Fault-Tolerant Support for Reliable Multicast in Mobile Wireless Systems: Design and Evaluation

(Extended Abstract)

Giuseppe Anastasi
Dip. di Ingegneria dell'Informazione, Università di Pisa
Via Diotisalvi 2, 56122 Pisa, Italy
Alberto Bartoli
Dip. di Elettrotecnica, Elettronica e Informatica, Università di Trieste
Via Valerio 10, 34100 Trieste, Italy
Flaminia L. Luccio
Dip. di Scienze Matematiche, Università di Trieste
Via Valerio 12/1, 34127 Trieste, Italy

July 10, 2002

Abstract

In this paper we present a protocol for reliable multicast within a group of mobile hosts that communicate with a wired infrastructure by means of wireless technology. The protocol assumes that the wireless coverage may be incomplete and message losses could occur even within cells, due to physical obstructions or to the high error rate of the wireless technology, for example. A novel feature of our proposal is that it tolerates failures in the wired infrastructure, i.e., crashes of stationary hosts and partitions of wired links. In particular, upon such failures mobile hosts simply observe that the covered area has shrunk. The covered area will enlarge back to its original extension when the failure recovers. We evaluate our protocol by means of an extensive simulation analysis. The main findings of our analysis are: the protocol exhibits very good scalability properties; the performance degradation induced by fault-tolerance is moderate.

Keywords: Group communication, replication, incomplete coverage, hand-off, total order.

1 Introduction

Computing architectures based on *portable computers* and *wireless networking* are becoming a reality. Users may be equipped with hand-held computing devices and roam around freely while maintaining connectivity with a wired computing infrastructure through a number of wireless cells.

Mobile wireless systems typically require special solutions, for a number of reasons. Traditional network protocols implicitly assume that hosts do not change their physical location over time. Mobile devices have severe resource constraints in terms of energy, processing and storage resources. Wireless networks are characterized by limited bandwidths and high error rates. Furthermore, mobility introduces new issues at the algorithmic level. For example, a mobile host may miss messages simply because of its movements, even with perfectly reliable communication links and computers that never crash [1]. All the above reasons imply that specialized protocols are required for extending to mobile hosts functionalities common for stationary ones.

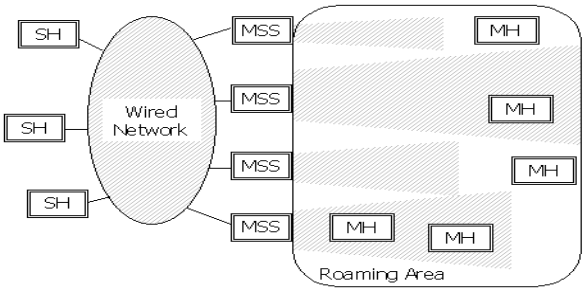

Figure 1: Example system with five MHs and seven SHs. Four SHs are MSS. Wireless cells are indicated as dashed areas within the roaming area. Notice that portions of the roaming area are out of coverage.

In this paper we present a protocol for *reliable* and *totally-ordered* multicast within a group of mobile hosts. By this we mean that: (i) each mobile host delivers all multicasts, without duplicates; and (ii) any two mobile hosts that deliver two multicasts deliver these multicasts in the same order.

Reliable and totally-ordered multicast is an important building block for applications composed of remote processes that have to cooperate tightly [16]. This communication primitive has proven its power in the context of traditional, i.e., static and wired, distributed computing. Our proposal makes this primitive available on mobile wireless systems. Moreover, we support this primitive in spite of (a certain number of) crashes of stationary hosts and partitions of wired links. The fault-tolerance properties of our protocol may greatly extend the scope of potential applications of mobile computing, including emergency management, plant control, traffic monitoring, stock market exchange, on-site data collection, for example. Fault-tolerant support for mobile wireless systems is, in our opinion, an important topic, yet it has not received much attention from the research community so far.

We model a mobile wireless system as follows (see Figure 1). There is a set of *stationary hosts* (SHs) connected by a wired network and a set of *mobile hosts* (MHs) that may move and communicate through wireless links. Some SHs, called *mobile support stations* (MSSs), may communicate also through wireless links. Each MSS defines a spatially limited *cell* covered by a wireless link. A MSS may broadcast messages to all MHs in its cell and send messages to a specific MH in its cell, whereas a MH may only send messages to the MSS of the cell where it happens to be located. Notice that we do not assume any network support for routing messages to a specific MH.

An important feature of our model is the *incomplete coverage* of wireless cells, i.e., MHs may roam in areas that are not covered by any cell. A MH may move across adjacent cells but it may also "disappear" within the uncovered area and enter any other cell, perhaps after a "long" time. Movements occur without prior negotiation. The resulting scenario is quite general because it accommodates contemporary wireless LAN's, infra-red networks requiring line-of-sight connectivity, disconnected modes of operation, long-range movements, picocellular wireless networks in which the cell size is of the order of a few meters, such as a room in a building.

The message pattern of our protocol follows common approaches for reliable multicasting among MHs in mobile wireless systems [1, 3, 7, 18, 23]. A MH wishing to issue a multicast sends a request to the MSS of the cell where it happens to be located. The MSS forwards the message to a SH that processes this request, includes the payload, and forwards the payload to all MSSs. MSSs broadcast the payload in the respective cell. Unlike earlier proposals, however, our protocol has no notion of *hand-off*: when a MH switches cell, there is no interaction between the MSSs of these cells. Full details will be given later.

Our work is based on a design philosophy aimed at improving reliability of final applications, in particular, with respect to failures:

1. Availability of MSSs must affect only availability of applications, not their correctness. In particular, a MSS failure should merely shrink the covered area, without affecting correctness. That is, the protocol should still deliver all messages, in total order and exactly-once, even if a MSS crashes.

2. The state shared among SHs should not be updated upon *each* movement of MHs. Otherwise, performance could be penalized and failure handling would be more difficult.

3. One should avoid to assume that the wireless coverage is complete. Otherwise, even a single physical obstruction, or particularly unfortunate area, or MSS malfunctioning, could compromise correctness.

4. One should avoid to make hypothesis on users' movements. Otherwise, even a *single* inopportune movement could compromise correctness.

5. Critical state information should not be kept on MSSs, but on "ordinary" SHs. This choice allows using systematic and established techniques for improving the availability of these hosts, such as replication based on group communication [10].

6. MSSs should be freely added or removed without stopping the system or compromising correctness. MSS addition may be necessary for upgrading or coverage enhancement, whereas MSS removal for maintenance or failure.

Our approach satisfies all the above requirements (note that they may be applied to mobile computing in general, not only to the specific problem of reliable multicast). In particular, a failure in the wired infrastructure may simply provoke a shrinkage in the covered area, that will expand to its original extension when the failure recovers. For example, if a MSS crashes, then all MHs that happen to be in the cell of the crashed MSS will become temporarily unable to communicate. They will be able to communicate again when either the MSS recovers, or they move to another cell.

We have analyzed the performance of the proposed protocol by simulation both in normal operating conditions (i.e., in the absence of failures) and in the presence of failures at MSSs and SHs. We found that the proposal is indeed practical. In normal operating conditions the impact of fault-tolerance on the performance of the protocol is moderate. The increases in the (average) message latency is of only few milliseconds with respect to the non fault-tolerant version of the protocol.

2 System model

Each wired link and each wireless cell provides FIFO-ordered communication without duplicates. Messages may be lost. Message loss in a wired link occurs as a result of network partitions. Such partitions may recover. Message loss in a wireless cell may occur because of physical obstructions or because of the intrinsic features of wireless technology, e.g., high error rate. Hosts communicate solely via messages. Of course, while a MH is out of coverage no communication with it is possible. Similarly, SHs partitioned from each other cannot communicate among themselves. SHs may crash and a crashed SH may recover. MHs do not crash (see also below).

The system is asynchronous in the sense that neither message delays nor computing speeds can be bounded with certainty. This characterization is a general and realistic one as it allows abstracting away such features as variable loads imposed by users and unknown scheduling strategies on hosts and communication links. Notice that a process cannot determine with certainty whether a remote process that appears to be unresponsive has crashed or happens to be very slow.

