Fault-Tolerant Support for Totally Ordered Multicast in Mobile Wireless Systems

Flaminia L. Luccio[1] *, Alberto Bartoli[2], and Giuseppe Anastasi[3]

[1] Dip. di Scienze Matematiche
Università di Trieste, Via Valerio 12, 34100 Trieste (Italy)
luccio@mathsun1.univ.trieste.it
[2] Dip. di Elettrotecnica, Elettronica e Informatica
Università di Trieste, Via Valerio 10, 34100 Trieste (Italy)
bartolia@univ.trieste.it
[3] Dip. di Ingegneria dell'Informazione
Università di Pisa, Via Diotisalvi 2, 56125 Pisa (Italy)
anastasi@iet.unipi.it

Abstract. Emerging architectures based on *portable computers* and *wireless networking* allow users equipped with hand-held computing devices to roam around freely while maintaining connectivity by means of wireless communication technology. In this paper we present a protocol for totally ordered multicast within a group of mobile hosts that communicate with a wired infrastructure by means of wireless technology. The protocol tolerates failures in the wired infrastructure, i.e., crashes of stationary computers and partitions of wired links. The wireless coverage may be incomplete and message losses could occur even within cells, due to physical obstructions or the high error rate of the wireless technology, for example. Movements of mobile hosts are accommodated efficiently because they do not trigger any interaction among stationary computers (i.e., there is no notion of hand-off).

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.

In this paper we present a protocol for totally ordered multicast within a group of mobile hosts that communicate with a wired infrastructure by means of wireless technology. The protocol is resilient to (a certain number of) crashes of stationary computers and partitions of wired links. Totally ordered multicast is a powerful building block for enabling a set of remote processes to maintain a replicated state. Our work makes this building block available on mobile wireless systems. 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.

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). Mobile hosts may move and communicate through wireless links. Hosts do not share memory and the system is asynchronous. 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.

Our failure assumptions are as follows: (i) Wireless communication within a cell is FIFO-ordered but messages may be lost; (ii) MHs may roam unexpectedly in areas that are not covered by any

* This work has been supported in part by M.U.R.S.T., progetto 40 % "Metodi e Problemi in Analisi Reale".

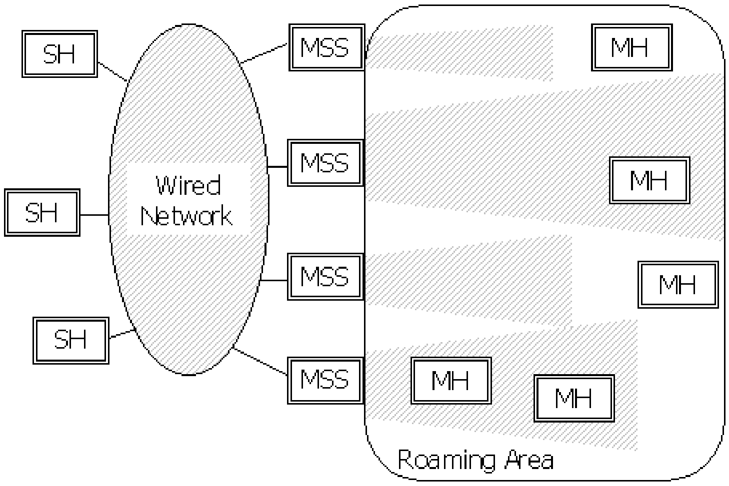

Fig. 1. Example system with five MHs and seven SHs. Four SHs are MSS. Wireless cells are indicated as dashed areas within the roaming area. Notice that portions of the roaming area are out of coverage.

cell; (iii) SHs may crash and wired links may partition; (iv) Every crash eventually recovers and every partition eventually heals; (v) MHs do not crash. Our work on mobile wireless systems is based on a design philosophy aimed to improve reliability of final applications:

1. 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.
2. Availability of MSSs should affect only availability of applications, not their correctness. In other words, a MSS failure should merely shrink the covered area, without affecting correctness.
3. One should avoid to make hypothesis on users' movements. Otherwise, even a *single* inopportune movement could compromise correctness.
4. One should avoid to assume that wireless coverage is complete. Otherwise, even a single MSS malfunctioning, or physical obstruction, or particularly unfortunate area, 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.

