Network Membership: A Partition Model for Reliable Mobile Communication

Romain Boichat
Communication Systems Department
Swiss Federal Institute of Technology
CH-1015 Lausanne EPFL, Switzerland

Laurence Duchien
CNAM - Laboratoire CEDRIC
292 Rue St-Martin
F-75141 Paris Cedex 03, France

Abstract

We give a formal specification of a new model called Network Membership for reliable mobile communication in asynchronous distributed systems. Our approach is new in the sense that the Network Membership service does not have any join or leave procedures. We let the system flow, views are not forced and are installed with stability. The model is less restrictive than others since no consensus is required. The Network Membership allows multiple partitions to operate simultaneously and provides connectivity feedback. We have build on top of this Network Membership service an efficient reliable broadcast service that is resistant to network partitions. The protocol insures that all recipients eventually receive the message even if a receiver has been partitioned away at any time. We show how we use an unreliable channel detector in conjunction with data forwarding and stability to achieve this goal.

1. Introduction

With the emergence of the Internet, applications are not restricted anymore to local area networks but to a wider scale. Network applications for mobile computing, collaborative work, replicate database, or data sharing involve cooperation between multiple processes disseminated in large networks. Reliable broadcast mechanisms have been a topic of intense research and development over the last years. Moreover, a study [20] shows that 30 percent of Internet traffic is multicast and foresees a growth up to 50 percent in the next few years. Possible partitioning of the communication network is an extremely important aspect. In the context of this paper, we define *partitioning* as the creation of at least two *partitions*, while *partitions* are a subset of processes that can communicate and that are isolated from all others. It might result in service reduction or degradation but it should not necessarily put applications in unavailable state.

Group communication applications as videoconference or distribution data require reliable broadcast and multicast protocols in wide area networks. They are often designed on IP-multicast [11]. [21] and [13] give a good survey and a taxonomy of these protocols. They manage data propagation, reliability mechanisms as defined in the network domain (i.e., error and flow controls), ordering delivery, group management, but none of them takes network partitionings in account. They leave applications, if necessary, to consider process crashes and messages retransmission when processes recover.

With mobile computing, applications must support disconnected operations and must also face partitions. Communication between *mobile hosts* (MHs) and static network are managed with *mobile support stations* (MSSs) that communicate directly with MHs via usually wireless links. MHs are able to connect to the static segment of the network at different times and locations. So, the network topology changes dynamically. This mobility introduces several problems. First, network protocols and distributed algorithms for mobile environments cannot assume that a host maintains a fixed and universal known location in the network at all times. The network layer must define new addressing schemes and protocols for routing messages to and from mobile hosts [16, 5, 15]. Second, mobile environments also have important implications for distributed data management. For example, mobile distributed filesystems [19, 23, 18] must not loose messages. Special communication layers buffer messages at their origin throughout the duration of the partition and retransmit them upon reconnection. We did not find partition models and group communication for these mobile applications. Each of them designs its own communication layer with logs or queues with point-to-point mechanisms.

Membership services and protocols maintains consistent information at all sites about membership of a group of processes [24, 10, 2, 17, 4, 3, 14]. Note that there are several partition models in distributed group communication. Two models are usually considered: (i) the *primary-partition* model [8], and (ii), the *minortity-partition* model [3]. In the *primary-partition* model, also called *majority-partition*, only processes in the partition that contains a majority of processes are allowed to make progress. With the *minority-partition* or partitionable model, processes in multiple partitions make progress even if they have only one part of the communication messages, and hence, increase the availability of the system. Acharia and Badrinath [1] propose a multicast protocol for mobile hosts. They mainly handle host view changes due to group members mobility (i.e. their location change). They assume that their membership does not change during the group's lifetime and that these ordered changes could be ordered by a central coordinator to disseminate change in MSSs.

Overview. In this paper, we focus on a partition model for mobile computing called *Network Membership* that allows multiple partitions to operate simultaneously, to route messages and provides connectivity feedback. We design a reliable broadcast protocol built on top of this model that aims large scale networks. With this protocol, a distributed message storage system allows to retransmit messages when connectivity comes back. The proposed partition model is less restrictive than *minority-partition* and *partition-aware* model because no consensus is required. Our model does not differentiate MHs and MSSs since a MSS can also crash, thus our model is more general.

In classical fault tolerant systems to overcome partitioning problem, a group membership service registers changes in the group and distributes the common *view* to each process. A view contains a subset of the system members that can be reached in the group. When a process is suspected to have crashed, it is removed upon agreement from the view. Our approach is different in many ways. Our *Network Membership* service is distributed, i.e., no single centralised server handles the global view of the group membership. Since *Network Membership* lets the system flow; it does not force view changes. It achieves its goal through an exchange of views that is self-stabilising in a sense similar, but not identical to the notion of self-stabilising systems defined by Dijkstra [12]. To update local views, a channel failure detector associated to each process records the local view changes.