It will be clear from the protocol description that the protocol is resilient also to crashes of MHs. The only problem is that state information about a crashed MH would never be discarded by SHs. In practice, however, SHs might unilaterally garbage-collect information not accessed for a very long time. Of course, a MH deemed crashed might reappear suddenly from the uncovered region, for example. On the other hand, there is no way to exclude such possibilities in asynchronous systems (unless one is willing to wait for an infinite time before deciding whether a remote process actually crashed). Another practical issue is that a crashed MH should be able to participate again in the application after its recovery. This feature may be achieved by: (i) supporting a dynamic group of MHs that may exchange multicasts; and (ii) requiring that multicasts be delivered only by current members of the group. The protocol proposed here assumes a static set of MHs but it may be extended toward supporting (i) and (ii) quite simply [6].

3 Related work

To the best of our knowledge, the only work with scope similar to ours is [2]. This work introduces resilience to failures of MSSs in a non fault-tolerant reliable multicast protocol proposed by Acharya and Badrinath [1] (see also below). However the system model is much more restrictive than ours because: (i) communication is reliable, both in the wired network and in wireless cells (in particular, it does not allow message loss thereby excluding, for example, physical obstructions within cells and partitions of wired links); and (ii) it assumes that a process can detect with certainty whether a remote process is active or crashed (fail-stop failures).

The protocol in [2] adds fault-tolerance to the one in [1] by associating each MH with a set of MSSs, denoted $\mathcal{S}$(MH), and by replicating state information about that MH at each member of $\mathcal{S}$(MH). Whenever a MH sends a message or there is a message addressed to it, members of $\mathcal{S}$(MH) have to execute a replica control protocol and this protocol must be able to tolerate host failures. Note that it is necessary to execute one instance of the replica control protocol for *each MH*, whereas in our proposal one instance of a replica control protocol suffices for *all* MHs. The cited work mentions two alternatives for such protocol. The one that is more efficient requires additional mechanisms (network flush or rollback) that are not detailed. Furthermore, no performance analysis is provided and the complex interaction among (i) replica control protocol, (ii) MSS recovery, and (iii) hand-off, are only outlined [1]. Our protocol is fully detailed and, in our opinion, is much simpler to implement.

The protocol by Acharya and Badrinath, hereinafter the AB-protocol, was the first multicast protocol ensuring reliable (FIFO) delivery in the context of mobile computing and has been highly influential in the design of later protocols [3, 7, 18, 23]. Although none of these protocols is fault-tolerant, it is useful to discuss them briefly to emphasize the differences with our proposal. Each MSS maintains, for each MH in its cell, an array of sequence numbers describing the multicasts already delivered by that MH. The MSS uses this array to forward pending messages in sequence and without duplicates. If the MH switches cell, the array is moved to the new MSS by means

[1] The paper claims that the composition of $\mathcal{S}$(MH) may change dynamically, but it appears that this issue has been oversimplified, in particular, with respect to the interaction just mentioned.

of a proper *hand-off* procedure. Therefore: (i) The state shared among SHs is updated upon *each* movement; (ii) MSSs maintain critical state information (i.e., each MSS remembers the sequence numbers of multicasts delivered by each MH in its cell); and (iii) each MSS is a single point of failure (i.e., if a MSS crashes, the above sequence numbers are lost for each MH in the cell). These features explain why the fault-tolerant extension in [2] requires a complex interaction among several sub-protocols.

The AB-protocol and the protocols derived from it assume reliable communication in the wired network and in the wireless network. As pointed out about [2], this assumption excludes partitions of wired links and appears to exclude uncovered regions and physical obstructions within cells, that are instead part of our system model.

The mobility assumptions of the protocols being discussed refer to each *single* move, e.g., a MH must remain in a cell long enough to complete each hand-off. In contrast, our protocol depends only on *global* assumptions, i.e., of the form "a group member does not move very fast, all the time". Movements at inopportune times can only cause an occasional performance penalty and do not affect correctness. Furthermore, such penalty disappears when the MH stops moving "too fast".

Note that the AB-protocol provides reliable delivery without requiring routing support for MHs, e.g., Mobile IP, much like our proposal. Multicast protocols that rely on Mobile IP are generally targeted at different application domains and provide unreliable, best-effort, unsequenced delivery [14, 22]. In particular, no messages are delivered during a cell switching and messages possibly lost will not be recovered in the new cell. With respect to the use of Mobile IP, note also that: (i) it would not solve the problem of recovering from lost messages; (ii) it would make it more difficult to exploit the broadcast capabilities of the wireless medium when many MHs are in the same cell; (iii) it would generate traffic in the wired network even while no new multicasts are generated, for tracking the location of each MH.

The protocol proposed here is an extension of the protocol in [9] that was not fault-tolerant and assumed reliable communication in the wired network. As an aside, performance analysis by simulation showed that the proposal in [9] outperforms the AB-protocol in terms of latency, scalability, bandwidth usage efficiency and quickness in managing cell switches of users [4]. The proposal in [9], as well as the one in this paper, borrows a crucial idea from the implementation of reliable multicasts in "static and wired" distributed systems: the use of a centralized sequencer for totally ordering multicasts and for storing multicasts that have not been acknowledged yet [16]. Note, however, that here we refer to a completely different system model: mobile hosts, wireless communication, incomplete spatial coverage.

Finally, other forms of fault-tolerant support for mobile hosts were proposed in [15, 17, 19]. All such supports refer to problems and operating environments different from ours. Support for a FIFO channel from a MH to a (fault-tolerant) SH is considered in [15]. Checkpointing and recovery from node failures is considered in [19], in the hypothesis of reliable communication and fail-stop node failures (i.e., failures are reliably detected). A similar problem is considered in [17], but without addressing failures of SHs.

4 Overview of the protocol

We begin by briefly outlining the non fault-tolerant version of the protocol. Messages have a field of enumerated type, called tag and indicated in SmallCaps, that indicates the purpose of the message. We say that a host H *receives* a message m when m arrives at the protocol layer at H, and that H *delivers* m when the protocol forwards m up to the application.

A MH wishing to issue a multicast sends the payload to the local MSS with a NEW message. MH retransmits this message until receiving an acknowledgment (possibly from a different MSS, if the sending MH moves during the handshake). The MSS forwards the message to a designated SH acting as *coordinator*. A NEW message carries a sequence number locally generated by the sending MH, which enables the coordinator to process NEW messages in sequence and to discard duplicates. The coordinator constructs a NORMAL message containing the payload of the NEW message and a locally generated sequence number. The resulting message is then multicast to MSSs that broadcast it in the respective cell. Each MH uses sequence numbers of NORMAL messages to deliver these messages in sequence without duplicates (i.e., in total order) and to detect missing messages. In the latter case, the MH sends a retransmission request to the local MSS. This request is tagged NACK and specifies an interval of missing sequence numbers. When a MSS receives a NACK, it relays the missing NORMAL messages to the sending MH. The MSS obtains such messages from a local *cache* or, in case of a miss, from the coordinator. MSS requests missing messages to the coordinator with a FETCHREQ specifying an interval of sequence numbers. The coordinator responds with a FETCHREP containing the required messages. A NACK from a MH implicitly acknowledges delivery of previous multicasts. MSSs extract this information and forward it to the coordinator, with STABINFO messages. Note that MSSs do not store critical state information: such information is kept by the coordinator and merely cached by MSSs for efficiency. Note also that each MSS reacts to cell switching without interacting with other MSSs.

The *fault-tolerant* extension proposed here is obtained as follows.

1. A MH no longer assumes that a message arrived at a MSS will eventually arrive at the coordinator — the MSS might crash, or a partition might occur. Instead, a MH keeps on retransmitting a NEW message until receiving the matching NORMAL message (a MSS that receives a NEW message does not respond to the sending MH with an acknowledgment, as this acknowledgment would be useless).

2. The role of the single coordinator is played by *a set* of SHs, called *coordinators*. This set appears to MSSs as a single "coordinator service". The service is available in spite of (a certain number of) failures of coordinators and connecting links. In particular, availability of the service requires a majority of coordinators. Coordinators interact among themselves through *group communication (GC)* [10]. GC may be thought of as a software layer exporting to applications a membership service and a communication service for reliable multicasting within a group of processes. These two services are tightly integrated so as to simplify the programming of distributed algorithms in the face of host crashes and recoveries, network partitions and mergers. More details will be given in section 5.5.

