

# Newtop: A Fault-Tolerant Group Communication Protocol

Paul D Ezhilchelvan<sup>†</sup>, Raimundo A Macêdo<sup>‡</sup> and Santosh K Shrivastava<sup>†</sup>

<sup>†</sup>Department of Computing Science, University of Newcastle upon Tyne  
Newcastle upon Tyne NE1 7RU, UK

<sup>‡</sup>Federal University of Bahia, CDC/DCC, Campus de Ondina,  
40.170-110, Salvador, Bahia, Brazil

**Abstract:** A general purpose group communication protocol suite called Newtop is described. It is assumed that processes can simultaneously belong to many groups, group size could be large, and processes could be communicating over the Internet. Asynchronous communication environment is therefore assumed where message transmission times cannot be accurately estimated, and the underlying network may well get partitioned, preventing functioning processes from communicating with each other. Newtop can provide causality preserving total order delivery to members of a group, ensuring that total order delivery is preserved for multi-group processes. Both symmetric and asymmetric order protocols are supported, permitting a process to use say symmetric version in one group and asymmetric version in other.

**Key words:** group communication, group membership, fault tolerance, network protocol, multicast protocol, causal order, total order.

## 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. Design and development of fault-tolerant group communication protocols for distributed systems satisfying certain order properties has been therefore an active area of research (e.g., [4][6]12][17]). We present a contribution to this area that makes use of the concept of logical clocks [10].

We begin by describing the motivation behind our work and describe the novel features. In section three we present the basic system model and definitions and in the fourth section we develop the main concepts behind the total order protocol Newtop (NEWcastle Total Order Protocol) by considering a static failure-free environment where group membership changes do not occur. We then remove this restriction in section five and describe ways of making Newtop dynamic and fault-tolerant in the presence of process crashes, departures and arrivals and network

partitions. In section six we compare and contrast our approach with some of the best known fault-tolerant multicast protocols. Section seven concludes the paper.

## 2. Motivation

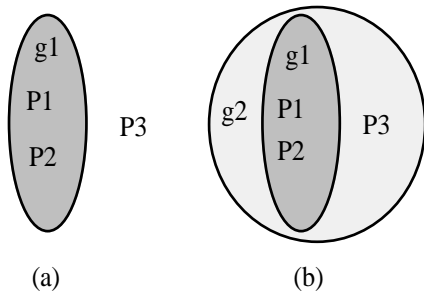
We are interested in a general purpose protocol suite that is suitable in a variety of settings. We assume that processes can simultaneously belong to many groups, group size could be large, and processes could be communicating over the Internet. We therefore model the communication environment as asynchronous, where message transmission times cannot be accurately estimated, and the underlying network may well get partitioned, preventing functioning processes from communicating with each other.

A multicast made by a process can be interrupted due to the crash of that process; this can result in some connected destinations not receiving the message. Process crashes should ideally be handled 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 [8]. This impossibility stems from the inability of a process to distinguish slow processes from crashed ones. Asynchronous protocols can circumvent this impossibility result by permitting processes to *suspect* process crashes [5] and by reaching agreement only among those processes which they do not suspect to have crashed. Despite efforts to minimise incorrect suspicions by processes, it is possible for a subgroup of mutually unsuspecting processes to wrongly agree (though rare it may be in practice) on a functioning and connected process as a crashed one, leading to a 'virtual' partition. Thus, there is always a possibility for a group of processes to partition themselves (either due to virtual or real network partitioning) into several subgroups of mutually unsuspecting processes.

A multicast protocol that delivers messages in a causality preserving total order to all functioning members

of a group is an important component of the underlying communication system. Further, like other researchers, we believe that applications will benefit if member processes are permitted simultaneously to belong to multiple groups [4, 9]. We give an example below (another example is given in the fuller version of this paper [7]).

*Online server migration:* Replica management is a well known application of total order protocols. Assume that it is necessary to migrate a member of a replicated server group to some other machine. The task is complicated because each server replica maintains substantial amount of state (say several megabytes of data), but it is required that the migration process must not cause any noticeable disruption in service or compromise availability. A possible solution will work as follows. Assume group  $g_1$  (fig. 1(a)) to be the server group, and  $P_2$  is to be migrated. A server process  $P_3$  is created at the intended location. This process initiates the formation of a new group,  $g_2$ , containing  $P_1$ ,  $P_2$  and itself (fig. 1 (b)). Within  $g_2$ ,  $P_1$  and  $P_2$  use some specific protocol for updating the state of  $P_3$  (e.g.,  $P_1$  updates the state, but if  $P_1$  fails,  $P_2$  takes over); at the same time,  $P_1$  and  $P_2$  remain responsive to clients by servicing requests directed to  $g_1$ ; eventually  $P_1$  departs from  $g_1$ , and  $P_2$  departs from both  $g_1$  and  $g_2$ , leaving  $g_2$  to be the surviving group with  $P_1$  and  $P_3$ \*. This specific solution also suggests the possibility of using multiple groups for developing a general approach for performing online software upgrades in a system (e.g., replace component  $P_2$  by  $P_3$ ).



**Fig. 1:** Multiple groups

The observations concerning failures and network partitions made earlier have motivated us to develop a membership service for Newtop that can support concurrent existence of multiple subgroups, leaving it to applications to decide on the eventual fate of such subgroups. Newtop provides causality preserving total order delivery to members of a group, and permits processes to belong to many groups, ensuring that total order delivery is preserved even for such processes. It permits a multi-group process to use both symmetric and asymmetric total order protocols in different groups.

\* It is assumed that communication level changes in group names can be hidden from applications.

Despite this, Newtop is not complex; it offers a very simple method for dealing with multiple process groups and has low message space overhead (the protocol related information contained in a multicast message is small). If order is not required, Newtop can provide just atomic delivery, ensuring that all the functioning members of a group are delivered a multicast. In addition, Newtop supports dynamic formation of new groups. This formation protocol exploits the fact that processes are permitted to belong to several groups. This in turn means that there is no need for supporting an explicit facility for joining a group (as similar effect can be obtained by processes forming a new group and exiting the previous ones).

Existing protocols do not simultaneously meet all the facilities outlined above. This has motivated us to develop Newtop.

### 3. Basic Concepts

A *group* is defined as a collection of distributed processes in which a member process communicates with other members only by multicasting to the full membership of the group. A given process can be a member of more than one group. We assume that a process execution consists of a sequence of events, each event corresponding to the execution of an action by a process. We will adopt  $send_i(m)$ ,  $receive_i(m)$  and  $delivery_i(m)$  to denote the events of sending, receiving and delivering a message  $m$  by a process  $P_i$  respectively. (We will drop suffix  $i$  if the identity of the process executing the action is not important.) We distinguish the event of *receiving* a multicast message from the event of *delivery*, since a received message may have to be delayed before delivery in order to satisfy some ordering constraint. We will use the *happened before* relation [10], denoted as ' $\rightarrow$ ', on  $send$  and  $delivery$  events in a given set of system events. (If  $send(m) \rightarrow send(m')$  then  $send_i(m) \rightarrow send_i(m')$  for some  $P_i$  or  $delivery(m) \rightarrow send(m')$ ). For simplicity, we will denote  $send(m) \rightarrow send(m')$  as  $m \rightarrow m'$ .

We assume that processes fail only by crashing, i.e., by stopping to function. Communication failures could lead to network partitions causing the members of a group to be split into disjoint subgroups, with the functioning members in one subgroup unable to communicate with the functioning members in the other sub-groups. We assume the existence of a message transport layer permitting uncorrupted and sequenced message transmission between a sender and destination processes, if the processes are alive and the destination processes are not partitioned from the sender. We assume an asynchronous communication environment, so no assumption about message transmission time will be made.