With our stabilisation property and with partition information sharing, we are able to build an efficient reliable broadcast protocol for partitions on top of *Network Membership*. The protocol ensures that all recipients eventually receive the message even if the sender or a receiver has temporarily crashed. In the *stabilised* phase, the protocol uses partition information to efficiently route messages. During the *stabilisation* phase, the protocol continues to send messages, but it does not guarantee to deliver messages in an efficient manner.

Contributions. The main contributions of this paper are to propose (i) a formal specification of a *Network Membership* in a partition model for mobile systems, and (ii) to show an efficient broadcast protocol based on this specification. Our *Network Membership* can be view as a part of a network layer that manages addressing and routing messages informations for communication groups of mobile hosts. Moreover, the partition model provides a connectivity feedback when processes are merged in a new partition. It is based on point-to-point communication mode. The reliable broadcast protocol is resistant to network partitions in asynchronous distributed systems. This protocol is reliable in the sense that a receiver eventually receives the message even if the receiver was partitioned away at any time. Moreover, to be efficient in large scale connected network, it gains advantage from information provided by the process view to route messages i.e, the broadcast protocol uses directly the local view to minimise messages in the network. This protocol complements our *Network Membership* to provide a reliable broadcast transport layer for mobile environments.

Roadmap. The paper continues as follow: section 2 introduces the system and communication models. Section 3 presents the overall architecture and its services. Section 4 gives the formal specification of our *Network Membership* along with our unreliable channel detector. Section 5 presents the specification, the implementation and the performance measurements of an efficient reliable broadcast algorithm in the context of partitions that we have built on top of the *Network Membership*. Ultimately, we conclude and forecast some future work in section 6. Due to a lack of space, we do not give proofs or algorithms but they can be found in a tech report [6]

2 Model

We adopt a notation and a terminology similar to that of Chandra and Toueg [9]. We consider asynchronous message-passing distributed systems in which there is no bound on message delay and a finite set Π of *processes*. We make no assumptions on the time it takes for a message to be delivered, neither on the relative speed of processes. The communication network implements channels connecting pairs of processes and the primitives *Send*$_p$*()* and *Receive*$_p$*()* for sending and receiving messages over them. We use a discrete global clock whose range ticks $\mathcal{T}$ is the set of natural numbers.

History. At each clock tick, each process executing a distributed algorithm performs an event chosen from a set S. Set S includes at least the *null* event (denoted as ϵ) and the *Send*$_p$*()* and *Receive*$_p$*()* primitives. The global history of a run of a distributed algorithm is a function σ from $\Pi \times \mathcal{T}$ to S. If a process p executes an event $e \in S$ at time t, then $\sigma(p, t) = e$; otherwise $\sigma(p, t) = \epsilon$ indicates that process p does not perform any event at time t.

Processes (MHs and MSSs). The system consists of a finite ordered set of *n* processes that have unique identifiers,

$\Pi = \{p_1, p_2, ..., p_n\}$. We do not consider Byzantine failures. Processes fail by crashing and may later recover. Formally: a *failure pattern* $F(t)$ is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that does not run at time t. Since processes may crash and recover, we say that process p is *up at time t (in F)* if $p \notin F(t)$, and p is *down at time t (in F)* if $p \in F(t)$. We state that *p crashes* at time t if p is up at time *t-1* and p is down at time t. We induce that *p recovers* at time $t \geq 1$ if p is down at time *t-1* and p is up at time t. We define *Correct(t)* the set of processes that are *up at time t*, and *Faulty(t)* the set of processes that are *down at time t*. Moreover, $\forall t$: *Correct(t)* $\cap$ *Faulty(t)* $= \emptyset$ and $\forall t$: *Correct(t)* $\cup$ *Faulty(t)* $= \Pi$.

2.1 Communication Channels Definitions

A process p sends a message m to a process q with the event $Send_p(m,q)$, and receives a message m through the event $Receive_p(m,q)$. We consider also that all messages sent are globally unique and that a message is received only if it has been previously sent, thus avoiding Byzantine communication failures. A communication channel between processes p and q is bidirectional but not FIFO (i.e., messages can be lost, duplicated, or unordered). We now define certain terms that will be used throughout the paper.