We used the above rules for designing a protocol like the one presented here, but not resilient to failures [Bar98]. This protocol, hereinafter *non-FT protocol*, provided totally ordered multicast delivery in the same architecture and system model, except that failure assumption (iii) was replaced by: (iii') SHs never crash and messages on the wired links are never lost. We believe that the significance of the above rules is corroborated by the following facts: (i) our non-FT protocol performs significantly better than other multicast protocols not based upon such rules (see next section); and (ii) the fault-tolerant extension proposed here is obtained from the non-FT protocol simply, with small and localized changes.

2 Related work

Forms of fault-tolerant support for mobile hosts were proposed in [CB94,PKV96,PS96] but they all consider different kinds of support. Support for a FIFO channel from a MH to a (fault-tolerant) SH is considered in [CB94]. Checkpointing and recovery from node failures is considered in [PS96], in the hypothesis of reliable communication and fail-stop node failures (i.e., failures are reliably detected). A similar problem is considered in [PKV96], but without addressing failures of SHs.

The only work with scope similar to ours that we are aware of is [ARV95], that introduces resiliency to failures of MSSs in a a non fault-tolerant protocol proposed by Acharya and Badrinath (see below, [AB96]). The system model is more restrictive than ours as it assumes fail-stop failures and reliable communication. The extension to [AB96]is obtained by associating each MH with a set of MSSs, denoted $\mathcal{S}$(MH), and by replicating state information about MHs at each MSS in $\mathcal{S}$(MH). Members of $\mathcal{S}$(MH) have to execute a replica control protocol whenever MH sends a message or there is a message addressed to it. This protocol must be able to tolerate node failures and two alternatives are given. The one that is more efficient in the absence of failures requires additional mechanisms (network flush or rollback) that are not detailed. The interaction among the replica control protocol, MSS recovery, hand-off (see below) are only outlined.

The protocol by Acharya and Badrinath (AB-protocol for short) was the first one to support reliable (FIFO) multicast for mobile hosts and has been highly influential [ARR97,AV97,PRS97,YHH97]. 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 of a proper procedure, called *hand-off*. Therefore: (i) The state shared among SHs is updated upon *each* movement; (ii) MSSs maintain critical state information; and (iii) the crash of a MSS affects correctness of the application. Amongst other things, these features explain why the fault-tolerant extension in [ARV95] requires a complex interaction among several sub-protocols. We observe also what follows:

- The mobility assumptions of the AB-protocol refer to each *single* move: 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 mobile host stops moving "too fast".
- The AB-protocol assumes that wireless communication is reliable. Unlike our protocol, it does not consider physical obstructions, uncovered regions, message losses due to the high error rate typical of wireless technology.

While our design philosophy is a significant departure from the AB-protocol, it does not appear to be detrimental to performance. We compared through simulation the AB-protocol to our non-FT protocol, in terms of latency, scalability, bandwidth usage efficiency and quickness in managing cell switches of users and found that our protocol performs significantly better [AB00].

3 Overview of the protocol

We begin by briefly outlining our *non-FT 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 the 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 (i.e., in total order) and to detect missing messages. In the latter case, the MH sends a retransmission request specifying an interval of missing sequence numbers to the local MSS. When a MSS receives one such request, tagged Nack, the MSS relays the missing messages to the sending MH. The MSS obtains missing messages from a local *cache* or, in case of a miss, from the coordinator. A request to the coordinator is tagged FETCHREQ and specifies an interval of missing 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. Notice that MSSs do not store critical state information: such information is kept by the coordinator and merely cached by MSSs for efficiency.

The *fault-tolerant* extension is obtained fairly simply, as follows. 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 does not respond to the sending MH with an acknowledgment, see also section 4.2). Moreover, the role of the coordinator is played by *a set* of SHs implementing a "coordinator service" that is more available than any of its composing SHs (and connecting links). Interaction among members of the coordinator service, i.e., among *coordinators*, is implemented through *group communication (GC)*. A detailed description of GC is beyond the scope of this paper (see [Bir93,VCKD99] for example). Below we shall provide only some background.

Coordinators execute on top of a software layer that implements GC and belong to a *group* (this notion of group has nothing to do with the group of MHs). The GC layer provides each coordinator with consistent group membership information in the form of a *view* of the group, i.e., a set of coordinators (and a view identifier). Messages sent by a coordinator in one view are delivered only to coordinators in the membership of that view, and only when they have the same view. The GC layer tracks membership changes and reports them to coordinators in the form of *view change* events, i.e., special messages carrying the new view. The view that is *current* at a coordinator is the one specified by the last view change received by that coordinator. A view has been *installed* by a coordinator only if the coordinator indeed received the corresponding view change.