Let  $G_i$  be the set of groups  $P_i$  belongs to:  $G_i = \{g_x \mid P_i \in g_x\}$ . Let us consider the membership of  $P_i$  in a given group  $g_x$ ,  $g_x \in G_i$  and let  $g_x$  be initially made up of

processes  $P_1, P_2, \dots, P_n$ . When  $P_i$  multicasts (or delivers) a message  $m$  with  $m.g = g_x$ , it actually does so only to (or from) those processes which it *views* as functioning members of  $g_x$ .  $P_i$  delivers its own messages also by executing the protocol in operation. When  $g_x$  is initially formed, each functioning  $P_i$  installs an initial view  $V_{x,i}^0$ , say,  $V_{x,i}^0 = \{P_1, P_2, \dots, P_n\}$ . If  $P_i$  is unable to communicate with some  $P_k \in V_{x,i}^0$  (this could be because  $P_k$  has crashed or disconnected or departed from  $g_x$ ), it installs a new view that does not include  $P_k$ . Let  $V_{x,i}^0, V_{x,i}^1, V_{x,i}^2, \dots, V_{x,i}^r$  be the series of views  $P_i$  has thus sequentially installed over a period of time, until it crashes or leaves the group  $g_x$ . (Note that once  $P_i$  leaves  $g_x$ , it maintains no membership view for  $g_x$ .) Newtop provides each  $P_i$  with a *group-view* process, denoted as  $GV_{x,i}$ , for each  $g_x, g_x \in G_i$ . The group-view process  $GV_{x,i}$  makes judicious use of timeouts for suspecting the absence of member processes; it executes a *membership protocol* with other members of the group to reach agreement on these suspicions, which if confirmed lead to an update of membership view (installation of a new view) of  $P_i$  for group  $g_x$ .

In Newtop, a new view will always be a proper subset of the old view(s) since processes do not join the group they have departed. Processes wishing to join their former co-members do so by forming a new group. A process can take part in the formation of a new group while retaining its existing memberships. Newtop thus eliminates the need to support an explicit facility for process joins as provided in current group communication protocols. Former members creating a *new* group in Newtop is equivalent to the former processes of a group rejoining the same group with *new* identifiers.

The Newtop membership protocol maintains view consistency in the presence of (real or virtual) partitions by permitting a group of processes to partition themselves into two or more sub-groups of connected processes with the property that: (i) the functioning processes within any given subgroup will have identical views about the membership; and (ii) the views of processes belonging to different subgroups are guaranteed to stabilise into non-intersecting ones.

When a group partitions into subgroups, members of every subgroup will consider themselves as the sole surviving members of the original (unpartitioned) group, and will not know the existence of other subgroups and their memberships. Newtop leaves it to applications to decide whether or not the applications should continue to maintain more than one subgroup. This flexibility makes Newtop different from 'primary-partition' protocols [14, 18] that can guarantee continued group operation only when the group partitions in such a way that exactly one subgroup can be uniquely identified as the primary. This

in turn requires at least a majority of processes in the group to remain operational and connected; this requirement may not always be possible to meet.

View updates must satisfy certain conditions so that message delivery can be 'atomic' with respect to view updates. In Newtop, view updates performed by processes of a group  $g_x$  satisfy the following *view consistency* (VC) properties:

**VC1:** The sequence of views installed by any two member processes of  $g_x$  that never crash nor suspect each other are identical (*validity*).

**VC2:** If a  $P_k \in V_{x,i}^r$  leaves  $g_x$  or crashes or gets disconnected from  $P_i$  and if  $P_i$  does not crash, then  $P_i$  will eventually install  $V_{x,i}^{r'}$  such that  $r' > r$  and  $P_k \notin V_{x,i}^{r'}$  (*liveness*).

**VC3:** any two member processes of  $g_x$  that never crash, deliver the same set of messages between two consecutive views that are identical. That is,  $V_{x,i}^r \equiv V_{x,j}^r$  and  $V_{x,i}^{r+1} \equiv V_{x,j}^{r+1} \Rightarrow$  the set of  $m, m.g = g_x$ , delivered by  $P_i$  and  $P_j$  in  $V_{x,i}^r$  are identical. VC3 states that the delivery of a message to the members of a group must be atomic with respect to a view update by the members. This atomic property has been called *virtual synchrony* in the ISIS system [4].

Virtual synchrony provided by Newtop is different to that of ISIS. Consider  $P_i$  multicasting  $m$  in view  $V_{x,i}^r$ . Let this event be denoted as  $send_i(m,r)$ . Suppose that  $P_i$  delivers  $m$  in view  $V_{x,i}^{r'}$ , for some  $r' \geq r$ ; denote this event as  $delivery_i(m,r')$ . The virtual synchrony model of ISIS guarantees  $r' = r$ . Newtop can be modified to provide this closure property, but only at the necessary expense of performance, by blocking *send* operations when a new view is being installed (this blocking occurs in ISIS as well).

In the presence of member crashes and departures, Newtop has the following message delivery (MD) properties for all  $m$  and  $m'$  multicast with  $m.g = m'.g = g_x$  (in stating them the suffix  $x$  will be dropped when only the group  $g_x$  is considered). We will use the notation  $m.s$  to denote the sender of  $m$ .

**MD1 (validity):** for any  $m$  and  $r \geq 0$ :  $delivery_i(m,r) \Rightarrow m.s \in V_i^r$ . In words: a process will deliver a message  $m$  in view  $V_i^r$ , only if the sender of  $m$  is in  $V_i^r$ .

**MD2 (liveness):** for any  $m$  and  $r' \geq r \geq 0$ :  $send_i(m,r) \Rightarrow$  either  $send_i(m,r) \rightarrow delivery_i(m,r')$  or  $send_i(m,r) \rightarrow$  departure of  $P_i$  from  $g_x$ . In words: if a  $P_i$  sends  $m$  in view  $V_i^r$ , then provided it continues to function as a member of  $g_x$ , it will eventually deliver  $m$  in some view  $V_i^{r'}$ ,  $r' \geq r$ .

**MD3 (atomicity):**  $\forall P_i, P_j$  s.t.  $V_i^r \equiv V_j^r \wedge V_i^{r+1} \equiv V_j^{r+1}$ :  $delivery_i(m,r) \Leftrightarrow delivery_j(m,r)$ . This property is equivalent to VC3.

Properties MD1 to MD3 together ensure live, atomic delivery in the presence of dynamic membership changes. The additional property MD4 (and its extension for multiple groups, MD4') ensure causality preserving total order message deliveries:

**MD4** (total order, single group):  $\forall P_i, P_j$  s.t.  $V_i^r \equiv V_j^r \wedge V_i^{r+1} \equiv V_j^{r+1}$ :  $delivery_i(m,r) \rightarrow delivery_i(m',r) \Leftrightarrow delivery_j(m,r) \rightarrow delivery_j(m',r)$ ; if  $delivery_i(m,r)$  and  $delivery_i(m',r')$  occur for a given  $P_i$  then  $m \rightarrow m' \Rightarrow delivery_i(m,r) \rightarrow delivery_i(m',r')$ .

Newtop extends the above delivery order also for messages multicast in different groups, ensuring a total delivery order when the same messages are delivered to processes that simultaneously belong to multiple groups. Let  $\mu$  be a message with  $\mu.g = g_y$  and  $\rho \geq 0$  be an integer:

**MD4'** (total order, multiple groups):  $\forall P_i, P_j$  s.t.  $V_{x,i}^r \equiv V_{x,j}^r \wedge V_{x,i}^{r+1} \equiv V_{x,j}^{r+1} \wedge V_{y,i}^\rho \equiv V_{y,j}^\rho \wedge V_{y,i}^{\rho+1} \equiv V_{y,j}^{\rho+1}$ :  $delivery_i(m,r) \rightarrow delivery_i(\mu,\rho) \Leftrightarrow delivery_j(m,r) \rightarrow delivery_j(\mu,\rho)$ ; if  $delivery_i(m,r)$  and  $delivery_i(\mu,\rho)$  occur for a given  $P_i$  then  $m \rightarrow \mu \Rightarrow delivery_i(m,r) \rightarrow delivery_i(\mu,\rho)$ .