- *Can Directly Communicate With (CDCW(p,q,t))*: a channel between p and q is said to be *open* at time t if the connection between p and q is *open on p and on q* at time t, and communication is possible in both directions. We denote this property $p \leftrightarrow_t q$ or *CDCW*(p,q,t) and remove t when there is no confusion. Intuitively, $p \leftrightarrow_t q \Leftrightarrow p \rightarrow_t q \wedge q \rightarrow_t p$. In any other case, a channel is *closed* at time t ($p \not\leftrightarrow_t q$). Closed links create communication failures which may cause temporary or permanent partitions in the network.

We assume that communication channels satisfy the following properties: (i) *Eventual Symmetry:* If communication is possible from p to q, unless p or q crashes or they are partitioned, communication is eventually possible from q to p; (ii) *Fairness:* If $p \leftrightarrow_t q$, only one $Send_p(m,q)$ from p is required for q to eventually receive m. This property is guaranteed since our channels transparently resend messages as long as these have not been acknowledged by the recipient. Furthermore, we now define:

- $Open_p(t)$: is the set of all *open* channels of p at time t.
- $Closed_p(t)$: denotes all closed channels of p at time t. Consequently, $Open_p(t) \cap Closed_p(t) = \emptyset$.
- *Can Communicate With (CCW(p,q,t))*: holds at time t for p and q if there is a sequence of processes $p = p_0,...,p_l = q$ such that $\forall i \in [0, l-1], p_i \leftrightarrow_t p_{i+1}$. We denote this relation by $p \rightsquigarrow_t q$. This relation indicates whether process q can be *reached* by process p at time t or not. If p cannot reach q, we denote it as $p \not\rightsquigarrow_t q$. We also denote this relation by *CCW*(p,q,t) and remove t when there is no confusion.

Causal Message Chain: is denoted $\mho_{p,q}(t_0,t_1)$, if it is from p to q between t_0 and t_1, and a sequence of both messages $m_0,...,m_l$ and processes $p = p_0,...,p_l = q$ such that: $\exists t_p, t_q \in [t_0,t_1]$: $\sigma(p,t_p) = Send_p(m_0,p_1)$ and $\forall i \in [1, l-2]$ $\exists t_{i_0} < t_{i_1}$: $\sigma(p_i,t_{i_0}) = Receive_{p_i}(m_{i-1},p_{i-1}), \sigma(p_i,t_{i_1}) = Send_{p_i}(m_i,p_{i+1})$ and $\sigma(q,t_q) = Receive_q(m_l,p_l)$.

2.2 Stability and Partition

We now formally define *stability* and *partition*. As described previously, communication channels crash and recover. *Stability* describes a stable state of the communication channels, while *partitions* represent the partitioning of the system composed of the processes. Partitions are defined between processes and might end in non-empty intersections. Since communication links break and recover more often than processes, transitivity is not always true.

Stability and Minimal Stability. The state of the communication channels is *stable* from time t on, if the states of all communication channels between all processes in the system do not change. In other words, all communication channels that are *open* at t_0 stay *open* and all communication channels that are *closed* at t_0 stay closed. Formally,

$$\forall t \geq t_0, \forall p \in \Pi, Open_p(t) = Open_p(t_0) \text{ and } Closed_p(t) = Closed_p(t_0).$$

However, it is very unrealistic that a system remains stable forever. We derive from *stability*, a less restrictive property called *minimal stability* that assures stability for a certain period sufficient for a *causal message chain* to be established between every pair of processes in the system. Formally,

$$\exists t_0, t_1, \forall t \in [t_0,t_1], \forall p \in \Pi, Open_p(t) = Open_p(t_0) \text{ and } Closed_p(t) = Closed_p(t_0) \text{ and } \forall (p,q)\ \exists\ \mho_{p,q}(t_0,t_1).$$

Partition. The relation $\rightsquigarrow$ defines an equivalence relation on the set of correct processes on a stable state of the communication channels. The equivalence classes are called *partitions*. The partition of a process p (the partition in which p is) at time t is denoted *partition(p,t)*. If $p \in F(t)$ then *partition(p,t)* $= \emptyset$. We can now define a partition pattern function P from $\mathcal{T} \times \Pi$ to 2^{Π}, where *P(p,t)* indicates at time t the set of processes that are not in the same partition as p. Formally, $P(p,t) = \{q \mid p \not\rightsquigarrow_t q\}$. Since network changes occur at any time and might imply several partitions changes, *P(p,t)* might not have anything in common with *P(p,t+1)*.

3 Services & Architecture

The architecture is divided in four layers *Communication*, *Network Membership*, *Multicast/Broadcast* and *Application* that are described below.