3. A MSS sends its messages to a designated coordinator, say C. The MSS might not receive a response for several reasons, including: (i) C is not able to interact with a majority of coordinators (section 5.5); (ii) the message from MSS to C is lost; (iii) the response from C to MSS is lost. Should a response not arrive within a specified timeout, the MSS will send the *next* request to another coordinator. The request not yet answered will be retransmitted by the originating MH, as pointed out above (1). The policy for associating coordinators with MSSs is irrelevant to this paper. Of course, timeouts expiring too soon must not affect correctness. To this end, the coordinator service maintains internally information sufficient to detect duplicate requests (section 5.5).

mid, G: process-id;
MH-buffer: **set of** NORMAL messages;
norm-seq, new-num, mylow: integer;
new-last: message;
timer-new, timer-nack: timer;
	Procedure MH-Init();
	mid := some-unique identifier;
	G, new-last := null;
	MH-buffer := $\emptyset$;
	norm-seq, new-num, mylow := 0;
	TimerResetNS(timer-new);
	TimerResetNS(timer-nack);

Table 1: Variables maintained at each MH and the initialization procedure MH-Init().

5 The algorithm

5.1 Notation

We use a self-explanatory pseudocode where blocks are sometimes delimited implicitly by indentation levels, and sometimes explicitly by curly brackets. We use several ancillary procedures and functions. Names ending with NS are used for procedures or functions whose body is not shown.

TimerSetNS(timer) schedules a time-out on the specified timer (the minimum time interval after which the time-out should occur is irrelevant to our discussion and omitted). If there is already a scheduled time-out on that timer, then the procedure does nothing. For simplicity, we model the expiration of a time-out as the receiving of a message. TimerResetNS(timer) cancels the time-out possibly scheduled on the specified timer. TimerIsSetNS(timer) returns the boolean true iff there is a scheduled timeout on the specified timer.

The following operations deal with sets of NORMAL messages. We want to enforce the following constraint: the aggregate size of the messages stored in a set cannot exceed a predefined limit (except at coordinators, where there is no such limit). This is because such sets are meant to be used as caches. CachePutNS(set, msg) inserts message msg in the specified set. The operation may discard messages stored in the set to make space for msg, if necessary. CachePutTryNS(set, msg) is similar, except that msg may or may not be inserted in the set, depending on the amount of space available and the messages currently stored. Notice that we are not concerned with the replacement policy of the cache: this issue is irrelevant to this paper. CacheSearchNS(set, seq) returns the message in the set with sequence number seq, or null if there is no such message. CachePurgeNS(set, seq) is similar, except that the returned message is removed from the set, thereby freeing the memory occupied by the message.

5.2 Beaconing

A MSS becomes aware of which MHs are in its cell by means of a *beaconing* protocol. Essentially, a MSS broadcasts a "greeting" message periodically and, upon receiving such message, a MH responds thereby notifying its presence in the cell. Each MSS uses the beaconing protocol for maintaining a local variable, that we call l-cell, containing the identifiers of the MHs in the cell. Upon receiving a message, MSS checks whether the sender of the message belongs to l-cell and, if not, it discards the message without further processing. A MSS uses l-cell only for allocating and deallocating local data structures. It follows that location information across the network need not be fully consistent or accurate. For example, a MH could temporarily belong to l-cell of multiple MSSs. Our requirements to the beaconing protocol are detailed in [9] and can be summarized as

follows: (i) if a MH remains in a cell for sufficiently long, then eventually MH will belong only to the l-cell of the MSS of that cell; (ii) if a MH remains out of coverage for sufficiently long, then eventually MH will not belong to any l-cell.

5.3 MH side

Each MH maintains the variables listed in table 1. Mid is the identifier of the MH. G is the identifier of the MSS that covers the cell where MH happens to be located, or null, if MH is out of coverage (G is maintained with the beaconing protocol). MH-buffer is a set of received NORMAL messages that cannot be yet delivered because otherwise total order would not be satisfied. Norm-seq contains the sequence number of the last NORMAL message delivered, new-num the (locally generated) sequence number of the last NEW message sent, mylow the smallest sequence number of the messages in MH-buffer. New-last contains the last NEW message sent. Timer-new and timer-nack are used for controlling retransmissions of NEW and NACK messages, respectively. All variables are initialized by MH-Init() (table 1).

A MH executes an endless loop (table 2) in which, at each iteration, it invokes RECEIVE() and handles the received message accordingly. A MULTICAST request from the application is handled by sending the payload to the MSS in a NEW message and by setting timer-new (lines 5-10). For ease of presentation, we assume that the application may issue a further MULTICAST only after delivering the previous one. Removing this assumption is not difficult. Upon receiving a NORMAL message, MH checks that the message is not a duplicate by inspecting the system-wide sequence number selected by coordinators (cseq field, lines 11-13). Then, MH executes procedure HandleNormal() that buffers the message (23-24) (UpdateMylowNS() sets mylow to the smallest cseq field, i.e., to the system-wide sequence number assigned by the coordinators, of messages in MH-buffer, or to 0 if MH-buffer $= \emptyset$), and delivers all buffered messages that can now be delivered, if any (26-30). If the received message was originated by MH itself, then timer-new is cleared (25); when timer-new expires MH resends the last NEW message (14-16). If there is a buffered message that cannot be delivered, then MH requests a retransmission by sending a suitable NACK message (32-34). The MSS will retransmit the messages with sequence number in between the second and third field of the NACK (see next section). Note that the second field of a NACK contains implicit acknowledgment information. Upon sending the NACK, MH sets timer-nack. As an optimization, later executions of HandleNormal() may send further retransmission requests only if timer-nack is not set already. When timer-nack expires, MH constructs and sends a further NACK message (17-19).

5.4 Mobile Support Station side

Each MSS maintains the variables listed in table 3. My-id is the identifier of the MSS. C-team is a set that identifies the coordinators. This set is defined statically and never changes. C is the member of C-team with which MSS communicates. L-cell is a set that identifies the MHs currently covered by the MSS (this set is updated by the beaconing protocol). Timer-c is a timer that monitors the responsiveness of C. Normal-cache is a set used as a cache of past NORMAL messages. Miss-table is a set with one element for each MH being brought up-to-date, i.e., by replaying messages specified with a NACK. Each element is a record, whose fields are: mid, that identifies the MH; curr, that is the sequence number of the last message sent to mid; low and high, that are the minimum and maximum sequence numbers requested by mid; hole-low and hole-high, that delimit the sequence numbers of messages not available in Normal-cache and that have been requested to C (see also below). All variables are initialized by the procedure MSS-Init() (table 3).

```
1 Main Program
2 MH-Init();
3 repeat msg := RECEIVE();
4  multi-if
5     ∎  msg.tag= MULTICAST → /* from the application,
6                              < MULTICAST, M > */
7        new-num := new-num +1;
8        new-last :=< NEW,msg.M,new-num,mid >;
9        SEND(new-last,G);
10       TimerSetNS(timer-new);
11    ∎  msg.tag= NORMAL → /* from G, < NORMAL,M,cseq,mid >*/
12       if (msg.cseq > norm-seq) and (msg ∉ MH-buffer)
13            then  HandleNormal(msg);
14    ∎  msg is a timeout from timer-new →
15       SEND(new-last,G);
16       TimerSetNS(timer-new);
17    ∎  msg is a time-out from timer-nack →
18       SEND(<NACK,norm-seq + 1,mylow - 1,mid>,G);
19       TimerSetNS(timer-nack);
20end-if
21 forever
22 Procedure HandleNormal(msg);
23CachePutNS(MH-buffer, msg);
24UpdateMylowNS();
25if msg.mid = mid then TimerResetNS(timer-new);
26while ∃ m ∈ MH-buffer: m.cseq=norm-seq +1 do
27     m := CachePurgeNS(MH-buffer, norm-seq+1);
28     UpdateMylowNS();
29     DELIVER-TO-APPLICATION(m);
30     norm-seq := norm-seq +1 end-do;
31if MH-buffer = ∅ then TimerResetNS(timer-nack)
32   else if not TimerIsSetNS(timer-nack) then
33     SEND(<NACK,norm-seq + 1,mylow - 1,mid>,G);
34     TimerSetNS(timer-nack);
```

Table 2: Algorithm executed by MH.

```
my-id, C: process-id;
C-team, l-cell: set of process-ids;
timer-c: timer;
Normal-cache: set of messages;
miss-table: set of record
            mid: process-id;
            curr: integer;
            low, high: integer;
            hole-low, hole-high: integer;

Procedure MSS-Init()
my-id := unique identifier;
C-team := statically defined set;
C := pick up a member of C-team;
l-cell, Normal-cache, miss-table := ∅;
TimerResetNS(timer-c);
```

Table 3: Variables maintained at each MSS and the initialization procedure MSS-Init().