For a given delivered  $m'$ , MD5 and MD5' state the situations in which the delivery of a causally precedent  $m$ ,  $m \rightarrow m'$ , is guaranteed by Newtop:

**MD5** (causal prefix, single group): for any  $m$  and  $m'$  s.t.  $m \rightarrow m'$ :  $delivery_i(m',r') \Rightarrow delivery_i(m,r)$ .

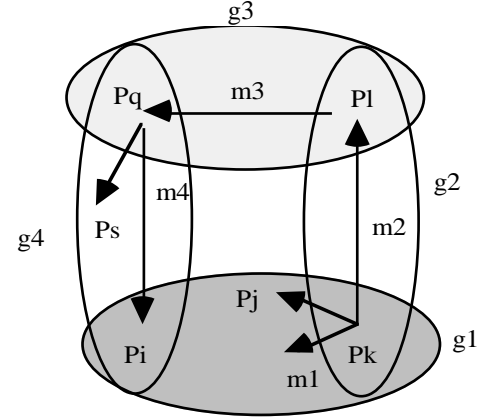
In words: if  $m'$  is delivered to  $P_i$  in view  $V_i^{r'}$  then every  $m$ ,  $m \rightarrow m'$  and  $m.g = m'.g$ , is delivered to  $P_i$  in some view  $V_i^{r'}$ . (Note that MD4 implies that  $delivery_i(m,r) \rightarrow delivery_i(m',r')$ .)

In extending MD5 to messages that are multicast in different groups, we use the notation  $V_{x,i} \ni e_i$  to denote the view  $V$  of a process  $P_i$  for  $g_x$  when the event  $e_i$  occurred in the execution sequence of  $P_i$ . For example,  $V_{x,i} \ni delivery_i(m,r)$  will be  $V_{x,i}^r$  if  $delivery_i(m,r)$  has occurred and  $m.g = g_x$ . Let  $\mu$  be a message with  $\mu.g = g_y$  and  $\rho \geq 0$  be an integer.

**MD5'** (causal prefix, multiple groups): for any  $m$  and  $\mu$  s.t.  $m \rightarrow \mu$ :  $delivery_i(\mu,\rho) \wedge m.s \in V_{m.g,i} \ni delivery_i(\mu,\rho) \Rightarrow delivery_i(m) \rightarrow delivery_i(\mu)$ .

In words: if  $\mu$  is delivered to  $P_i$  then this delivery is guaranteed to have happened after the delivery of every  $m$ ,  $m \rightarrow \mu$ , that was sent by a process in  $\mathfrak{v}$ , where  $\mathfrak{v}$  is  $P_i$ 's view for  $m.g$  when  $\mu$  is delivered to  $P_i$ .\*

To explain MD5' and its importance, we will use the scenario depicted in fig. 2 which shows a causal message chain,  $m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4$ , with  $m_1$  and  $m_4$  having a common destination  $P_i$ .

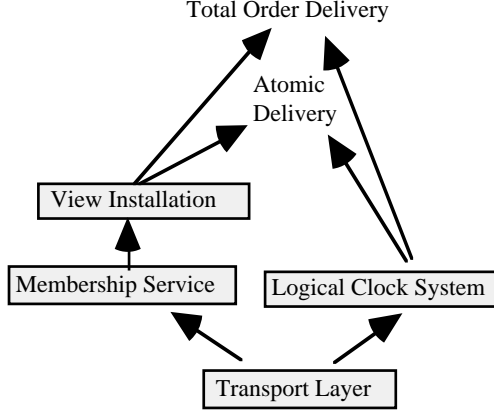


**Fig.2:** Causal chain of messages.

Suppose that a network partition disconnects  $P_k$  from  $P_i$  and  $P_j$  while  $m_1$  is being multicast, and consequently  $P_i$  and  $P_j$  do not receive  $m_1$ . Say  $P_s$  and  $P_i$  are functioning processes that never suspect each other, and  $m_4$  is delivered to  $P_s$ . MD3 requires that  $m_4$  be delivered to  $P_i$  as well. Meeting MD5' will then require any one of the following: (a)  $m_1$  is somehow retrieved and delivered to  $P_i$  before  $m_4$  is delivered; (b) if  $m_1$  cannot be retrieved (because, say the partition is permanent) then  $P_k$  should be excluded from  $P_i$ 's view for  $g_1$ , before  $m_4$  is delivered to  $P_i$ . In the latter case, the network failure that actually occurred *during* the multicast of  $m_1$ , is perceived by  $P_i$  to have happened *before* the multicast; the total ordering of events by  $P_i$  would indicate that  $P_k$  was excluded from  $P_i$ 's view of  $g_1$  *before*  $m_1$  was multicast.

The situation where  $m_4$  must be delivered and any causally preceding  $m_1$  cannot be retrieved, has arisen because of multiple overlapping groups; it would not have happened had all processes been communicating within a single group: the irretrievable loss of  $m_1$  would then mean that  $P_k$ ,  $P_l$ , and  $P_q$  are disconnected from the rest, and  $m_4$  becomes an *orphan* message that is "erased" off the system (see example 1 in section 5 and [4] for more examples). This situation can be avoided by piggybacking every multicast with all causally preceding and unstable messages. (A message is stable, if the process knows that the message has been received by the intended destinations). With piggybacking, receiving  $m_4$  will enable  $P_i$  to obtain  $m_1$ . Newtop does not adopt this expensive mechanism, preferring to use option (b) above. Consequently, it has the advantage of low message space overhead (which is even smaller than the overhead of ISIS vector clocks). MD5' specifies how situations such as depicted in fig. 4 are handled in Newtop: when the process is delivered a message, it is guaranteed that all causally preceding  $m$  have been delivered, if  $m.s$  is currently in the process's view for  $m.g$ .

\* When  $\mu.g = m.g$ , MD5' is implied by MD5.



**Fig. 3:** System architecture

Fig. 3 shows the basic system architecture of Newtop, depicting the abstraction hierarchy. The logical clock system, to be described in the next section, assigns sequence numbers to messages and is used for message ordering (note: strictly speaking, the logical clock system can be bypassed for providing just atomic delivery).

#### 4. Newtop without failures

We first consider a static, failure-free environment where membership changes do not occur. In such an environment, the functionality provided by the transport layer will be that of atomic delivery, so we need only concentrate on the provision of total order delivery.

##### 4.1 Symmetric total order version

To start with, we will consider only a single group,  $g_X = \{P_1, P_2, \dots, P_n\}$  and as stated above, assume that no  $P_i$ ,  $1 \leq i \leq n$ , ever fails or leaves  $g_X$ . This means that the initial membership view of  $P_i$  is  $g_X$  and that  $P_i$  installs no other view. So, if  $V_{X,i}$  denotes the (current) membership view of  $P_i$  at any given time, then  $V_{X,i} = V_{X,i}^0 = g_X$ . Each  $P_i$  maintains a logical clock (a counter) denoted as  $LC_i$ , that is used for numbering messages as in [10]:

**CA1 (Counter Advance during send<sub>i</sub>(m)):** Before sending  $m$ ,  $P_i$  increments  $LC_i$  by one, and assigns the incremented value to the message number field  $m.c$ ; and,

**CA2 (Counter Advance during receive<sub>i</sub>(m)):** When  $P_i$  receives  $m$ , it sets  $LC_i = \max\{LC_i, m.c\}$ .

Based on CA1 and CA2, the following two properties can be stated:

**pr1:**  $send_i(m) \rightarrow send_j(m') \Rightarrow m.c < m'.c$ ; and

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

Together these two properties imply that for any distinct  $m, m'$ :  $send(m) \rightarrow send(m') \Rightarrow m.c < m'.c$  [10].