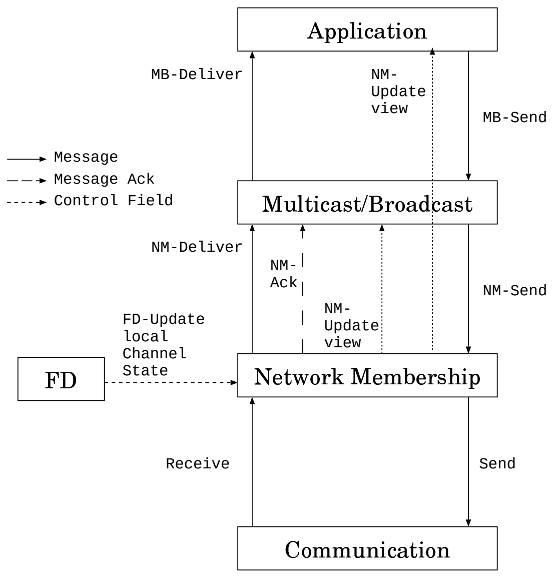


Figure 1. Architecture

Communication. This layer handles point-to-point as well as multi-point communication schemes for the entity. The *Communication layer* is based on the model described in section 2. The layer has no other functionality beside handling sends or receives and creating or destroying communication links.

Network Membership. This layer keeps track of the processes and channels states. The *Network Membership* layer handles all channel state updates either *locally* from the channel failure detector (FD) or *externally* from the *Communication layer*. The failure detector signals a channel update to the *Network Membership layer* with the upcall *FD-UpdatelocalChannelState*. In the same context, this layer indicates to the upper ones (*Multicast/Broadcast* and *Application*) the changes in the *Network Membership* with the primitive *NM-UpdateView*. This upcall is used to inform the above layers of *Network Membership* changes. Moreover, this layer sends and delivers messages using the *Communication layer* with the primitives *Send* and *Receive*. The *Network Membership* is described in more details in section 4.

Multicast/Broadcast. This layer handles multicasts and broadcasts messages with different semantics to a process group since the layer has the knowledge of the *Network Membership*. The various semantics are *reliable or simple* sends and receives. The *Multicast/Broadcast layer* sends (resp. delivers) messages through the primitives *NM-Send* (resp. *NM-Deliver*). The *Network Membership* changes are indicated to this layer by the primitive *NM-UpdateView*. The layer is updated with *NM-Ack* which enables it to know which message has been received from which process. The specification along with the implementation of this reliable protocol is described in section 5.

Application. The *Application* layer is the programmer API. A programmer invokes the *MB-Send* to reliable deliver messages in the system. The type of diffusion (broadcast or multicast) is given as a parameter in the *MB-Send* primitive. The application is notified of the delivery of a message by the upcall *MB-Deliver* and is updated of the view change by the upcall *NM-UpdateView*.

4 Network Membership

First, we introduce formally the notions of *stable view* and *local view*. We then describe our *Channel Failure Detector* and its properties. Finally, based on this knowledge, we define formally our *Network Membership*.

4.1 Stable View and Local View

Our specification for *Network Membership* is different in many ways to classical group membership and we outline the main differences. As with group membership, views are abstractions of the environment with respect to process crashes/recoveries and network partitions/merges. They are shared by all valid processes that belong to the same partition. Our notion of *view* is less restrictive, i.e., there is no explicit agreement on views. We distinguish between two kinds of views: *stable view* and *local view*.

First, a *stable view* corresponds to the view shared by all valid processes as defined in group membership. A *stable view* exists only when the system has undergone *minimal stability*, i.e., the system is in a *stabilised phase*. The system evolves from one *stable view* to another *stable view* but this event cannot be appended to S since the system never knows when a *stable view* is installed.

A *stable view* sv represents a set of processes. A *stable view* sv has a global unique identifier, $\overline{sv}$ denotes the set of members (processes) of *stable view* sv. *Stable views* can be concurrent since the system is partitionable, requiring a common *stable view* for all processes is clearly not feasible when processes are in different partitions and each one has a different perception of the membership. The current *stable view* of process p at time t is formally $sview(p,t) = sv$, $sview(p,t)$ represents the last *stable view* that was reached by p before time t. If sw *succeeds* sv at p, then $sv \prec_p sw$. *Stable view* sw is called *immediate successor of* sv at p or sv is the *immediate predecessor of* sw. We call *s-vchg(sw)* when the system undergoes a *stable view* installation sw. Formally,

$$\exists t_0 \leq t_1 \leq t_2, \forall t \in [t_0, t_2] : Open_p(t) = Open_p(t_0) \text{ and } Closed_p(t) = Closed_p(t_0) \text{ and } \forall (p,q)\ \exists\ \mho_{p,q}(t_0, t_1) \Leftrightarrow \exists\ sv, \forall t' \in [t_1, t_2] : sview(p, t') = sv.$$

Second, the *local view* that we simply call *view* corresponds to the *local process view*. When a process catches some changes in neighbour links and processes, it then updates its *local view*. A *view* v has a global unique identifier, $\overline{\mathrm{v}}$ denotes the set of members (processes) of *view* v. The current *view* of process p at time t is formally $view(p,t) = v$, if v is the last *view* installed at p before time t. By analogy, *view* w is called *immediate successor of* v at p denoted $v \prec_p w$ or v is the *immediate predecessor of* w. However, in contrary to stable view, we append S with an event called *vchg(w)* describing a *view* change that installs *view* w.