Each MSS executes an endless loop (table 4). At each iteration it checks whether a MH being brought up-to-date has left the cell, in which case it drops the corresponding element from miss-table (line 4; we omit the obvious synchronization for accessing l-cell, that may be concurrently updated by the beaconing protocol). Then the MSS executes RECEIVE() and handles the received message accordingly. Upon receiving a NEW message, MSS forwards the message to C and sets timer-c (7-8). Timer-c is reset upon receiving a message tagged either ACK, or NORMAL, or FETCHREP. A NORMAL message is multicast in the cell and possibly stored in Normal-cache (lines 11-14).

Upon receiving a NACK message, MSS allocates a miss-table entry, forwards the relevant stability information to C and initiates the state transfer (lines 15-23). MSS could receive further NACKs from a MH already associated with a miss-table entry. The freshness of each NACK is determined by inspecting its low field (line 20), i.e., the smallest sequence number requested by the sending MH. Procedure StateTransfer(x) (table 5) sends to the requesting MH all messages that can be found in Normal-cache, in order. If a message is not available, then a FETCHREQ message specifying an interval of missing sequence numbers is sent to C (line 12). StateTransfer() terminates when either all messages have been sent, in which case the related miss-table entry is cleared (14), or when a FETCHREQ has been sent. The response to a FETCHREQ is a FETCHREP message (line 24 of table 4), whose miss-req field contains a copy of messages with sequence numbers in [l,h]. MSS stores such messages in Normal-cache and resumes all the suspended sequences that are waiting for such values, calling the StateTransfer() procedure for each one of them (25-28). Finally, upon receiving a timeout on timer-c (29-30), MSS reassigns C by selecting another member of C-team (of course, one could keep a copy of the last unacknowledged messages sent to C, retransmit them and pick up a new C after a number of retries).

5.5 Coordinator side

Interaction among coordinators occurs through *group communication (GC)*. A detailed description of GC is beyond the scope of this paper (see [10, 20] for example). Below we shall provide only some background. GC is implemented by a dedicated software layer at each coordinator.

Coordinators form a *group* (this notion of group has nothing to do with the group of MHs). The GC layer provides consistent information about the set of coordinators that appear to be currently reachable. This information takes the form of *views*. The GC layer determines a new view as a result of crashes, recoveries, network partitions and merges. New views are communicated to coordinators automatically, through special messages called *view changes*. When a coordinator C receives a view change carrying the new view V, C is informed that it can communicate with the

```
1 Main Program
2 MSS-Init();
3 repeat
4   foreach x ∈ miss-table do if x.mid ∉ l-cell then miss-table := miss-table - x;
5   msg := RECEIVE();
6     multi-if
7       ∎ msg.tag = NEW → /* from MH, < NEW, M, new-num, mid > */
8           SEND(msg,C); TimerSetNS(timer-c);
9       ∎ msg.tag = ACK → /* from coords, < ACK > */
10          TimerResetNS(timer-c);
11      ∎ msg.tag = NORMAL → /* from coords, < NORMAL, M, cseq, mid > */
12          MULTICAST(msg) in cell;
13          CachePutTryNS(Normal-cache,msg);
14          TimerResetNS(timer-c);
15      ∎ msg.tag = NACK → /* from MH, < NACK, low, high, mid > */
16          x := element of miss-table : x.mid=msg.mid;
17          if x = null then
18              allocate entry x ∈ miss-table;
19              x.mid := msg.mid; x.low := 0;
20          if msg.low ≥ x.low then
21              x.low, x.curr := msg.low; x.high := msg.high;
22              SEND(< STABINFO,msg.low,msg.mid >, C);
23              StateTransfer(x);
24      ∎ msg.tag = FETCHREP → /* from coords, < FETCHREP, l, h, miss-req > */
25          TimerResetNS(timer-c);
26          foreach m ∈ msg.miss-req do CachePutNS(Normal-cache,m);
27          foreach x ∈ miss-table do
28              if x.curr ∈ [msg.l, msg.h] then StateTransfer(x);
29      ∎ msg.tag is a timeout from timer-c →
30          C := pick up another member of C-team;
31    end-if
32 forever
```

Table 4: Algorithm executed by MSS.

```
1 Procedure StateTransfer(x)
2 msg := CacheSearchNS(Normal-cache, x.curr);
3 while msg ≠ null and (x.curr < x.high + 1) do
4   SEND(msg, x.mid);
5   x.curr := x.curr + 1;
6   msg := CacheSearchNS(Normal-cache, x.curr);
7 end-do;
8 if x.curr < x.high + 1 then
9   x.hole-low := x.curr; x.hole-high := x.high;
10  k := minimum sequence number in Normal-cache : k > x.hole-low;
11  if k ≠ null and k < x.high then x.hole-high := k − 1;
12  SEND(< FETCHREQ,x.hole-low,x.hole-high,my-id >, C);
13  TimerSetNS(timer-c);
14 else miss-table := miss-table - x;
```

Table 5: Procedure StateTransfer().

coordinators listed in V. To proceed further, we need a few simple definitions: (i) C *installed* a view V means that C indeed received the corresponding view change; (ii) two views V, W are *consecutive* means that a coordinator installs V and then installs W; (iii) the view that is *current* at C is the one specified by the last view change received by C; (iv) C delivers message m *in view* V means that C delivers m when the view that is current at C is V.

The key guarantee of GC is that view changes are globally ordered with respect to the receiving of multicasts: *Given two consecutive views V and W, any two coordinators that install both views must have received the same set of multicast messages in view V.* For example, consider a coordinator C_1 that delivered V and suppose W is delivered as a result of the crash of C_1. If C_1 crashed while performing a multicast m, then (i) all coordinators that install V and W receive m (and do so *before* installing W); or (ii) none of them receives m. Clearly, this property is very powerful for reasoning about fault-tolerant algorithms.

The GC layer supports *partitionable* membership, i.e., it allows multiple views of the group to exist concurrently, to model network partitions. Moreover, the GC layer supports *uniform multicast*: if any member of view V delivers multicast m, then each member of V delivers m or crashes. We present the algorithm in the hypothesis that a view including a majority of coordinators always exists. The algorithm may be extended to accommodate the more general case in which the majority view temporarily disappears.

Each coordinator maintains the variables listed in table 6. MSS-team identifies the set of MSSs, that is defined statically and never changes. My-id identifies the coordinator. Threshold is the minimum number of view members that defines a majority view (for simplicity, we assume that the number of coordinators is odd). V contains the composition of the current view whereas old-v the composition of the previous current view. The remaining variables are collected in a record called coord-state. This record contains the variables that have to be transferred to coordinators that enter the majority view. The value of this record is meaningful only in the majority view and after the state transfer has been completed. For simplicity, in the following we shall refer to the fields of coord-state by their names, i.e., by omitting the name of coord-state. Boss identifies a

```
MSS-team, v, old-v : set of process-ids;
my-id : process-id;
threshold : integer;
coord-state : record
    boss: process-id;
    cseq : integer;
    Normal-buffer: set of messages;
    member-table: set of record
        mid: process-id;
        new-num: integer;
        cseq-mid: integer;
        delivered: integer;
```

```
Procedure Coord-Init()
MSS-team := statically defined set;
my-id := some unique identifier;
threshold := predefined number;
v, old-v, boss := null;
cseq := 0;
Normal-buffer := ∅;
member-table := statically defined table;
foreach x ∈ member-table do
    x.new-num := 0;
    x.delivered := 0 end-do;
```

Table 6: Variables maintained at each coordinator and the initialization procedure Coord-Init().

designated view member (see below). Cseq is the sequence number of the last NORMAL message sent. Normal-buffer is a set containing all NORMAL messages that might not be *stable*, i.e., that are not known to have been delivered by each MH. Member-table has one element for each MH. Each element is a record whose fields are: mid, that identifies the MH; new-num is the sequence number (generated by the MH) of the last NEW message received from mid; cseq-mid is the cseq assigned to the last NORMAL message generated upon processing a NEW sent by mid; delivered, is the highest sequence number of a NORMAL message that has certainly been delivered by mid.

Each coordinator executes Coord-Init() (table 6), joins the group and then executes CoordControl() (table 7) that never returns. This procedure waits for delivery of a majority view. Then, SynchronizeNS() brings the state of the coordinator up-to-date (see also below). Finally CoordService() (table 8) implements the actual processing of messages from MSSs[2]. CoordService() returns upon a view change (line 36). If the new view is still a majority, then SynchronizeNS() and CoordService() are executed again. Otherwise, the coordinator waits again for delivery of a majority view, in particular, by ignoring messages possibly received from MSSs.