Each  $P_i$  maintains a vector called the *Receive Vector*, denoted as  $RV_{X,i}$ . This vector has one integer field for every  $P_j \in V_{X,i}$ ; this field records the counter value of the latest message received from  $P_j$ . Let  $D_{X,i}$  denote the minimum value in  $RV_{X,i}$ :  $D_{X,i} = \min\{RV_{X,i}[j] \mid P_j \in V_{X,i}\}$ . As  $V_{X,i}$  includes  $P_i$ ,  $D_{X,i} \leq LC_i$  at any given time. Recall that messages from a given process are sent with increasing numbers and are received in FIFO order (transport layer assumption). Therefore,  $D_{X,i} \leq LC_j$  for all  $P_j \in V_{X,i}$  and  $P_i$  is guaranteed not to receive any new  $m$  such that  $m.c \leq D_{X,i}$ . So  $P_i$  can 'safely' deliver all received  $m$ ,  $m.c \leq D_{X,i}$ . Bearing in mind that  $P_i$  is not a member of any other group, the message delivery conditions for  $P_i$  in  $g_X$  are stated below:

**safe1:** a received  $m$ ,  $m.g = g_X$ , is deliverable if  $m.c \leq D_{X,i}$ ;

**safe2:** deliverable messages are delivered in the non-decreasing order of their numbers; a fixed pre-determined delivery order is imposed on deliverable messages of equal 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  $V_{X,i}$  remain *lively* by sending messages so that  $D_{X,i}$  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 it is not generating computational messages. We assume that this mechanism for a given  $P_i$  prompts  $P_i$  to send a null message, if no (null or non-null) message was sent by  $P_i$  in the past interval of a fixed length, say,  $\omega$ . Null messages contain only protocol related information (such as number, destination group identifiers etc.). When a null message is sent or received by  $P_i$ ,  $LC_i$  is advanced as per CA1 and CA2; however, when it is due for delivery, it is not supplied for processing.

The time-silence mechanism can increase the message overhead of the protocol. However, such a mechanism or some equivalent one (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 (e.g., see [15, 16]). Also, more importantly, it is essential for the detection of process crashes without which a (synchronous or asynchronous) membership service cannot be built (irrespective of the total order protocol being symmetric or asymmetric).

We will now remove the single group assumption and permit  $P_i$  to be a member of more than one group. Let  $G_i$  be the set of groups  $P_i$  belongs to:  $G_i = \{g_X \mid P_i \in g_X\}$ ,  $|G_i| > 1$ . Each process in the system maintains only one LC, irrespective of the number of groups it belongs to; further, this LC is advanced as per CA1 and CA2 irrespective of the group in which that process sends or

receives (null or non-null) messages. Therefore the properties pr1 and pr2 will be true for all messages in the system. A process  $P_i$  maintains a distinct receive vector  $RV_{x,i}$  for each group  $g_x$  in  $G_i$ , representing  $m.c$  of the last  $m$  received from every  $P_j \in V_{x,i}$ . Let  $D_i$  be the minimum of all  $D_{x,i}$  computed for every  $g_x$  in  $G_i$ :  $D_i = \min\{D_{x,i} \mid \forall g_x \in G_i\}$ . Then, it is only necessary to modify the delivery condition **safe1** to:

**safe1'**: a received  $m$  is deliverable if  $m.c \leq D_i$ .

The time-silence mechanism of  $P_i$  will operate independently for each  $g_x$  in  $G_i$ , prompting  $P_i$  to send a null message in a given group  $g_x$ , if no (null or non-null) message was sent by  $P_i$  in that group  $g_x$  for the past  $w$  time units. This ensures that  $D_{x,i}$  of different  $g_x$  in  $G_i$  advance independent of each other and that the value of  $D_i$  increases with time, ensuring that any received  $m$  will eventually become deliverable.

Conditions **safe1'** and **safe2** ensure that a received  $m$  becomes deliverable for  $P_i$  only after a  $m'$ ,  $m'.c \geq m.c$ , is received from every  $P_j \in V_{x,i}$  and for all  $g_x$  in  $G_i$ . These conditions, together with the time-silence mechanism can therefore cope with arbitrarily complex group structures.

## 4.2 Asymmetric total order version

The asymmetric version of Newtop uses one of the members of a group as a sequencer for ordering messages. Though the main idea behind the protocol for single group members has been known for a long time, Newtop extends this idea to overlapping groups with great ease and simplicity. Specifically, unlike [9], it does not require that a common sequencer be chosen for overlapping groups nor that the sequencers of different overlapping groups coordinate their sequencing activities. Further, as we show in the next sub-section, Newtop permits a multi-group member process to execute asymmetric version in one group and symmetric version in another group.

We first consider the case of a process  $P_i$  belonging only to one group  $g_x$  (recall that in a failure-free and static-membership environment, each  $P_i \in g_x$  has an identical view  $V_{x,i} = V_{x,i}^0 = g_x$ ). To multicast a message  $m$  in  $g_x$ ,  $P_i$  unicasts it to a member process, called the *sequencer*, which  $P_i$  selects out of the processes in its (current) membership view of  $g_x$  using a deterministic algorithm (so processes that have the same view are guaranteed to choose the same sequencer). The sequencer multicasts the unicast messages it receives to all processes in its view in the received order and  $P_i$  delivers messages (including its own) in the order they are received from the sequencer process. (A process that also happens to be the sequencer will logically follow the same procedure, unicasting to itself, and then multicasting.) Each process maintains the logical clock according to the rules CA1 and CA2; sending and receiving of unicasts update the logical

clock exactly in the same manner as multicasts do. This ensures that the messages that were consecutively unicast by a given process will be multicast by the sequencer with increasing message numbers. So, when  $P_i$  receives a multicast message  $m$ , it will no longer receive a message with number smaller than  $m.c$  and hence  $P_i$  can be delivered  $m$  straightaway.

Before extending the above scheme to the case where  $P_i$  can belong to multiple groups, we will observe that when  $P_i$  is not the sequencer, it disseminates its message  $m$  to the group members (not by a direct multicast as in symmetric version, but) indirectly through another process. When the sequencer multicasts  $m$ , it assigns a new  $m.c$  which will be different from, and larger than, the number  $P_i$  assigned to  $m$  in its unicast.  $P_i$  cannot know the new  $m.c$  of its own  $m$  until it receives  $m$  from the sequencer. Therefore  $P_i$  observes the following blocking rule when it is a member of multiple groups:

**Send Blocking Rule:** A multi-group member process  $P_i$  must delay unicasting of a message  $m$  (to the sequencer), until it has received (from the relevant sequencers) all the previous  $m'$ ,  $m'.g \neq m.g$ , which it has unicast.

The above rule ensures that the number given to  $m$  by  $P_i$  (and therefore by the sequencer of  $m.g$ ) will be larger than the number given to  $m'$  by the sequencer of  $m'.g$ . That is, consecutive messages disseminated by  $P_i$  in different groups are guaranteed to be multicast by respective sequencers with increasing numbers.

Let  $G_i = \{g_x \mid P_i \in g_x\}$  and  $|G_i| \geq 1$ .  $P_i$  does not maintain a receive vector as it can compute  $D_{x,i}$  simply as the number of the last received message from the sequencer of  $g_x$ . It computes  $D_i$  exactly as in the symmetric version, ie.,  $D_i = \min\{D_{x,i} \mid \forall g_x \in G_i\}$ , and uses conditions **safe1'** and **safe2** for delivery.

It is necessary for only the sequencer of a group to operate the time-silence mechanism for that group. This will ensure that the value of  $D_i$  increases with time and the protocol is live.