4.2 Channel Failure detector

Each process p has access to a local failure detector module which outputs hint about the closed channels of p with other processes. The channel failure detector history CH is a function from $\mathcal{T} \times \Pi$ to 2^{Π} that outputs the closed channels of the process. Formally,

$q \in CH(p,t) \Leftrightarrow p \not\to_t q$; $q \notin CH(p,t) \Leftrightarrow p \to_t q$.

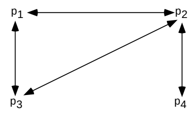
We assume that the channel failure detector is perfect with respect to our (virtual) channels. A channel loss due to a failure in the network is eventually always detected. If the failure affects the existing connection, but the network still offers a correct physical path between the processes, the channel will be re-established. The same action takes place in the case of false suspicion. During such *glitches*, the system is considered being in an *unstable* phase.

The *Channel Failure* detector also initialises the *channelState*$_p$. When a process receives a message from another process for the first time, the process appends it to its *channelState*$_p$. Otherwise, a process p only updates its *channelState*$_p$ as described earlier. If *minimal stability* is ensured, the following lemma holds.

Lemma 1. All processes share the same *network knowledge*.

Channel State. We call *network knowledge* the information that a process p has about the network and it is kept in a matrix called *channelState*$_p$. It reflects the state of the processes that can communicate with process p. The matrix *channelState*$_p$ allows each process to know the state of communication links for the whole system. The matrix $channelState_p$ is divided in n *channelState vectors*, each corresponding to a line of *channelState*$_p$ matrix. With each *channelState vector* is associated a logical time stamp that is initialised to 0 denoted $ts_p(i)$.

Notation: $channelState_p[q,r]$ represents the state of the channel from q to r as assumed by p; *channelState*$_p$*(i)* = *i-th channelState vector* for process p for a given $q = i$ such that $\forall r$ *CDCW(q,r)* as seen by p. Figure 2 shows a typical *channelState*$_p$ matrix in a stable system where all processes share the same *channelState*$_p$; $\times$ means that the link is *closed* or does not exist, and $\bigcirc$ that the link is *open*.



(a) Links status

	p_1	p_2	p_3	p_4
p_1		$\bigcirc$	$\bigcirc$	$\times$
p_2	$\bigcirc$		$\bigcirc$	$\bigcirc$
p_3	$\bigcirc$	$\bigcirc$		$\times$
p_4	$\times$	$\bigcirc$	$\times$	

(b) ChannelState$_p$

Figure 2. *channelState*$_p$ from the link status

Network knowledge propagation. The behaviour of the primitives *NM-Send*$_p$*()* and *NM-Deliver*$_p$*()* allows to understand how each process learns about the other processes status. To simplify, we did not add in the algorithm the data structure that allows each process p to remember for every process the last sent *channelState*$_p$. When sending a message, p checks if *channelState*$_p$ has changed since the last sent message to process q. Process p then piggy-backs, if any, the new data structure (updated *channelState*$_p$ with the associated updated timestamps) on the message. When p receives a message m from q, p compares all received timestamps (ts) and replaces all older *channelState vectors*.

```
this_prss: process id in the system
procedure NM-Send_p($m$,$q$)                                    {$p$ sends $m$ to $q$}
    for all processes $r$ do
        if channelState_p(r) has changed since the last message sent
        to $q$ then
            $ts_p(r) = ts_p(r) + 1$, such that $p$ = this_prss
            append channelState_p(r) + $ts_p(r)$ to $m$
    Send_p(m,q)

procedure NM-Deliver_p($m, q$)                          {upon receive $m$ from $q$}
    for all received channelState_q(i) do
        if $ts_p(i) < ts_q(i)$ then
            channelState_p($i$) = channelState_q($i$)
            $ts_p(i) = ts_q(i)$
    Receive_p(m,q)
```

4.3 Network Membership Specification

The *Network Membership* is defined as a set of properties on *stable view* compositions and *stable view* achievements. We specify *Network Membership* as a set of properties on *stable view* compositions and *stable view* installations, stated in terms of the partitioning pattern that occurs during an execution. In an asynchronous system, a *stable view* cannot reflect the actual partition pattern. To circumvent this barrier, we define some requirements:

(NM1) *Stable View Agreement*. If process p reaches *stable view* sv_1 and its immediate successor sv_2, both containing q, then p reaches sv_2 after q reached sv_1. Formally,

$$\forall p, q, sv_1, sv_2 : sv_1 \prec_p sv_2 \text{ and } p, q \in sv_1 \cap sv_2 \text{ and}$$
$$\forall t \in [t_0, t_1], sview(p,t) = sv_1 \text{ and}$$
$$\forall t' \in [t_2, t_3], sview(q,t') = sv_1 \Rightarrow t_2 < t_1.$$

(NM2) *Stable View Accuracy*. If $p \rightsquigarrow q$ holds forever, then eventually when the system reaches *stable view* sv, p and q

eventually have the same *view*. Formally,

$$\exists t_0, \forall t \geq t_0 : p \rightsquigarrow_t q \Rightarrow \exists t_1, \forall t' \geq t_1 : \textit{sview(p,t')} = \textit{sview(q,t')}.$$

(NM3) *Stable View Completeness*. If all processes q in some partition Ω hold $p \not\rightsquigarrow q$ forever, then eventually when the system reaches *stable view*, p does not have any process $q \in \Omega$ in its *stable view*. Formally,

$$\exists t_0, \forall q \in \Omega, \forall p \notin \Omega, \forall t \geq t_0 : p \not\rightsquigarrow_t q \Rightarrow \exists t_1, \forall t' \geq t_1 : sview(p,t') \cap \Omega = \emptyset.$$

(NM4) *Stable View Integrity*. Every process p that reaches a *stable view* is included in that *stable view*. Formally,

$$\forall p, t : p \in sview(p,t).$$

Stable View Accuracy and *Stable View Completeness* are slightly different because the *reachable* property between processes is not transitive. Our *Network Membership* is defined by the properties NM1, NM2, NM3 and NM4.

5 *MBR-Broadcast*

First, we define a reliable broadcast protocol based on *Network Membership*. We describe the protocols for *MB-Send* and *MB-Deliver* which invoke *NM-Send* and *NM-Deliver*. Finally, we give some performance measurements of our implementation.

5.1 Specification of *MBR-Broadcast*

We redefine the properties of the Chandra-Toueg [9] *R-broadcast* (hereafter denoted *CTR-broadcast*) for partitions. *CTR-broadcast* guarantees that (a) all correct process deliver the same set of messages, (b) all messages broadcast by correct processes are delivered and (c) no spurious messages are ever delivered. Transposed to partition, these properties become when the system is stable:

(a) **Validity**: If a correct process *p MB-Sends* a message *m*, then it eventually *MB-Delivers m*.

(b) **Agreement**: If a correct process *p MB-Delivers* a message *m*, then all processes that belonged before for some time in *partition(p)* eventually *MB-Deliver m*.

(c) **Uniform integrity**: For any message *m*, any process *p* that *MB-Delivers m*, *MB-Delivers* it at most once and only if it was previously *MB-Sends* by sender(*m*).

5.2 General concepts

A reliable broadcast algorithm ensures that when a process broadcasts a message m that m reaches every process in the network. Data propagation is based on the knowledge of the received message ids (*message information*). For a process p to know that some process q received a message m requires a causal chain between p and q. Depending on the network topology, the causal chain length ranges from *1* to *n-1*, n being the number of processes in the system. To ensure data propagation, a process sends with the messages the ids of the newly received messages, therefore a message acknowledges anterior messages. In a stable system, the following lemma holds.

Lemma 2. All processes receive message m.

Algorithm description. Our reliable broadcast bases its strength on *data propagation* and *Network Membership*. Intuitively, the algorithm works as follows. When a process broadcasts a message m, it sends m to all its neighbours. Each process then verifies if it must forward m to a process which has not yet received m. In order to be efficient, we define p as the *first process* of q for message m (based on the *channelState$_p$*), when p is the neighbour process with the lowest id that has a direct link with q and has received m. In a stable system, for each process, there is only one process p in the whole system that it is the *first process* to q for m. This scheme of *first process* allows processes to forward only the minimum number of messages necessary to achieve reliable broadcast, thus avoiding unnecessary traffic with redundant messages.

Our messages have unique identifiers and contain two fields: a *control* field and a *data* field. *Control* fields consist of message ids sent to neighbour processes to inform them that the process received a message; in fact they correspond to positive acknowledgements. Messages ids are kept on each process in a table called *idMessagesReceived$_p$*. We denote *message information* the set of all *idMessagesReceived$_p$* on a process and it is updated every time a message is received. *Data* fields are application reliable broadcast messages. The primitive *Receive$_p$()* makes the distinction between those two fields: when receiving a message, it updates its *message information* with the acknowledgements that are appended to the message and then treats the *data* field. Based on *network knowledge* and *message information*, a process decides if it needs to forward some message to another process. The function *check&forward$_p$* forces a process p to forward a message m when p is the *first process* of q for m. The function *check&forward$_p$* ensures that missing messages are sent to lagging processes.