CoordService() consists of an endless loop in which at each iteration the coordinator executes RECEIVE() and then processes the returned message or view change. NEW or FETCHREQ messages arriving from MSS are replied with an ACK message (line 3). Then NEW messages are forwarded to the boss (6) that multicasts them within the majority view (7; a different tag, COORDNEW, is used only for sake of clarity). Upon receiving a COORDNEW, each coordinator updates the local coord-state as appropriate and the boss multicasts the associated NORMAL message to MSSs (8-20). Duplicate NEWs are ignored because of the check on the new-num field (line 10). Should the boss crash before multicasting to MSSs, MHs will eventually detect a hole in the stream of sequence numbers and ask retransmission. Since communication among coordinators provides uniformity, it is guaranteed that the surviving coordinators can indeed honor the request. Notice that a duplicate COORDNEW is ignored, except when it is a duplicate of the *last* NEW message by the sending MH. In this case, the associated NORMAL message $m1$ is resent (lines 17-20), because the sending MH might have missed $m1$ (a MH keeps on sending a NEW until receiving the matching NORMAL).

FETCHREQ messages are processed locally and the matching FETCHREP is constructed by using the Normal-buffer (23-26). In principle, the FETCHREQ might refer to a NORMAL message still in transit from the boss to the coordinator that is processing the FETCHREQ. In this (unlikely) case,

[2]For space reasons, we omit the obvious details for creating the first majority view upon bootstrapping.

```
1 Procedure CoordControl()
2 repeat
3     msg := RECEIVE();
4     if msg is a view change then { old-v := v; v := msg; }
5     while #members of v ≥ threshold do {
6           SynchronizeNS();
7           if #members of v ≥ threshold then CoordService(); }
8 forever
```

Table 7: Procedure CoordControl.

the FETCHREQ is forwarded to the boss (22). A STABINFO message is multicast within the majority view (such a message describes the NORMAL messages certainly delivered by a specified MH) with the new COORDSTABINFO tag (27-29). Upon receiving this multicast, each coordinator records the related information in the pertinent entry of member-table and clears from Normal-buffer messages that have been delivered by every group member.

Procedure SynchronizeNS() installs the same (up-to-date) coord-state to all members of the majority view. It performs the following actions (we do not show the pseudocode for brevity). Coordinators whose previous view was a majority assign a variable u-set to the intersection between the current view v and the previous view (u-set is a shorthand for "up-to-date-set"). The others assign u-set to null. Then, they all enter a loop in which at each iteration:

1. Coordinators in u-set check whether boss is a member of v; if not, they elect another boss by applying a deterministic function to u-set.

2. The boss sends the current value of coord-state to members of v that are not in u-set.

3. Each member of v sends a COORDALIVE message to the boss (members of v that are not in u-set send this message after acquiring the current coord-state).

4. Having received a COORDALIVE from every member of v, the boss multicasts a COORDSYNCH.

5. Upon receiving this message, the procedure returns.

If a further view change occurs during the above steps, there are two cases, as follows. If the new view is not a majority, then SynchronizeNS() returns and CoordControl() is not executed (table 8). Otherwise, u-set is intersected with the composition of the new view and the loop is repeated.

Notice that CoordService() returns upon *every* view change, e.g., even when the new majority view is a subset of the previous view and no state transfer would be necessary. The reason is as follows. A GC layer supporting partitionable membership should allow the existence of concurrent views whose compositions overlap, otherwise the GC layer itself might block [8]. It follows that, with practical GC layers, there could exist multiple concurrent majority views, even though at most one of them may have been indeed received by a majority. CoordService() returns upon every view change, thus provoking execution of SynchronizeNS(), to make sure that a majority indeed received the current view.

```
1 Procedure CoordService()
2 repeat msg := RECEIVE();
3   if msg.tag = NEW or FETCHREQ and sender is MSS then SEND(< ACK >, MSS);
4   multi-if
5     ■ msg.tag = NEW → /* from MH through MSS or coords, < NEW, M, new-num, mid > */
6       if my-id ≠ boss then SEND(msg,boss);
7         else { change tag of msg to CoordNew; UniformMulticast(msg); }
8     ■ msg.tag = COORDNEW → /* from CB, structure like NEW */
9       m-elem := element of member-table : m-elem.mid = msg.mid;
10      if msg.new-num = m-elem.new.num + 1 then
11          cseq := cseq + 1;
12          n-msg := < NORMAL, msg.M, cseq, msg.mid >;
13          CachePutNS(Normal-buffer, n-msg);
14          m-elem.new-num := m-elem.new-num + 1;
15          m-elem.cseq-mid := cseq;
16          if my-id = boss then Multicast(n-msg, MSS-team);
17        else if m-elem.new-num = msg.new-num then
18          if my-id = boss then
19            n-msg := CacheSearchNS(NormalBuffer, m-elem.cseq-mid);
20            Multicast(n-msg, MSS-team);
21    ■ msg.tag = FETCHREQ → /* from MSS or coords, < FETCHREQ, hole-low, hole-high,id > */
22      if CacheSearchNS(Normal-cache, msg.hole-high) = null then SEND(msg, boss);
23        else resp := ∅;
24          foreach i ∈ [msg.hole-low, msg.hole-high ] do
25            resp := resp ∪ CacheSearch(Normal-buffer, i);
26          Send(< FETCHREP, msg.hole-low, msg.hole-high, resp >,msg.id);
27    ■ msg.tag = STABINFO → /* from MSS, < STABINFO, low, mid > */
28      change tag of msg to CoordStabInfo;
29      UniformMulticast(msg);
30    ■ msg.tag = COORDSTABINFO → /* from coords, structure like STABINFO */
31      m-elem := element of member-table : m-elem.mid = msg.mid;
32      m-elem.delivered := msg.low;
33      foreach m ∈ Normal-buffer {
34        if ∀ x-elem ∈ member-table, x-elem.delivered ≥ m.cseq then
35          CacheExtractNS(Normal-buffer, m) };
36    ■ msg is a view change → { old-v := v; v := msg; return; }
37 end-if
38 forever
```

Table 8: Algorithm executed by coordinators. Notice that the multicast at lines 16 and 20 are addressed to non group members, i.e., it does not require the semantics of virtual synchrony.

6 Simulation

Our simulation analysis is divided into two parts. In the first part we evaluate the system performance in normal operating conditions, i.e., in the absence of failures. The purpose of this part is to capture the inherent cost of fault-tolerance in the proposed protocol. In the second part we assume that failures may occur, in the form of crashes at gateways and coordinators, and we analyze the ability of the protocol to react to such failures.

6.1 Simulation Environment

We set the numerous parameters that characterize the protocol similar to [6], which provides a simulation analysis for the non fault-tolerant version. We consider an operating environment with 40 MSSs and 100 MHs. A MH remains in a cell for a random time interval. The length of this interval is exponentially distributed (see [3, 11]), and its average T_{cell} is set to 10 seconds for each MH. The wireless coverage is complete and the message loss rate in the wireless network is 0.1%. Message losses are assumed to be independent.

All MHs receive multicasts (N_r=100) whereas a subset of them with N_s=10 members may generate 512-byte messages. Message generation is a Poisson process, i.e., times between the generation of successive messages are random variables exponentially distributed [3, 14, 18, 21]. Each sender generates, on the average, 8 messages/sec corresponding to a bit rate of approximately 33 Kbps.

We consider a wireless bandwidth of 1 Mbps and a wired bandwidth of 10 and 100 Mbps, respectively. In particular, we assume that coordinators are interconnected by a high speed wired LAN at 100 Mbps while gateways are connected each other and to coordinators by means of a 10 Mbps wired infrastructure (a preliminary version of this work analyzed the case in which all stationary nodes were connected by a 10 Mbps wired infrastructure [5]). Propagation delays, i.e., times messages take to travel from one node to another, are as follows. Wireless propagation delays are negligible as cells are supposed to be relatively small in size (ten to hundred meters). Wired propagation delays are assumed to be exponentially distributed [18]. The average for messages from MSSs to coordinators or back is 1.5 msec, whereas for messages from coordinators to the boss is 2 msec (these point-to-point messages are sent through group communication). Uniform multicast from the boss to coordinators is modeled as an additional exponential delay with average ($N_c + 0.4$) msec, where N_c is the number of coordinators [12].

In our simulations we also consider processing time, i.e., time necessary to process a message. We estimated it as described in [6] and it ranges from a few msec to hundreds of msec, depending on the message tag (NEW, NORMAL, etc.) and on the host where the message is processed (MH, MSS, coordinator, boss).