We shall assume that the GC layer supports *partitionable* membership, i.e., it allows multiple views of the group to exist concurrently, to model network partitions. We shall also assume that the GC layer supports *uniform multicast*, i.e., a multicast such that if any member of view v delivers multicast m, then each member of v delivers m or crashes[1]. We present the algorithm in the hypothesis that a view including a majority of coordinators always exists. The algorithm could be extended to accommodate the more general case in which the majority view temporarily disappears, but this topic is largely independent of the main scope of this paper.

In our algorithm, coordinators whose current view is a majority may generate NORMAL messages and respond to FETCHREQs from MSSs. Coordinators whose current view is not a majority wait. A MSS sends its messages to one coordinator C, e.g., the nearest one. The MSS could not receive a response for several reasons, including: (i) C receives the message when its current view is not a majority; (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 (of course, timeouts expiring too soon do not affect correctness), the MSS will send the next request to another coordinator. The policy used by a MSS for selecting this coordinator is irrelevant to this paper. The request not yet answered will be retransmitted by a MH (section 4.2).

4 The algorithm

4.1 Notation and beaconing

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.

[1] Supporting uniform multicast in a partitionable system may require a GC layer that may deliver either "regular" views or "transitional" views [VCKD99]. For our algorithm, one should just ignore views of the latter kind.

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

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

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.

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 [Bar98] 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.

4.2 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 6-10). 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) and delivers all buffered messages that can now be delivered, if any (27-32). If the received message was originated by MH itself, then timer-new is cleared (25-26; when timer-new expires MH resends the last NEW message, lines 14-16) If there is a buffered message that cannot be delivered, then MH requests a retransmission by sending a suitable NACK message (35-37). Upon sending this message, 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).

4.3 Mobile Support Station side

Each MSS maintains the variables listed in table 3. C-team is a set that identifies the coordinators. This set is defined statically and never changes. L-cell is a set that identifies the MHs currently covered by the MSS (this set is updated by the beaconing protocol). C is the member of C-team with which MSS communicates. 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; 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); curr, that is the sequence number of the last message sent to mid. All variables are initialized by the procedure MSS-Init() (table 3).

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 15-24). 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) 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 44). StateTransfer() terminates when either all messages have been sent, in which case the related miss-table entry is cleared (46), or when a FETCHREQ has been sent. The response to a FETCHREQ is a FETCHREP message (24), 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).

```
Main Program
MH-Init();
repeat
    msg := RECEIVE();
    multi-if
        ▮ msg.tag= MULTICAST → /* from the application, < MULTICAST, M > */
            new-num := new-num +1;
            new-last :=< NEW,msg.M,new-num,mid >;
            SEND(new-last,G);
            TimerSetNS(timer-new);
        ▮ msg.tag= NORMAL → /* from G, < NORMAL,M,cseq,mid > */
            if (msg.cseq > norm-seq) and (msg ∉ MH-buffer) then
                HandleNormal(msg);
        ▮ msg is a timeout from timer-new →
            SEND(new-last,G);
            TimerSetNS(timer-new);
        ▮ msg is a time-out from timer-nack →
            SEND(<NACK,norm-seq + 1,mylow - 1,mid>,G);
            TimerSetNS(timer-nack);
    end-if
forever;

Procedure HandleNormal(msg);
CachePutNS(MH-buffer, msg);
UpdateMylowNS();
if msg.mid = mid then
    TimerResetNS(timer-new);
while ∃ m ∈ MH-buffer: m.cseq=norm-seq +1 do
    m := CachePurgeNS(MH-buffer, norm-seq+1);
    UpdateMylowNS();
    DELIVER-TO-APPLICATION(m);
    norm-seq := norm-seq +1;
end-do;
if MH-buffer = ∅ then
    TimerResetNS(timer-nack);
elseif not TimerIsSetNS(timer-nack) then
    SEND(<NACK,norm-seq + 1,mylow - 1,mid>,G);
    TimerSetNS(timer-nack);
```

Table 2. Algorithm executed by MH. UpdateMylowNS() sets mylow to the smallest cseq field (cseq is a system-wide sequence number assigned by the coordinators) of messages in MH-buffer, or to 0 if MH-buffer = ∅.

my-id: process-id;	**Procedure** MSS-Init()
C-team, l-cell: **set of** process-ids;	my-id := unique identifier;
C: process-id;	C-team := statically defined set;
timer-c: timer;	l-cell := $\emptyset$;
Normal-cache: **set of** messages;	C := pick up a member of C-team;
miss-table: **set of record**	TimerResetNS(timer-c);
mid: process-id;	Normal-cache, miss-table := $\emptyset$;
curr: integer;	
low, high: integer;	
hole-low, hole-high: integer;	