For example, suppose the same network topology and local *view* of Figure 2, assume that the process ids are $p_1 < p_2 < p_3 < p_4$ and that changes are instantaneous for all processes. When p_1 broadcasts m, p_1 sends m directly to p_2 and p_3; all processes then check based on their local *view* if they need to forward any messages to some neighbour processes. Process p_2 then forwards m to p_4 since p_2 is the *first process* of p_4 for m. Process p_2 is the *first process* of p_4 for m since it is the lowest id process that received m and has a direct link with p_4. As a second example, assume that the link between p_2 and p_4 breaks and p_2 broadcasts messages $m_2..m_5$. The messages are received and delivered by p_1 and p_3 since they have direct links between them. When the link between p_3 and p_4 comes back up, p_3 updates p_4 with the missing messages ($m_2..m_5$) since p_3 is the *first process* for p_4 for $m_2..m_5$. This shows that even if p_4 was not in the partition at sending time, it received the messages when joining back.

5.3 Performance

We give here some performance measurements of our prototype which were made on two LANs interconnected by Fast Ethernet (100MB/s) on normal working days. The first LAN consisted of 60 SPARCstation 20 (model 502: 2 SuperSPARC CPU, 64Mb RAM, 1Gb Harddisk) machines, and the second one of 60 UltraSUN 10 (256Mb RAM, 9 Gb Harddisk) machines. All stations were running Solaris 2.6, and the message passing layer was running on Solaris JVM (JDK 1.2.1, native threads, JIT). The message objects were of a size of 1Kb in serialised form. Each MSS had ten MHs attached and we vary the number of MSSs in the system. We give here the performance of only MSSs since they were the only one to make forced logs. MHs were fast enough to handle all messages.

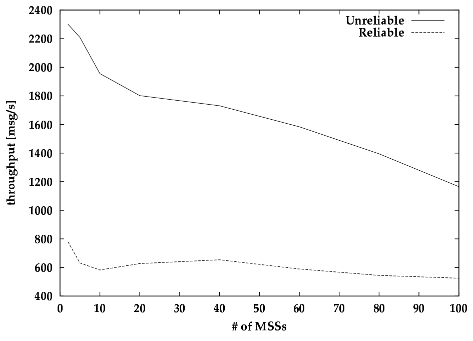
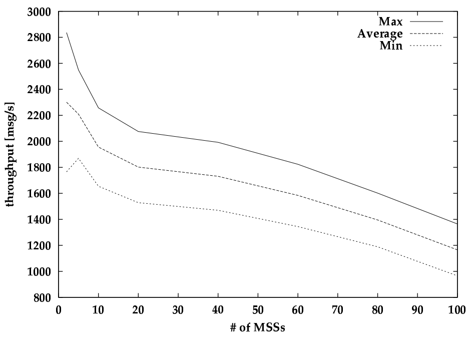


Figure 3. Unreliable & MB-Reliable broadcast

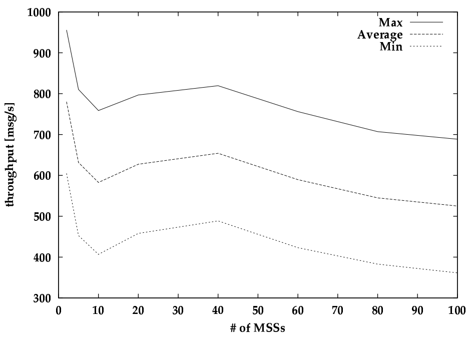
Figure 3 summarises the results of the throughput measurements and compares *MBR-Broadcast* with an unreliable broadcast. Our implementation is made up of the four layers described in Section 3. The different layers communicate through method invocation and listeners for upcalls. All messages are buffered in each layer to avoid network bottleneck. For example, if a message cannot be sent because buffers are full, the *Communication* layer notifies the *Network Membership* layer which itself notifies its upper layer, and so on. Each connection is handled by the Communication layer and one thread handles the *Network Membership* and upper layers.

As conveyed by the measurement results, the performance of *MBR-Broadcast* remains stable over an increasing number of processes. After 100 MSSs, the performance varies very little due to the high overhead of the forced logs into stable storage. On the other hand, the performance of the unreliable broadcast is less stable. It is limited by the overall performance of the network, which is noticeable by the quickly decreasing throughput. Figures 4(a) and 4(b) depict in more details the performance of both broadcast protocols, together with the variance of the measurements.