## 4.3. Generic total order version

We now present the generic version of Newtop for a process  $P_i$  that can execute the symmetric protocol version in one group (say  $g_y$ ) and the asymmetric protocol version in another (say  $g_z$ ). Let  $G_i = \{g_x \mid P_i \in g_x\}$  and  $|G_i| \geq 1$ . Such mixed-mode working is made possible because both the protocols use the same message numbering scheme. The asymmetric blocking rule needs to be modified as follows:

**Mixed-mode Blocking Rule:** A multi-group member process  $P_i$  must delay unicasting or multicasting of a message  $m$ , until it has received (from the relevant

sequencers) all the previous  $m', m'.g \neq m.g$ , which it has unicast.

In addition,  $P_i$  will operate the time-silence mechanism and compute  $D_{x,i}$  for each  $g_x \in G_i$  as discussed in sub-section 4.1 or 4.2, depending on whether symmetric or asymmetric version is being run in  $g_x$ . It computes  $D_i$  as  $D_i = \min\{D_{x,i} \mid \forall g_x \in G_i\}$  and uses conditions **safe1'** and **safe2** for delivery.

## 5. Fault-tolerant, Dynamic Newtop

We now describe how to extend Newtop to make it dynamic and fault-tolerant: ordering and liveness is preserved even if membership changes occur due to (suspected) process failures, voluntary process departures and new group formations. This requires every process to operate the timesilence mechanism independently in every group in which the process is a member. This is necessary even if simple atomic delivery of messages is sufficient, since failures cannot be detected otherwise. As stated earlier, Newtop provides each  $P_i$  with a *group-view* process, denoted as  $GV_{x,i}$ , for each  $g_x, g_x \in G_i$ .  $GV_{x,i}$  is responsible for maintaining  $P_i$ 's view of the group membership of  $g_x$ . Informally, this extension has the following aspects: (i)  $GV_{x,i}$  uses timeouts to *suspect* a failure of some remote process ( $P_j$ ) that does not seem to be responding; (ii) in which case  $GV_{x,i}$  can initiate a *membership agreement* on  $P_j$ , the outcome of which is that either processes agree to eliminate  $P_j$  from the group view, with an agreement on the last message sent by  $P_j$ , or  $P_j$  continues to be a member and  $P_i$  is able to retrieve missing messages of  $P_j$ .

In the rest of the section, in order to save space, we will consider only the symmetric version of Newtop. The modifications necessary to make the mechanisms presented here applicable to the generic version and just for atomic delivery are discussed in the full version of this paper [7].

### 5.1 Message Stability

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

*Message Stability*: A message  $m$  becomes stable in  $P_i$  if  $P_i$  knows that all processes in the current view of  $m.g$  have received  $m$ .

In Newtop (as in other published protocols), message stability information is piggybacked on the transmitted messages. That is, when a message  $m, m.g = g_x$ , is transmitted by  $P_i$ , a field  $m.ldn$  (ldn: largest deliverable message number) will have the current value of  $D_{x,i}$ . To

identify stable messages,  $P_i$  maintains a vector called  $SV_{x,i}$  (Stability Vector) for each  $g_x$ . At process  $P_i$ ,  $SV_{x,i}[j]$  represents the latest  $m.ldn$  value received from  $P_j$ . If  $\min(SV_{x,i})$  represents the minimum value in  $SV_{x,i}$ , then all  $m, m.c \leq \min(SV_{x,i})$  will be stable. A process can safely discard stable messages after delivery.

## 5.2 Managing Group Membership

Group-view process  $GV_{x,i}$  of  $P_i$  works as if  $P_i$  is not a member of any other group. So, we can ignore the fact that  $P_i$  can be a member of more than one group, and will describe the  $GV_{x,i}$  of  $P_i$  for a given  $g_x$ , dropping for convenience suffix  $x$  when no confusion is likely.

$GV_i$  uses a communication primitive called *mcast(m)* to transmit its message  $m$  to all GV processes of  $P_j \in V_{x,i}$  and the messages are delivered (by the transport layer) to (functioning and connected) destination GV processes in the sent order.  $GV_i$  has a *failure suspector* module,  $S_i$ , which monitors the liveness of every  $P_j, j \neq i$  and  $P_j \in V_{x,i}$ . If  $S_i$  observes that no multicast message has been received from  $P_j$  for a period  $\Omega > \omega$  ( $\omega$  = the time-silence timeout duration) then it suspects the crash of  $P_j$  and notifies  $GV_i$  of its suspicion. In practice,  $\Omega$  should be tuned to a value that minimises the possibility of unfounded suspicions.

The event driven algorithm for  $GV_i$  is given below, dropping the suffix  $i$  for all the set variables used exclusively by  $GV_i$ ; these set variables are initialised to empty and a Boolean variable *consensus* is initialised to *false*, when the group  $g_x$  is formed. The algorithm describes the steps taken by  $GV_i$ , once a certain condition holds. The algorithm for  $GV_i$  has two components, *membership agreement* (for reaching agreement on processes suspected to have failed) and *view installation*. The *membership agreement* component is based on the approach used in Psync [14, 15], adapted to the context of logical clocks.

#### Membership Agreement:

- (i) *notification*  $\{P_k, ln\}$  received from  $S_i$ : suspicions := suspicions  $\cup \{P_k, ln\}$ ; mcast(i, suspect,  $\{P_k, ln\}$ );
- (ii) ( $j, suspect, \{P_k, ln\}$ ) received: **if**  $P_k \neq P_i$  **then** record the suspicion  $\{P_k, ln\}$  of  $GV_j$  in *gossip*; **if**  $P_k = P_i$  **then** discard the received message;
- (iii) *suspicion*  $\{P_k, ln\}$  of  $GV_j$  is recorded in *gossip*  $\wedge (m, m.c > ln, is\ received\ from\ P_k)$ : mcast(i, refute,  $\{P_k, ln\}$ ); /\*  $P_i$  has received a message from  $P_k$  numbered  $> ln$ , so refute  $GV_j$ 's suspicion of  $P_k$ ; all received  $m$  of  $P_k, m.c > ln$ , can be piggybacked on the refute message \*/

(iv)  $(j, \text{refute}, \{P_k, \text{ln}\}) \text{ received} \wedge \{P_k, \text{ln}\} \in \text{suspicious}$ :  
 $\text{suspicious} := \text{suspicious} - \{P_k, \text{ln}\}$ ; recover the missing  
 $m, m.c > \text{ln}$  of  $P_k$ ;  $\text{mcast}(i, \text{refute}, \{P_k, \text{ln}\})$ ;  
(v) for every  $\{P_k, \text{ln}\} \in \text{suspicious}$ , suspect messages  
received from every  $GV_j$  of  $P_j \in V - \{\{P_k \mid \{P_k, \text{ln}\} \in \text{suspicious}\} \cup \text{failed}\}$ :  
 $\text{detection} := \text{suspicious}$ ;  $\text{suspicious} := \{\}$ ;  $\text{mcast}(i, \text{confirmed}, \text{detection})$ ;  $\text{consensus} := \text{true}$ ;  
(vi)  $(j, \text{confirmed}, \text{detection}_j) \text{ received} \wedge \text{detection}_j \subseteq \text{suspicious}$ :  
 $\text{detection} := \text{detection}_j$ ;  $\text{suspicious} := \text{suspicious} - \text{detection}_j$ ;  
 $\text{mcast}(i, \text{confirmed}, \text{detection})$ ;  $\text{consensus} := \text{true}$ ;  
(vii)  $(j, \text{confirmed}, \text{detection}_j) \text{ received} \wedge (P_j, \text{ln}) \in \text{detection}_j$   
for some  $\text{ln}$ : force  $S$  to suspect  $P_j$ ;  $! * P_j$  has succeeded in suspecting  $P_i$ ,  
so reciprocate by suspecting  $P_j$  \*/