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

4.4 Coordinator side

Each coordinator maintains the variables listed in table 5. MSS-team identifies the MSSs. This set 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 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 5), joins the group and then executes Coord-Control() that never returns (table 6). This procedure waits for delivery of a majority view. Then, SynchronizeNS() brings the state of the coordinator up-to-date (see also below) and finally Coord-Service() implements the actual processing of messages from MSSs[2]. CoordService() returns upon a view change. 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 12). Then NEW messages are forwarded to the boss that multicasts them within the majority view (14; 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 (17-29). Duplicate NEWs are ignored because of the check on the new-num field (line 19). 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 26-29), because the sending MH might have missed $m1$ (a MH keeps on sending a NEW until receiving the matching NORMAL).

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

```
Main Program
MSS-Init();
repeat
    foreach x ∈ miss-table do if x.mid ∉ l-cell then miss-table := miss-table - x;
    msg := RECEIVE();
        multi-if
            ■ msg.tag = NEW → /* from MH, < NEW, M, new-num, mid > */
                SEND(msg,C); TimerSetNS(timer-c);
            ■ msg.tag = ACK → /* from coords, < ACK > */
                TimerResetNS(timer-c);
            ■ msg.tag = NORMAL → /* from coords, < NORMAL, M, cseq, mid > */
                MULTICAST(msg) in cell;
                CachePutTryNS(Normal-cache,msg);
                TimerResetNS(timer-c);
            ■ msg.tag = NACK → /* from MH, < NACK, low, high, mid > */
                x := element of miss-table : x.mid=msg.mid;
                if x = null then
                    allocate entry x ∈ miss-table;
                    x.mid := msg.mid; x.low := 0;
                if msg.low ≥ x.low then
                    x.low, x.curr := msg.low; x.high := msg.high;
                    SEND(< STABINFO,msg.low,msg.mid >, C);
                    StateTransfer(x);
            ■ msg.tag = FETCHREP → /* from coords, < FETCHREP, l, h, miss-req > */
                TimerResetNS(timer-c);
                foreach m ∈ msg.miss-req do CachePutNS(Normal-cache,m);
                foreach x ∈ miss-table do
                    if x.curr ∈ [msg.l, msg.h] then StateTransfer(x);
            ■ msg.tag is a timeout from timer-c →
                C := pick up another member of C-team;
        end-if
forever

Procedure StateTransfer(x)
msg := CacheSearchNS(Normal-cache, x.curr);
while msg ≠ null and (x.curr < x.high + 1) do
    SEND(msg, x.mid);
    x.curr := x.curr + 1;
    msg := CacheSearchNS(Normal-cache, x.curr);
end-do;
if x.curr < x.high + 1 then
    x.hole-low := x.curr; x.hole-high := x.high;
    k := minimum sequence number in Normal-cache : k > x.hole-low;
    if k ≠ null and k < x.high then x.hole-high := k − 1;
    SEND(< FETCHREQ,x.hole-low,x.hole-high,my-id >, C);
    TimerSetNS(timer-c);
else miss-table := miss-table - x;
```

Table 4. Algorithm executed by MSS.

MSS-team: **set of** process-ids;	**Procedure** Coord-Init()
my-id : process-id;	MSS-team := statically defined set;
threshold : integer;	my-id := some unique identifier;
old-v, v : **set of** process-ids;	threshold := predefined number;
coord-state : **record**	old-v, v := null;
boss: process-id;	boss := null;
cseq : integer;	cseq := 0;
Normal-buffer: **set of** messages;	Normal-buffer := $\emptyset$;
member-table: **set of record**	member-table := statically defined table;
mid: process-id;	**foreach** x $\in$ member-table **do**
new-num: integer;	x.new-num := 0;
cseq-mid: integer;	x.delivered := 0;
delivered: integer;	**end-do**;

Table 5. Variables maintained at each coordinator and the initialization procedure Coord-Init().

FETCHREQ messages are processed locally and the matching FETCHREP is constructed by using the Normal-buffer (30-33). 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, the FETCHREQ is forwarded to the boss (32).

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 6). 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 [BDGS95]. 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.