In the case of the unreliable broadcast, the variation decreases when the number of processes increases, as conveyed by Figure 4(a). This is due to the fact that the performance is bound by the global performance of the network. In the case of Figure 4(b) in return, the variance remains more stable since the delay of the network is negligible compared to the time required for the disk accesses.



(a) Unreliable broadcast



(b) MB-Reliable broadcast

Figure 4. Throughput

6 Concluding remarks

Our *Network Membership* model handles network partitions in a mobile computing environment. It uses an unreliable channel detector in conjunction with data forwarding and stability to achieve reliability in the context of partition. The *Network Membership* layer and its protocol do not make any assumption on the network used to transport their messages (i.e., order or reliable functionalities), except that it must guarantee the absence of Byzantine failures.

The major benefits of this work are, first, to propose a formal specification for a group membership for the mobile computing model, and second, to define a reliable broadcast protocol based on this specification. The model is efficient and not restrictive. It lets the system flow in comparison with *minority-partition* model. The proposed partition model is less restrictive than *minority-partition* model because no consensus is required. The reliable broadcast protocol considers partitions and, then, enables processes to receive messages when they were not part of the partition at sending time.

We foresee many interesting enhancements. First, we want to extend the limits of our *Network Membership* model. We plan to study contributions in the model as local consensus or order relations. Second, it would be interesting to see how we can improve the multicast protocol proposed in [7]. Third, we could use the idea of semantic multicast along the lines of [22] to reduce the number of messages in the network.

References

[1] A. Acharia and B. R. Badrinath. A framework for delivering multicast messages in networks with mobile hosts. *Mobile Networks and Applications (MONET)*, 1(3), 1996.

[2] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P.Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Intl Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[3] O. Babaoğlu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. UBLCS 98-01, Department of Computer Science, University of Bologna University, Apr. 1998.

[4] O. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System support for partition-aware network applications. UBLCS 97-08, Department of Computer Science, University of Bologna University, Oct. 1997.

[5] P. Bhagwat and C. E. Perkins. A mobile networking system based on internet protocol ip. In *Usenix Symposium on Mobile and Location-Independent Computing*, Aug. 1993.

[6] R. Boichat and L. Duchien. Network Membership: How to solve R-broadcast efficiently ? Technical Report 99-12, Laboratoire CEDRIC-CNAM, Paris, Sept. 1999. http://cedric.cnam.fr/duchien/RR99-12.ps.gz.

[7] R. Boichat and L. Duchien. Reliable Broadcast and Multicast tolerant to partitions. In *Proceedings of the 11th IASTED International Conference Parallel and Distributed Computing and System*, Boston, USA, Nov. 1999.

[8] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. Technical Report TR95-1548, Department of Computer Science, Cornell University, Oct. 1995.

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[10] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, Feb. 1991.

[11] S. Deering. Host extensions for ip multicasting. Internet Request for Comments RFC 1112, Aug. 1989.

[12] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[13] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication : A survey of protocols, functions, and mechanisms. *IEEE Journal on Selected Areas in Communications*, 15(3):277–290, Apr. 1997.

[14] C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 282–291, Seattle, WA, USA, June 1997. IEEE, IEEE Computer Society Press.

[15] J. Ioannidis, D. Duchampd, and G. Q. Maguire. Ip-based protocols for mobile internetworking. In *Proceeding of ACM SIGCOMM Symposium on Communication, Architecture and Protocols*, pages 235–245, Sept. 1991.

[16] Internet report. RFC 2002, 1996.

[17] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols. In *Proceedings of the 12th Symposium on Reliable Distributed Systems (SRDS-12)*, pages 2–11, Princeton, NJ, USA, 1993.

[18] A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computer Systems*, 10(1):3–25, 1997.

[19] J. J. Kitsler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[20] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and wide-area backbone failures. CSE-TR 382-98, Department of Eletrcial Engineering and Computer Science, University of Michigan, 1998.

[21] K. Obraczka. Multicast transport protocols : A survey and taxonomy. *IEEE Communications Magazine*, 1997.

[22] J. Pereira, R. Oliveira, and L. Rodrigues. Semantically reliable multicast protocol. In *Proceedings of the 19th Symposium on Reliable Distributed Systems (SRDS-19)*, Nuerenberg, Germany, Oct. 2000.

[23] G. Popek, R. Guy, T. Page, and J. Heidemann. Replication in ficus distributed file systems. In *the Workshop on Management of Replicated Data*, Nov. 1990.

[24] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proc. annual ACM Symposium on Principles of Distributed Computing*, Aug. 1991.