The independent replication method is used to estimate the performance measures, and the confidence level is set to 90%.

6.2 Analysis in the absence of failures

Fault-tolerance obviously comes at a cost. A protocol designed to be fault-tolerant is likely to exhibit, *even in the absence of failures*, performance worse than that of a protocol that does not tolerate failures. In this section we evaluate such costs in order to capture the inherent cost of fault-tolerance in the proposed protocol. Accordingly, we assume reliable communication in the wired network and SHs that do not crash.

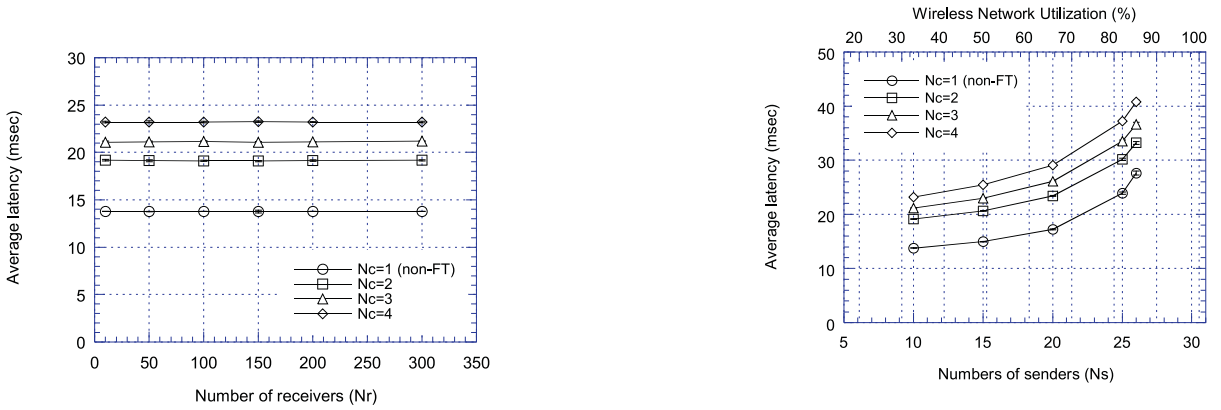

Figure 2: Average latency as a function of the number of receivers (left) and of the number of senders (right), for different numbers of coordinators.

We compare the performance of the proposed protocol with the non fault-tolerant version. This comparison is based on the *average message latency*, i.e., the average time elapsed from the instant at which a message is generated at a sending MH to the instant at which the same message is delivered by a (generic) destination MH. We analyzed latency for varying number of coordinators, sending MHs, mobility of MHs and message loss rates in the wireless network. Curves labelled as $N_c = 1$ refer to the non fault-tolerant version while the other curves relate to the fault-tolerant protocol proposed in this paper. The non fault-tolerant version is a significant baseline because, as shown in in [4], it outperforms earlier approaches (AB-protocol, see section 3)

Figure 2-left shows the average latency as a function of the number of receivers for different number of coordinators (N_c). The key observation is that the fault-tolerant version maintains the very good scalability properties of the non fault-tolerant one. Figure 2-right shows the average latency as a function of the number of senders, i.e., of the aggregate message rate. Although average latency increases with the number of senders, it is important to observe that even in this case curves for different values of N_c are approximately parallel. In other words, the fault-tolerant version maintains approximately the same scalability properties of the non fault-tolerant one even when the aggregate message rate is increased.

The fault-tolerant protocol exhibits higher average latency as a result of the following factors:

1. Each message m experiences an additional step with respect to the non fault-tolerant version: from the coordinator associated with the MH that originated m to the boss.

2. When the boss receives a NEW message it does not multicast the related NORMAL message immediately, but it sends a uniform multicast within the view and waits for delivery of this multicast.

3. NEW messages are implicitly acknowledged by the related NORMAL message from the boss, while in the non fault-tolerant version they are explicitly acknowledged by the local MSS. It follows that MHs have to use longer time-outs in order to minimize useless retransmissions, but this delays retransmission of messages that are actually lost. In our experiments we set the time-out for the fault-tolerant version to 100 msecs, and the one for the non fault-tolerant version to 10 msecs.

We show in the following that the contribution of point 3 is indeed significant and cannot be neglected. The contribution d_1 due to point 1 can be calculated as follows. Let $d_{C \to B}$ be the latency

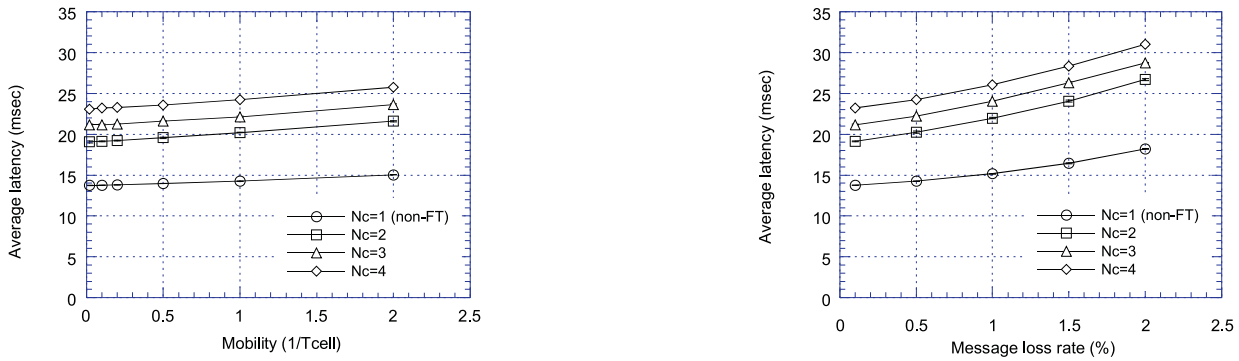

Figure 3: Average latency as a function of mobility (left) and of the wireless network unreliability (right), for different numbers of coordinators.

for messages from a coordinator C to the boss. Since messages originated by MHs associated with the boss do not incur in this latency, we have:

$$d_1 = \frac{N_c - 1}{N_c} \cdot d_{C \to B}$$

The average value of d_1 can be obtained from this formula having observed that the average value of $d_{C \to B}$ is 2.04 msec (resulting from the sum of transmission and propagation delays). The average value for the contribution d_2 due to point 2 is $(N_c + 0.4)$ msec, where N_c is the number of coordinators (see the previous section). For $N_c = 2$ the increase in latency with respect to the non fault-tolerant protocol (curve labelled N_c=1 in Figure 2-left) is approximately 5 msecs. Since $N_c = 2$ corresponds to $d_1 = 1.02$ msec and $d_2 = 2.40$ msec, it follows that the contribution of these two factors does not suffice to characterize the cost of fault-tolerance.

Figure 3 shows the influence of MH mobility (Figure 3-left) and wireless network unreliability (Figure 3-right). Mobility is expressed in terms of the number of cell switches per second experienced by each MH, which is the inverse of the T_{cell} parameter, i.e., the average time interval a MH remains inside a cell. Unreliability of wireless links is expressed as the percentage of lost messages.

Both plots exhibit a similar behavior. In particular, the difference between the non fault-tolerant protocol and the fault-tolerant protocol (for example with N_c=2) increases as mobility or message loss rate grows up. The similarity can be easily understood if one considers that movements of MHs cause message losses. The increase in the distance between curves related to $N_c = 1$ and $N_c > 1$, respectively, is a consequence of point 3 above. If the fraction of NEW messages which get lost increases, then the average delay for recovering these messages increases accordingly. Furthermore, the buffering delay at the sending MH increases as well (MHs use a simple stop and wait transmission scheme).