5 Concluding remarks

We have presented a protocol for offering fault-tolerant support to totally ordered multicast within a group of mobile hosts. The protocol tolerates crashes of stationary hosts and partitions of wired links, as well as unreliable wireless communication and incomplete coverage. A key feature of our protocol is that movements of mobile hosts do not require any interaction among stationary hosts,

```
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

9 Procedure CoordService()
10 repeat
11   msg := RECEIVE();
12   if msg.tag = New or FetchReq and sender is MSS then SEND(< Ack >, MSS);
13   multi-if
14        ▮ msg.tag = New → /* from MH through MSS or coords, < New, M, new-num, mid > */
15             if my-id ≠ boss then SEND(msg,boss);
16             else { change tag of msg to CoordNew; UniformMulticast(msg); }
17        ▮ msg.tag = CoordNew → /* from CB, structure like New */
18             m-elem := element of member-table : m-elem.mid = msg.mid;
19             if msg.new-num = m-elem.new.num + 1 then
20                  cseq := cseq + 1;
21                  n-msg := < Normal, msg.M, cseq, msg.mid >;
22                  CachePutNS(Normal-buffer, n-msg);
23                  m-elem.new-num := m-elem.new-num + 1;
24                  m-elem.cseq-mid := cseq;
25                  if my-id = boss then Multicast(n-msg, MSS-team);
26             elseif m-elem.new-num = msg.new-num then
27                  if my-id = boss then
28                       n-msg := CacheSearchNS(NormalBuffer, m-elem.cseq-mid);
29                       Multicast(n-msg, MSS-team);
30        ▮ msg.tag = FetchReq → /* from MSS or coords, < FetchReq, hole-low, hole-high,id > */
31             if CacheSearchNS(Normal-cache, msg.hole-high) = null then
32                  SEND(msg, boss);
33             else resp := ∅;
34                  foreach i ∈ [msg.hole-low, msg.hole-high ] do
35                       resp := resp ∪ CacheSearch(Normal-buffer, i);
36                  Send(< FetchRep, msg.hole-low, msg.hole-high, resp >,msg.id);
37        ▮ msg.tag = StabInfo → /* from MSS, < StabInfo, low, mid > */
38             change tag of msg to CoordStabInfo;
39             UniformMulticast(msg);
40        ▮ msg.tag = CoordStabInfo → /* from coords, structure like StabInfo */
41             m-elem := element of member-table : m-elem.mid = msg.mid;
42             m-elem.delivered := msg.low;
43             foreach m ∈ Normal-buffer {
44                  if ∀ x-elem ∈ member-table, x-elem.delivered ≥ m.cseq then
45                       CacheExtractNS(Normal-buffer, m) };
46        ▮ msg is a view change →
47                       { old-v := v; v := msg; return; }
48   end-if
49 forever
```

Table 6. Algorithm executed by coordinators. Notice that the multicast at lines 25 and 29 are addressed to non group members, i.e., it does not require the semantics of virtual synchrony.

which is important for performance and simplicity. Moreover, MSSs do not store any critical state information, which make it simple to accommodate their failure and recovery. Our protocol is based on reasonable mobility assumptions: (1) a MH does not move too fast all the time (e.g., if it always enters and leaves cells too quickly, then it could always miss a given NORMAL message); (2) a MH does not always remain out-of-coverage or in cells of crashed MSSs. A detailed correctness argument for the non-FT algorithm can be found in [Bar98]. Such argument may be extended to this work by observing that: (I) the coordinator service implemented with group communication appears to MSSs and MHs as a single, fault-tolerant coordinator; and (II) message loss in the wired network is recovered by means of suitable retransmissions.

References

[AB96] 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.

[AB00] G. Anastasi and A. Bartoli. On the structuring of reliable multicast protocols for mobile wireless computing. Technical Report DII/00-1, January 2000. Submitted for publication. Available at http://www.iet.unipi.it/∼anastasi/papers/tr00-1.pdf.

[ARR97] 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.

[ARV95] 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.

[AV97] S. Alagar and S. Venkatesan. Causal ordering in distributed mobile systems. *IEEE Transactions on Computers*, 46(3):353–361, March 1997.

[Bar98] 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.

[BDGS95] 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.

[Bir93] Ken Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[CB94] Kenjiro Cho and Kenneth P. Birman. A group communication approach to mobile computing. Technical Report CS TR94-1424, Cornell University, May 1994.

[PKV96] 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.

[PRS97] 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.

[PS96] 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.

[VCKD99] R. Vitenberg, G. Chockler, I. Keidar, and D.Dolev. Group communication specifications: A comprehensive study. Technical Report 790, MIT Laboratory for Computer Science, 1999.

[YHH97] 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.