A notification from  $S_i$  to  $GV_i$  will be of the form  $\{P_k, \text{ln}\}$  - indicating that  $P_k$  is suspected to have crashed and  $\text{ln}$  is the number of the last message  $P_i$  has received from  $P_k$ .  $GV_i$  maintains a set  $\text{suspicious}_i$  where notifications from  $S_i$  are entered.  $GV_i$  also multicasts a *suspect* message  $(i, \text{suspect}, \{P_k, \text{ln}\})$  to  $GV$  processes of all processes (including  $GV_k$ ) that are in its current membership view  $V_i$ . If  $GV_i$  receives confirmation that all other unsuspected members in  $V_i$  also suspect each  $\{P_k, \text{ln}\}$  in its  $\text{suspicious}_i$ , it decides to treat each  $P_k$  of  $\text{suspicious}_i$  as having failed and  $P_k$  is added to a set called *failed*.  $P_i$  discards any messages received from  $P_k$  and  $GV_k$ , if either  $P_k \in \text{failed}_i$  or  $P_k \notin V_i$ . Also, once suspicion  $\{P_k, \text{ln}\}$  has been added to  $\text{suspicious}_i$ ,  $GV_i$  will keep the messages received from  $P_k$  and  $GV_k$  as pending. If suspicion  $\{P_k, \text{ln}\}$  is subsequently refuted, the pending messages will be assumed to have been just received, and will be handled appropriately; if, however, suspicion  $\{P_k, \text{ln}\}$  is confirmed as a failure, then the pending messages of  $P_k$  and  $GV_k$  are discarded.

Suppose that  $GV_j$  receives the message  $(i, \text{suspect}, \{P_k, \text{ln}\})$  from  $GV_i$ . If  $\{P_k, \text{ln}\}$  is already in  $\text{suspicious}_j$ ,  $GV_j$  regards  $GV_i$  as yet another process that holds the same suspicion as itself; if however  $\{P_k, \text{ln}\}$  is not in  $\text{suspicious}_j$ , it records this suspicion from  $P_i$  in *gossip*, but suspends judgement on it pending confirmation from its own  $S_j$ . If in the mean time  $P_j$  receives a message  $m$  from  $P_k$  with  $m.c > \text{ln}$ , then  $GV_j$  removes  $\{P_k, \text{ln}\}$  from *gossip* and multicasts a *refute* message  $(j, \text{refute}, \{P_k, \text{ln}\})$ . When  $GV_i$  receives this *refute* message, it stops suspecting  $P_k$  for  $\text{ln}$ , and removes  $\{P_k, \text{ln}\}$  from  $\text{suspicious}_i$ ; it also initiates an attempt to recover the missing messages of  $P_k$  (a missing  $m$  can be piggybacked in the *refute* message; by definition any missing  $m$  is

unstable, so would not have been discarded by  $P_j$ ;  $P_j$  can therefore always piggyback  $m$ ). After recovery of the missing message,  $P_i$  multicasts  $(i, \text{refute}, \{P_k, \text{ln}\})$  message. If  $GV_i$  ever receives a message  $(k, \text{suspect}, \{P_j, \text{ln}\})$ , it takes no action in the hope that some  $GV_j$  will refute that suspicion. When  $GV_i$  confirms all of its suspicions (condition (v)) or a subset of them (condition (vi)) into agreed failure detection, it sets the Boolean *consensus* to true. Functioning members that hold identical views and do not suspect each other, will confirm identical *detection* sets in an identical order. (A proof of this can be seen in [14].) Every agreement on a new *detection* set leads to the installation of new view that excludes the processes in the *detection* set.

#### View Installation:

(viii)  $(\text{consensus} = \text{true})$ :  $\text{failed} := \{P_k \mid P_k \in \{P_k, \text{ln}\} \in \text{detection}\}$ ;  
 $\text{ln}_{mn} := \min\{\text{ln} \mid \{P_k, \text{ln}\} \in \text{detection}\}$ ; **for** every  $P_k \in \text{failed}$  **do** instruct  $P_i$  to discard any  $m$  received from  $P_k$  with  $m.c > \text{ln}_{mn}$  **od**;  
 $\text{update\_view}(\text{failed}, \text{ln}_{mn})$ ; **for** every  $P_k \in \text{failed}$  **do**  $\text{RV}[k] := \infty$ ;  $\text{SV}[k] := \infty$ ; **od**;  
 $\text{failed} := \{\}$ ;  $\text{consensus} := \text{false}$ ;

The view installation component assumes the use of a primitive *update-view*( $F, N$ ) which, upon being invoked, will be executed asynchronously and will install a new view before any  $m, m.c \geq N+1$ , is delivered to  $P_i$ . The algorithm is as follows:

$\text{update\_view}(F: \text{set\_of\_processes}; N: \text{integer})$ :  
 $\{\text{wait until } P_i \text{ is delivered the last } m, m.c \leq N; V := V - F; \}$

Absent or rejected messages from suspected processes in the detection set prevents  $D$  from increasing beyond  $\text{ln}_{mn}$  and any received  $m, m > \text{ln}_{mn}$ , of any group will be blocked from delivery. Setting  $\text{RV}[k] := \text{SV}[k] := \infty$  will allow  $D$  to increase more than  $\text{ln}_{mn}$  and message delivery to resume if the value of  $D$  has been stuck at  $\text{ln}_{mn}$ . Before setting  $\text{RV}[k]$  and  $\text{SV}[k]$  to infinity, a message  $m$  of a failed  $P_k$  with  $m.c > \text{ln}_{mn}$  is discarded, even though it has been agreed that  $m$  was sent before  $P_k$  failed. This is a safety measure, necessary to preserve MD5 (see example 1).

After treating all the processes in a given set *detection* as having failed "together" and ignoring their messages with  $m.c > \text{ln}_{mn}$ ,  $GV_i$  calls the primitive *update\\_view* ( $\text{failed}_i, \text{ln}_{mn}$ ) to install the new view,  $V - \text{failed}_i$ , just before any  $m, m.c \geq \text{ln}_{mn} + 1$ , is to be delivered to  $P_i$ .  $\text{RV}$  and  $\text{SV}$  are also updated to reflect the new view. As new view is installed only after the last  $m, m.c < \text{ln}_{mn} + 1$ , is delivered,  $m.s$  will be in the current view for any  $m$  delivered in group  $g_x$  (MD1 is met).

**Example 1:** Suppose that functioning  $P_i$  and  $P_j$  hold identical views and never permanently suspect each other.



Let  $P_r$  crash during the multicast of  $m$ , such that only  $P_s$  receives  $m$ . Let  $P_s$  deliver  $m$  (possible, if the arrival of  $m$  from  $P_r$  causes  $m$  to become deliverable), multicast  $m'$  that is received by  $P_i$  and  $P_j$ , and crash before it could refute the suspicion  $\{P_r, ln\}$  for some  $ln < m.c$ , held by  $GV_i$  and  $GV_j$ .  $P_r$  and  $P_s$  will be detected together by  $GV_i$  and  $GV_j$ , and  $m'$ ,  $m \rightarrow m'$ , is guaranteed not to be delivered when  $m$  cannot be delivered.  $P_i$  and  $P_j$  confirm identical detection sets in identical order; they will execute `update_view()` for identical parameter values in identical order. This ensures that MD3, MD4 and MD4' are met.

**Example 2:** To see that MD5' is met, consider the scenario depicted in fig. 1. Let  $P_i$  and  $P_j$  get (permanently) partitioned from  $P_k$  while  $m_1$  was being multicast, and let them not receive  $m_1$  at all. Since  $m_1 \rightarrow m_4$ ,  $m_4.c > m_1.c$  and  $P_i$  cannot be delivered  $m_4$  until  $D_{1,i}$  increases beyond  $m_1.c$  which will not happen until  $P_k$  is detected to have failed. The prolonged silence of  $P_k$  will cause  $GV_i$  to suspect  $\{P_k, ln_k\}$  for some  $ln_k < m_1.c$ , and then to reach agreement with  $GV_j$  on that suspicion.  $P_k$  will be removed from  $V_{1,i}$  before any  $m$ ,  $m.c \geq ln_k + 1$ , is delivered. Thus, when  $m_4$ ,  $m_1 \rightarrow m_4$ , is being delivered to  $P_i$ ,  $m_1.s$  is guaranteed not be in  $V_{1,i}$ , if  $m_1$  cannot be delivered to  $P_i$  at all. So, MD5' is will not be violated.

