geq (m.b - 1) - (N-1)$. So when $D_i \geq m.b - 1$ becomes true, rule fl3.2 is satisfied for P_i send *m*; also, $S_i \geq m.b - N$ becomes true which ensures that the number of unstable messages sent by P_i are at most (N-1) - satisfying the rule fl3.1 for P_i to send *m*.

Theorem 3: The algorithm 3 ensures that a message buffer will not be used for storing an incoming message until the message contained in the buffer is received by all $P_j \in V_i(g)$ (*safety*), and that P_i will not be indefinitely blocked when it intends to multicast a (null or non-null) message (*liveness*).

Proof: Follows from Lemmas 3.1, 3.2 and 3.3.

3.5. Enhancement: Hungry Application Processes

We will now 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.

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 inter-transmission time 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.

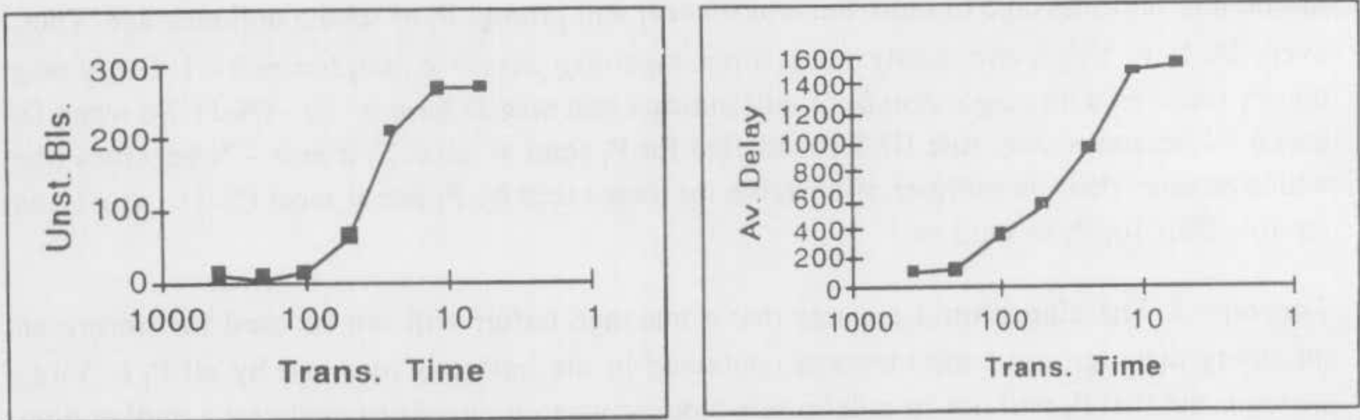

Fig. 2: 1-sender and 5-receivers with flow control switched off.