To summarize, we can conclude that the latency cost induced by fault-tolerance in normal operating conditions (absence of failures) is in the order of a few msecs. Components of the additional delay indicated above as d_1 and d_2 are fixed and cannot be reduced (for a fixed wired network technology and operating environment). The delay component d_3 could be lowered by using at MHs: (i) a transmission scheme more sophisticated than the simple stop and wait approach (e.g., a window-based scheme); and/or (ii) a shorter time-out for NEW messages. However, the former would induce higher computational load at MHs while the latter would cause useless retransmissions and, thus, wastage of wireless bandwidth as well as computing and energy resources at the MH. Based on the above results, we believe that these solutions would lead to minor performance

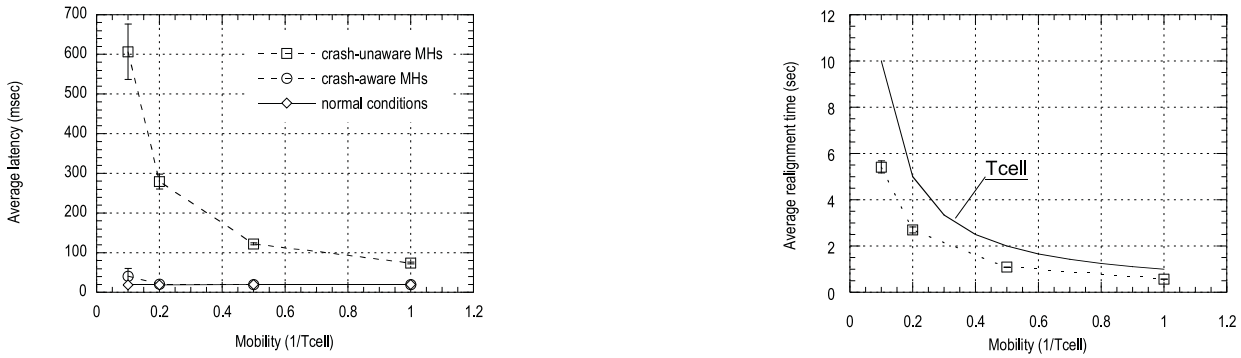

Figure 4: Average message latency (left) and average message realignment time (right) with a crashed MSS, as a function of mobility.

improvements that would not compensate for their drawbacks. However, in a different scenario, e.g., when MSSs are distributed in a geographical area rather than in a local area, use of a window-based transmission scheme could be appealing.

6.3 Analysis in the presence of failures

In this section we investigate the behavior of the protocol in the presence of failures. The possible range of failure scenarios is obviously enormous. For this reason, we focus on the failure of either one MSS or one coordinator. We consider that failures are permanent, that is, a failed node remains not available for the entire simulation experiment. We assume N_c=2 in all the experiments.

6.3.1 Crashes at MSSs

Whenever a MSS crashes the related cell becomes an uncovered area. MHs that happen to be located within that cell are unable to communicate. They will reconnect as soon as they enter the covered area again, which may occur either because they move to another cell, or because the crashed MSS recovers.

Clearly, the crash of a MSS may affect performance. First, MHs that pass through the uncovered area tend to deliver messages after a longer delay. Second, the number of lost messages tends to grow, thereby increasing the overall system load. Note, the latter issue implies that even MHs that never pass through the uncovered area could experience some performance degradation.

To evaluate the effects of a MSS crash we divided MHs into two classes: *crash-aware MHs* never enter the uncovered area; *crash-unaware MHs* may pass through the uncovered area. We measured the average message latency experienced by MHs in each class, as a function of mobility. The results are in Figure 4-left (the curve corresponding to the absence of failures is also shown, to simplify the comparison). As expected, the failure of a MSS affects also the performance perceived by crash-aware MHs. The reason for the strong effect of mobility is quite evident: If the mobility increases (i.e., T_{cell} decreases) the number of messages missed in the uncovered area decreases accordingly.

To characterize performance in more detail, we also measured the *average realignment time* for crash-aware MHs, i.e., the average time a MH takes to recover all messages missed while in the uncovered area. More precisely, this is the average time interval between: (i) the instant at which the MH leaves the uncovered area; and (ii) the instant at which the last message missed in the uncovered area is delivered to the application layer at the MH. Clearly, this is an important aspect that is not captured by the measures of average latency. The results are in Figure 4-right. The

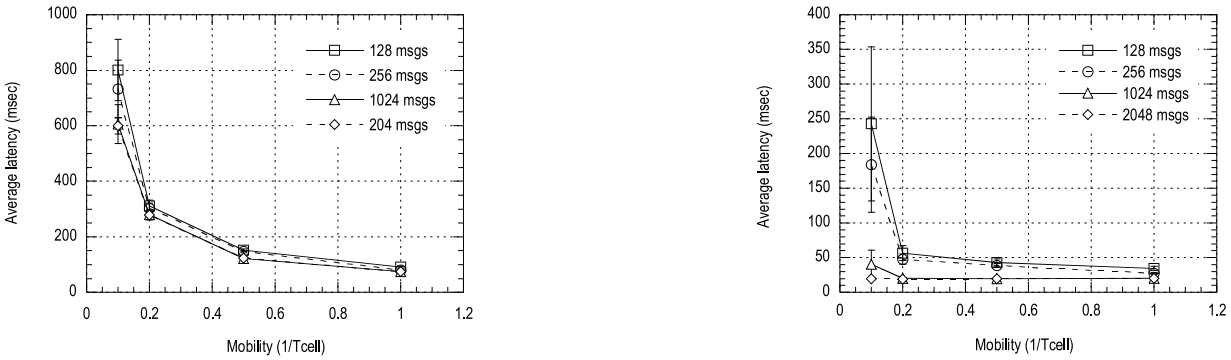

Figure 5: Average message latency with a crashed MSS as a function of mobility, for crash-unaware MHs (left) and crash-aware MHs (right). Each graph includes curves corresponding to different sizes of the cache at MSSs.

reason for the strong dependence on mobility is the same as that discussed for Figure 4-left. It can be seen that the average realignment time is always less than T_{cell}. This means that, on the average, all messages missed in the uncovered area are recovered in the next cell.

Some form of control on the performance degradation caused by MSS crashes can be exercised by proper sizing of resources in the wired network, i.e., MSS cache and number of coordinators. Intuitively, increasing the former may result in a smaller number of messages to be fetched from coordinators, thereby lowering the overall system load. Increasing the number of coordinators may result in a lower load (on the average) on each coordinator. We investigated these two options and summarize our results below.

Figures 5-left and 5-right show the average message latency as a function of mobility for crash-unaware and crash-aware MHs, respectively. Each graph includes curves for various cache sizes. It can be observed that increasing the cache size indeed results in a smaller latency and that this improvement occurs for both classes of MHs. However, increasing the cache size beyond a certain value (1024 messages in our experiments) does not produce any further improvement in latency. The reason is because the cache hit ratio is already close to 1. With respect to the realignment time, we have found that the MSS cache size is largely irrelevant. The reason is because the realignment time is almost completely determined by the wireless transmission times and queuing delays at the MSS, and these delays are not influenced by the cache size.

As for increasing the number of coordinators, we found by simulation that this strategy might be useful only when the cache size at MSSs is small (hit ratio sensibly smaller than 1). When the hit ratio is close to 1, increasing the number of coordinators has no effect on the latency and realignment time for crash-unaware MHs, but it has a negative impact on the latency for crash-aware MHs. The reason is that increasing the number of coordinators results in a higher latency for the communication between coordinators.

6.3.2 Crashes at Coordinators

In the previous section we analyzed the steady-state performance of the protocol following the crash of a MSS. The steady-state performance anaysis following the crash of a coordinator may be obtained from the analysis in the absence of failures (section 5.2): If N_c is the number of coordinators in normal operating conditions, the steady-state performance after a coordinator has crashed is that observed in the same conditions but with $N_c - 1$ coordinators.

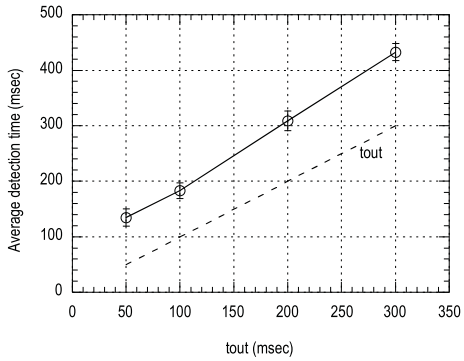

Figure 6: Average detection time as a function of the retransmission time at MSSs.

However, it is important to investigate how quickly the system reacts to the crash of a coordinator. Recall that MSSs detect failures of coordinators by means of a simple timeout mechanism. When a MSS sends a message to a coordinator C it also sets a timer. If the expected acknowledgment does not arrive within a predefined timeout t_{out}, MSS assumes that C is no longer alive and selects a new coordinator from the coordinator pool. It follows that when a coordinator crashes there will be some messages (those addressed to that coordinator) that experience an additional delay. Clearly, this is an important performance-related issue that is not captured by the steady-state analysis. For this reason, we decided to evaluate the *average detection time*, that we define as the average time interval from (i) the instant at which a coordinator crashes; and (ii) the instant at which a MSS detects the crash.