The final example is intended to show that the concurrent group views stabilise into non-intersecting ones.

**Example 3:** Consider a group  $g = \{P_i, P_j, P_k, P_l, P_m\}$  in which each functioning member holds the initial view  $V^0 = g$ . Assume  $P_i$  and  $P_j$  never suspect each other and also,  $P_k$  and  $P_l$  never suspect each other. Let  $GV_i, GV_j, GV_k$  and  $GV_l$  suspect  $P_m$  for the same  $ln_m$ , and send to each other the suspect message  $(*, suspect, \{P_m, ln_m\})$ . Let a network failure occur, partitioning  $P_i$  and  $P_j$  from  $P_k$  and  $P_l$  after the suspect messages of  $GV_i$  and  $GV_j$  have been received by  $GV_k$  and  $GV_l$ , but before the suspect messages of  $GV_k$  and  $GV_l$  can be received by  $GV_i$  and  $GV_j$ . After four  $(*, suspect, \{P_m, ln_m\})$  messages are received - one from  $GV$  of each process in  $V^0 - \{P_m\}$  -,  $GV_k$  (also  $GV_l$ ) will install the new view  $V^1 = \{P_i, P_j, P_k, P_l\} = V^0 - \{P_m\}$  after all received  $m$ ,  $m.c \leq ln_m$ , are delivered. (Thus, in view  $V^0$ ,  $P_k$  and  $P_l$  would have delivered an identical set of non-null  $m$  that were multicast in  $g$ .)

$GV_i$  and  $GV_j$ , on the other hand will not receive any suspect messages from  $GV_k$  and  $GV_l$ , so will not succeed in installing the view  $V^0 - \{P_m\}$ ; they will however start suspecting  $P_k$  and  $P_l$ , eventually agree and form  $detection_i = detection_j = \{\{P_m, ln_m\}, \{P_k, ln_k\}, \{P_l, ln_l\}\} = detection_{ij}$  (say) and  $failed_i = failed_j = \{P_m, P_k, P_l\} =$

$failed_{ij}$  (say), and will update their view to  $V^1 = \{P_i, P_j\} = V^0 - failed_{ij}$  after all received  $m$ ,  $m.c \leq \min \{ln_m, ln_k, ln_l\}$ , are delivered.

The existence of intersecting concurrent views is, however, short lived as  $GV_k$  and  $GV_l$  must subsequently suspect  $P_i$  and  $P_j$ : either by receiving  $(i$  (or  $j)$ , *confirmed*,  $\{P_m, P_k, P_l\}$ ) and executing step (vii) if the network partition is transient, or by being notified from the local suspector if the network partition is long lived.  $GV_k$  and  $GV_l$  will therefore eventually update their views to  $V^2 = \{P_k, P_l\}$ . The temporary existence of intersecting concurrent views occurred due to multiple failures - failure of  $P_m$  and network partition - which was perceived to have occurred 'simultaneously' by  $P_i$  and  $P_j$ , and in succession by  $P_k$  and  $P_l$ .

### 5.3. Group Formation

We now describe the main aspects of the group formation protocol of Newtop. We assume that the formation of a new group can be initiated by any process. Selection of such a process and the names of other processes that should belong to the group are dictated by higher level applications; we therefore assume that a process  $P_i$  (initiator) has the names of the intended members of a new group  $g_n$ .  $P_i$  must not be a member of any  $g_x$  such that  $V_{x,i} = g_n$ . The protocol given below has the following characteristics. A two phase protocol is used (with  $P_i$  as the coordinator) to form the group (steps 1-3). If this succeeds, then a member process uses time-silence and group view process to monitor liveness of other processes (step 4); the first message  $P_k$  sends in the new group  $g_n$  is a special message *start-group* that is multicast for reaching agreement (in step 5) on the minimum value for message number ( $m.c$ ) with which application-related computational messages are to be multicast in  $g_n$ .

*Step1:*  $P_i$  sends 'form group  $g_n$ ' message to each intended member of  $g_n$ , inviting them to form a group; the message contains the process-ids of the intended members of  $g_n$ .

*Step2:* When a  $P_j$ ,  $j \neq i$ , receives an invitation to form  $g_n$ , it diffuses this message to each intended member of  $g_n$ , piggybacking its 'yes' or 'no' decision.

*Step3:* A 'no' message acts as a 'veto';  $P_i$  sends its 'yes' message if it receives a 'yes' from the rest within some time duration, else it sends a 'no'.

*Step4:* If a  $P_k \in g_n$  receives an 'yes' message from every proposed member of  $g_n$ , it activates the time-silence mechanism and a process  $GV_{n,k}$  for the newly-formed  $g_n$ ; the initial view  $V^0_{n,k}$  is set to  $g_n$  and  $RV_{n,k}$  is initialised to 0. The first message  $P_k$  sends in the new group is a special message *start-group* which contains an integer

field called the *start-number* that is set to the *m.c* of the message. This number indicates  $P_k$ 's proposed minimum value for message number with which application-related computational messages are to be multicast in  $g_n$ .

*Step5:*  $P_k$  waits for the following condition to be satisfied before it can send any application related, computational message in  $g_n$ : receive a *start-group* message from every  $P_j$  in its current view  $V_{n,k}$ . (Note that the current view need not be  $V_{n,k}^0$  due to view updates by  $GV_{n,k}$  which is executing in parallel; also,  $P_k$  is not blocked from sending null messages in  $g_n$  when prompted by the time-silence). While  $P_k$  is waiting for the condition to become true,  $D_{n,k}$  is not allowed to be modified except when  $P_k$  receives a *start-group* message with *start-number* larger than  $D_{n,k}$ , in which case  $D_{n,k}$  is increased to the proposed *start-number* of the incoming message. Once all the required *start-group* messages are received,  $P_k$  sets  $D_{n,k}$  to *start-number-max* = the maximum of *start-numbers* proposed by all  $P_j$  in view  $V_{n,k}$ ;  $LC_k$  is set to *start-number-max* if *start-number-max* is larger;  $P_k$  then starts sending and delivering application-related computational messages of  $g_n$ .

To see the correctness of the group formation protocol, suppose that  $P_k$  is already a member of one or more groups when it is attempting to form a new  $g_n$ . While it is waiting for the condition of step 5 to become true, the value of  $D_{n,k}$  is incremented cautiously so that  $P_k$  is not delivered any  $m$ ,  $m.c > \text{start-number-max}$ , until that condition becomes true. Any computational message that was multicast in  $g_n$  will have  $m.c > \text{start-number-max}$ . This ensures that  $P_k$  can be delivered the messages multicast in  $g_n$  together with those multicast in other groups, in a non-decreasing order of message numbers.

## 6. Comparison with Related Work

Psync/Consul [15, 17] is one of the best known protocol suite that implements causal and symmetric total delivery protocols; however, it has no support for multiple (overlapping) process groups. The Trans and Transis family of protocols [1, 6, 12] use elegant symmetric solutions for providing total order delivery, but are not quite general purpose, as they rely on network level broadcast communication; further, like Psync/Consul, the issue of a process belonging to multiple process groups has not been addressed. ISIS was the first system to include support for multiple groups; however the vector clock based protocols of ISIS [4] become quite difficult and expensive to implement for arbitrary group structures (e.g., cyclic groups, such as shown in fig. 2)). All previously published symmetric total order protocols require multicast messages to contain explicit information about causally preceding messages, and represent the received messages in a directed acyclic graph. The task of maintaining such a graph is much more complicated -

especially for multiple groups - than the simple approach of using receive vectors adopted in Newtop. Newtop is able to offer this advantage because it does not attempt to precisely represent the *absence* of causal relation among multicasts as this is not essential for total order message delivery. The net effect is that Newtop has low and bounded message space overhead (the protocol related information contained in a multicast message is small) and is relatively easy to implement even when process groups overlap in an arbitrary manner. Further, Newtop has the capability, not available on any existing protocols, of supporting both symmetric and asymmetric protocols.