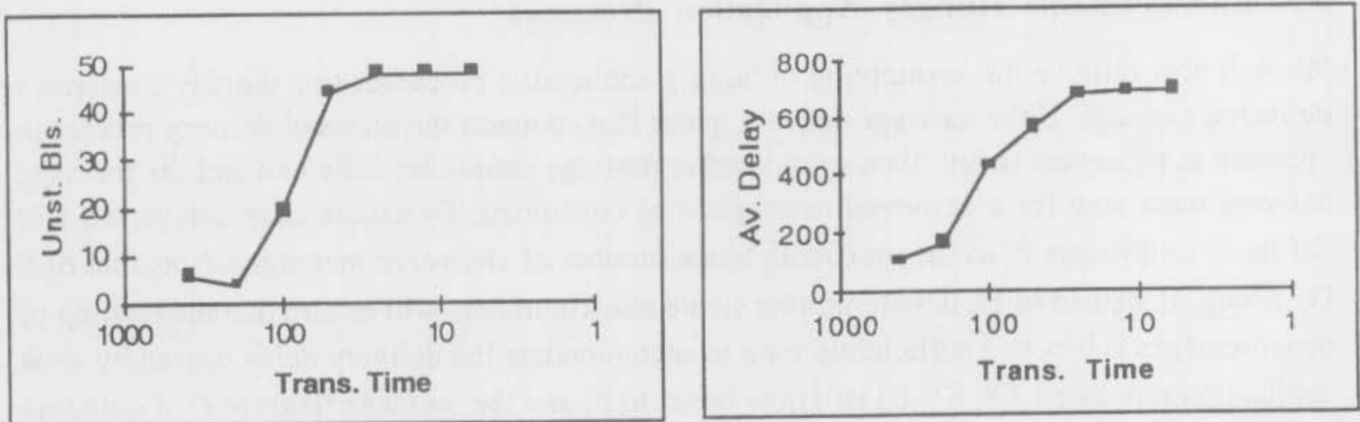

Fig. 3: 1-sender and 5-receivers with flow control switched on.

5. 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 buffer overflow. Each process computes the size of the sliding window that consists of unstable messages. Sending of new messages 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 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 overflows 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 an 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.

6. 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 three flow control algorithms for a multicast protocol that ensures that a sender process does not cause buffer to overflow from a 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.

We have implemented our first algorithm as part of an early version of Newtop. We are presently working on the integration of the third algorithm with a Group Communication Platform being developed at LaSiD-UFBA, the system BCG[1]

[1] From the Portuguese "Base Confiável de Comunicação em Grupo"

Acknowledgements

This work has been supported in part by grants from **CNPq/Brazil** (grant Nº 200811/89.4), **ESPRIT** basic research project 6360 (Broadcast), and **UK MOD** and the Engineering and Physical Sciences Research Council (GR/H1078).

References

[Amir92a] - Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki, "Transis: A Communication Sub-System for High Availability", 22nd International Symposium on Fault-Tolerant Computing (FTCS-22nd), Boston, July 1992.

[Amir92b] - Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki, "Membership Algorithm for Multicast Communication Groups", Proc. of the 6th International Workshop on Distributed Algorithm, pp 292-312, November 1992.

[Birman91] - Birman, K., Shiper 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. 2510273.

[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.

[Ezhilchelvan95] - Ezhilchelvan, Paul, Macêdo, Raimundo J. A., and Shrivastava, Santosh K., "Newtop: a Fault-Tolerant Total Order Multicast Protocol", 15th International Conference on Distributed Computing Systems, Vancouver-Canada, June 1995. IEEE.

[Fischer85] - M. Fischer, N. Lynch, and M. Peterson, "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. ACM, 21, 7 (July 1978), pp. 558-565.

[Macedo93] - Macêdo, R. J. A., Ezhilchelvan, P., and Shrivastava, S. K., "Newtop: a Total Order Multicast Protocol Using Causal Blocks", Broadcast deliverable report, Volume I, First open Broadcast workshop, Newcastle, october, 1993.

[Macedo94] - Raimundo A. Macêdo, "Fault-Tolerant Group Communication Protocols for Asynchronous Systems", Ph.D. Thesis, Department of Computing Science, University of Newcastle upon Tyne, 1994.

[Macedo95a] Raimundo J. A. Macêdo and Santosh. K. Shrivastava, "The Implementation and Performance Analysis of a Total Order Delivery Protocol for Group Communication". The Proceedings of XXI Latin American Conference on Informatics and the XV Congress of the Brazilian Computer Society. Pages 287-299, July, 1995, Canela-RS, Brazil.

[Macedo95b] Macêdo, R. J. A., Ezhilchelvan, P., and Shrivastava, S. K. "Flow Control Schemes for a Faul-Tolerant Multicast Protocols", The proceedings of the Pacific Rim International Symposuim on Faul-Tolerant Systems (PRFTS'95), December 4-5, 1995. Newport Beach, California, USA. IEEE Computer Society.

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

[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.

[Peterson89] - L. L. Peterson, N. Bucholz, and R. Schlichting, "Preserving and using context information in interprocess communication", ACM Transactions on Computer Systems, Vol. 7, No 3, August 1989, pp. 217-246