Figure 6 shows the average detection time as a function of t_{out} (it is assumed that the group communication layer detects failures of coordinators much faster than t_{out}). It can be observed that the average detection time grows linearly with t_{out}, but it is greater than t_{out} itself. To explain this behaviour, consider the two extreme cases: (i) the coordinator crashes just *before* sending the acknowledgment; (ii) the coordinator crashes immediately *after* sending the acknowledgment. In case (i) the MSS detects the crash as soon as the timer expires, i.e., after t_{out} sec. Note that this case corresponds to the faster possible detection for the MSS. In case (ii) the MSS will receive the acknowledgment, thus it will assume that the coordinator is still alive. It follows that the crash will be detected after t_{out} sec following the *next* transmission, i.e., the detection time in this case is greater than t_{out} and depends on the message inter-arrival time at the MSSs.

With respect to the effect of the crash of a coordinator on the average latency, we observe that only the *first* message that a MSS sends after the crash experiences an additional delay (due to the detection time), whereas the subsequent messages from that MSS will not experience any additional delay because they will be forwarded directly to the new coordinator. Besides, a smaller number of coordinators implies a smaller time for the synchronization steps amongst coordinators.

Finally, we observe that most of the above discussion applies unchanged to the case in which a network partition isolates a MSS from the coordinator it is associated with. Being based on time-outs, the failure detection mechanism used by MSSs is clearly unable to distinguish between a failure of the coordinator and a network failure.

7 Concluding remarks

We have presented a protocol for offering fault-tolerant support to (totally ordered) reliable multicast within a group of MHs. The protocol tolerates crashes of SHs and partitions of wired links. To our knowledge, no other protocol provides these functionalities.

A key feature of the proposed protocol is that movements of MHs do not require any interaction among SHs, which is important for performance and simplicity. Not only this feature simplifies failure handling, as there is no need to implement a fault-tolerant hand-off, it also contributes to efficiently support large populations of MHs roaming across small cells. This scenario is gaining importance due to the trend toward smaller cells motivated by their advantages in terms of improved aggregate throughput and smaller power required for transmitting [13].

MSSs do not store any critical state information, which simplifies the accommodation of their failure and recovery. They merely act as forwarding switches and as *caches* of state information whose primary copy is kept elsewhere, i.e., at coordinators. Replication through group communication is the main tool for enhancing availability of this information and for preserving its consistency in spite of failures.

Users may roam in areas not covered by the wireless network and messages could be lost even within cells. Users may enter and leave cells at any time and without prior negotiation: they are only required to not move very fast *all the time*. If users move too fast or at times that are inopportune for the protocol, then they will experience an occasional performance penalty (i.e., they will loose messages), but the protocol will still be correct. Moreover, the penalty will disappear once they stop moving too fast. Other protocols are based on more restrictive mobility assumptions, e.g., users do not move *while* hand-off is in progress [1, 3, 7, 18, 23].

As discussed in more detail in [6], size of message headers, size of data structures at MHs, number of messages in the wired network for each multicast do not depend on the number of group members. These factors contribute to make the protocol highly scalable.

We performed an extensive simulation analysis to evaluate our protocol both in normal operating conditions and in the presence of failures at MSSs and coordinators. The results obtained show that the protocol is indeed practical and exhibits very good scalability properties. In normal operating conditions the increase in the average latency experienced by messages (with respect to the non fault-tolerant version of the protocol) is limited to some milliseconds. Such increase, that represents the inevitable cost to achieve fault-tolerance, depends on the number of coordinators thereby enabling one to obtain the desired trade-off between performance and availability. We also evaluated the impact on the system performance due to crashes at coordinators and MSSs, and the responsiveness of the protocol in detecting and managing coordinator (or link) failures. Specifically, we found that the performance degradation caused by MSS failures can be significantly reduced by an appropriate sizing of MSS caches.

Acknoweledgements

The work described in this paper has been partially carried out under the financial support of the Italian Ministero dell'Istruzione, Università e Ricerca (MIUR) in the framework of the projects "Strumenti, Ambienti e Applicazioni per la Società dell'Informazione" and "Matematica per le scienze e la tecnologia".

The authors wish to express their gratitude to Francesco Donati who implemented the simulator and ran the simulation experiments.

References

[1] A. Acharya and B. R. Badrinath. A framework for delivering multicast messages in networks with mobile hosts. *ACM/Baltzer Journal of Mobile Networks and Applications*, 1(2):199–219, 1996.

[2] S. Alagar, R. Rajagoplan, and S. Venkatesan. Tolerating mobile support station failures. In *Proc. of the First Conference on Fault Tolerant Systems*, pages 225–231, Madras, India, December 1995. Also available as Technical Report of the University of Texas at Dallas.

[3] S. Alagar and S. Venkatesan. Causal ordering in distributed mobile systems. *IEEE Transactions on Computers*, 46(3):353–361, March 1997.

[4] G. Anastasi and A. Bartoli. On the structuring of reliable multicast protocols for mobile wireless computing. Technical Report DII/00-1, Università di Pisa, January 2000. Submitted for publication. Available at http://www.iet.unipi.it/∼anastasi/papers/tr00-1.pdf.

[5] G. Anastasi, A. Bartoli, and F. Luccio. Fault-tolerant support for reliable multicast in mobile wireless systems. In *Proc. IFIP TC-6 Networking Conference*, Lecture Notes in Computer Science, pages 1069–1080. Springer Verlag, May 2002.

[6] G. Anastasi, A. Bartoli, and F. Spadoni. A reliable multicast protocol for distributed mobile systems: Design and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1009–1022, October 2001.

[7] V. Aravamudhan, K. Ratnam, and S. Rangajaran. An efficient multicast protocol for PCS networks. *ACM/Baltzer Journal of Mobile Networks and Applications*, 2(4):333–344, 1997.

[8] Ö. Babaoğlu, R. Davoli, L. Giachini, and P. Sabattini. The inherent cost of strong-partial view-synchronous communication. In *Distributed Algorithms*, Lecture Notes in Computer Science, pages 72–86. Springer Verlag, 1995.

[9] A. Bartoli. Group-based multicast and dynamic membership in wireless networks with incomplete spatial coverage. *ACM/Baltzer Journal on Mobile Networks and Applications*, 3(2):175–188, 1998.

[10] Ken Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[11] K. Brown and S. Singh. Relm: Reliable multicast for mobile networks. *Computer Communications*, 21(16):1379–1400, 1998.

[12] R. K. Budhia. Performance engineering of group communication protocols. Technical report, University of California, S. Barbara (USA), August 1997. Ph.D. dissertation.

[13] R. Caceres and V. Padmanabhan. Fast and scalable wireless handoffs in support of mobile internet audio. *ACM/Baltzer Journal of Mobile Networks and Applications*, 3(4):351–363, 1998.

[14] V. Chikarmane, C. Williamson, R. Bunt, and W. Mackrell. Multicast support for mobile hosts using mobile IP: Design issues and proposed architecture. *ACM/Baltzer Journal of Mobile Networks and Applications*, 3(4):365–379, 1998.

[15] Kenjiro Cho and Kenneth P. Birman. A group communication approach to mobile computing. Technical Report CS TR94-1424, Cornell University, May 1994.

[16] F. Kaashoek and A. Tanenbaum. An evaluation of the amoeba group communication system. In *Proc. 16-th IEEE International Conference of Distributed Computing Systems*, pages 436–447, May 1996.

[17] D. Pradhan, P. Krishna, and N. Vaidya. Recovery in mobile environments: Design and trade-off analysis. In *26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, June 1996. Also available as Technical Report of the Texas A&M University.

[18] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. *Journal of Parallel and Distributed Computing*, pages 190–204, March 1997.

[19] R. Prakash and M. Singhal. Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1035–1048, 1996.

[20] R. Vitenberg, G. Chockler, I. Keidar, and D.Dolev. Group communication specifications: A comprehensive study. Technical Report 790, MIT Laboratory for Computer Science, 1999.

[21] C. Williamson, T. Harrison, W. Mackrell, and R.Bunt. Performance evaluation of the mom mobile multicast protocol. *ACM/Baltzer Journal of Mobile Networks and Applications*, 3:189–201, 1998.

[22] G. Xylomenos and G. Polyzos. IP multicast for mobile hosts. *IEEE Personal Communications*, pages 54–58, January 1997.

[23] L. Yen, T. Huang, and S. Hwang. A protocol for causally ordered message delivery in mobile computing systems. *ACM/Baltzer Journal of Mobile Networks and Applications*, 2(4):365–372, 1997.