As explained with respect to fig. 3, virtual synchrony provided by Newtop is more flexible than that of ISIS. In this respect, Newtop has the same functionality as other modern group communication systems, such as Transis and Relacs [3].

The membership algorithm of Newtop is based on the approach used in Psync/Consul, adapted to the context of logical clocks and extended to coordinate view updates with message delivery. Our protocol maintains view consistency in the presence of (real or virtual) partitions by permitting a group of processes to partition themselves into two or more sub-groups of connected processes with the property that: (i) the functioning processes within any given subgroup will have identical views about the membership; and (ii) the views of processes belonging to different subgroups are guaranteed to stabilise into non-intersecting ones. This makes Newtop more powerful than many other protocols [14, 18] that can guarantee continued group operation only when the group partitions in such a way that exactly one subgroup can be uniquely identified as the primary. The membership service of Newtop is essentially similar in functionality to those of Transis [2], the protocols of [12, 19] and Relacs [3]. Below we briefly examine [19].

In the protocol of [19], concurrent views are always non-intersecting (considering example 3 of the previous section, the situation where  $V^1$  of  $P_k$  and  $P_l$  intersect with  $V^1$  of  $P_i$  and  $P_j$  will not occur). Never-intersecting concurrent views are guaranteed in [19] essentially by defining a process view as a set of process *signatures*, where a signature is a tuple: {process-id, sequence-number}. It is possible to adapt this approach in Newtop. Let  $GV_i$  replace  $V_i$  by  $\vartheta_i = \{\{P_j, e_i\} \mid \forall P_j \in V_i\}$ , where  $e_i$  is the total number of processes  $GV_i$  has excluded from the initial view;  $e_i=e_j$ , if  $V_i^r=V_j^r$  for every  $r \geq 0$ . Thus, in the example,  $\vartheta^0 = \{\{P_i, 0\}, \{P_j, 0\}, \{P_k, 0\}, \{P_l, 0\}, \{P_m, 0\}\}$  for all functioning processes of  $g$ . After partitioning,  $\vartheta_i^1 = \vartheta_j^1 = \{\{P_i, 3\}, \{P_j, 3\}\}$  which do not intersect with  $\vartheta_k^1 = \vartheta_l^1 = \{\{P_i, 1\}, \{P_j, 1\}, \{P_k, 1\}, \{P_l, 1\}\}$ ; after stabilising,  $\vartheta_k^2 = \vartheta_l^2 = \{\{P_k, 3\}, \{P_l, 3\}\}$ .

Finally, unlike other protocols, Newtop supports dynamic formation of new groups. The formation protocol exploits the fact that processes are permitted to belong to

several groups. The group formation facility is more powerful than 'joining an existing group' facility of current protocols, as the effect of joining a group can be obtained by processes forming a new group and exiting the previous ones.

## 7. Concluding Remarks

Newtop is a general purpose protocol suite that is suitable in a variety of settings: processes can simultaneously belong to many groups, group size could be large, and processes could be geographically widely separated, communicating over the Internet. It supports both symmetric and asymmetric ordering protocols, allowing a multi-group process to use both. Newtop offers this flexibility for a small price: new multicast in a given group is blocked only if any multicast made in a different asymmetric group is awaiting distribution by the sequencer. If only symmetric version is used, Newtop is totally non-blocking on send operations. Newtop is however not complex, as it offers a very simple method for dealing with multiple process groups and has low message space overhead. The membership service of Newtop supports concurrent existence of multiple subgroups, leaving it to applications to decide on the eventual fate of such subgroups. In addition, Newtop supports dynamic formation of new groups. We have also designed and implemented a flow control mechanism that ensures that a sender process does not cause buffers to overflow at any of the functioning destination processes. The interested reader is referred to [11] for details.

**Acknowledgements:** This work has been supported in part by grants from the UK MOD and the Engineering and Physical Sciences Research Council (Grant no. GR/H1078), ESPRIT basic research project 6360 (BROADCAST) and CNPq/Brazil. Comments from and Discussions with Michel Raynal, Ozalp Babaoglu, Dalia Malki and Sam Toueg clarified our understanding. Colin Low hinted at the possibility of using overlapping groups for solving the problem of online server migration.

## References

[1] Amir, Y., et al, "Transis: A Communication Sub-system for High Availability", Digest of Papers, FTCS-22, Boston, July 1992, pp. 76-84.  
[2] 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.  
[3] Babaoglu, O., Baker, M., Davoli, R., and Gianchini, L., "Relacs: a Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems", BROADCAST Project deliverable report, October 1994 (available from Dept. of Computing Science, University of Newcastle upon Tyne, UK).

[4] 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.  
[5] Chandra, C.T. and S.Toueg, "Unreliable Failure Detectors for Asynchronous Systems" Proc. of 10th ACM Symp. on Principles of Dist. Comp., Montreal, , August 1991, pp. 257-272.  
[6] Dolev, D., Kramer, S. and Malki, D., "Early Delivery Totally Ordered Multicast in Asynchronous Environment", Digest of Papers, FTCS-23, Toulouse, pp. 544-553, June 1993.  
[7] Ezhilchelvan, P.E., Macedo, R. A., and Shrivastava, S. K., "Newtop: A Fault-tolerant Group Communication Protocol", BROADCAST Project deliverable report, October 1994 (available from Dept. of Computing Science, University of Newcastle upon Tyne, UK).  
[8] Fischer, M., Lynch N., and Paterson, M., "Impossibility of Distributed Consensus with One Faulty Process", J. ACM, 32, April 1985, pp 374-382.  
[9] Garcia-Molina, H., and Spauster, A., "Ordered and Reliable Multicast Communication", ACM Transactions On Computer Systems, Vol. 9, No. 3, August 1991, pp. 242-271.  
[10] Lamport, L., "Time, clocks, and ordering of events in a distributed system", Commun. of ACM, 21, 7, July 1978, pp. 558-565.  
[11] Macedo, R. A., "Fault-tolerant Group Communication Protocols for Asynchronous Systems", Ph. D. thesis, 1994, University of Newcastle upon Tyne.  
[12] 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.  
[13] Melliar-Smith, P.M., Moser L.E., and Agarwala, V., "Membership Algorithms for Asynchronous Distributed Systems", Proc. of 12th Intl. Conf. on Distributed Comp. Systems, pp. 480-488, May 1991.  
[14] Mishra, S., Peterson L., and Schlichting, R., "A membership Protocol Based on Partial Order", Proc. IFIP Conf. on Dependable Computing For Critical Applications, Tuscon, Feb. 1991, pp 137-145.  
[15] Mishra, S., Peterson L., and Schlichting, R., "Consul: a Communications Substrate for Fault-Tolerant Distributed Programs", Distributed Systems Engineering, 1 (1993), pp. 87-103.  
[16] Mostefaoui, A., Raynal, M., "Causal Multicasts in Overlapping Groups: Towards a Low Cost Approach", Research Report, IRISA Campus de Beaulieu -35042 RENNES, France.  
[17] 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.  
[18] Ricciardi, A.M., and Birman, K., "Using Process Groups to Implement Failure Detection in Asynchronous Environments", Proc. of Annual ACM symposium on PoDC, pp. 341-352, August 1991.  
[19] Schiper, A., and Ricciardi, A.M., "Virtually Synchronous Communication based on a Weak Failure Susceptor", Digest of Papers, FTCS-23, Toulouse, pp. 534-543, June 1993.