Group-based multicast and dynamic membership in wireless networks with incomplete spatial coverage

Alberto Bartoli

Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Via Diotisalvi 2, 56126 Pisa, Italy

In this paper we examine the problem of group-based multicast communication in the context of mobile computing with wireless communication technology. We propose a protocol in which group members may be mobile computers and such that the group membership may change dynamically. Multicasts are delivered in the same order at all group members (totally-ordered multicast). Mobile computers are resource-poor devices that communicate with a wired network through a number of spatially limited cells defining wireless links. The spatial coverage provided by wireless links may be either complete or incomplete, which makes the overall system model both general and realistic. The proposed protocol is simple and does not require any hand-off in the wired network upon movements of group members. Moreover, there is no part of the protocol requiring that group members do not move during its execution. This feature leads to mobility assumptions that are practical because they involve only the global movement of group members, e.g., assumptions of the form "a group member does not move very fast all the time".

1. Introduction

Group-based multicast communication has proven to be a useful paradigm for the development of distributed programs, in particular, when the application requirements result in multiple processes that must closely cooperate together [3,5]. This programming paradigm is attractive also in the context of mobile computing with wireless communication technology. Computers may be carried by users to where their presence is actually needed, which may greatly extend the scope of such applications as inventory control, factory automation, on-site data collection, traffic monitoring, collaborative work, multiperson interaction, stock market exchange and alike. However, a number of fundamental problems not present in "stationary" group-based technology must be faced. Even in the absence of failures of either communication links or computers, group members could miss multicasts because of their movements [1]. Technological limitations in the lifetime of batteries make power consumption a major issue. The energy necessary for sending a message across a wireless link may be much greater than the energy for receiving a message, e.g., one order of magnitude [12]. Tracking the location of each group member, in order to route messages directed to it, may be costly [1]. Users could pass through areas where wireless communication is not possible, either because of physical obstructions or because the area is not covered by the wireless network. Mobile computers could be resource-poor devices (such as the so-called Personal Digital Assistants) that have, in particular, limited storage space available.

In this paper we propose a multicast protocol in which group members may be *mobile* computers and such that multicasts are delivered in the same order at all members (*totally-ordered* multicast) [6,14,16]. An important feature of the protocol is its support for *dynamic membership*, which allows mobile computers to dynamically join and leave the group. Mobile computers communicate with a wired network through a number of spatially limited *cells* that define wireless links. A major design assumption is the *incomplete spatial coverage* of wireless links, that is, the union of all cells may *not* cover the entire area where mobile computers may be located. In other words, a mobile computer may smoothly migrate across adjacent cells, but it may also disappear for an arbitrary (bounded) amount of time and suddenly reconnect at any cell. Moreover, we do not assume that communication within a cell is reliable, which models the possibility of small physical obstructions within a cell. This scenario is quite general because it accommodates contemporary wireless LANs, disconnected modes of operation, long-range movements, and the emerging *picocellular* wireless networks [13]. These are wireless networks in which the cell size is of the order of ten meters, such as a room in a building. In particular, the assumption that spatial coverage is incomplete appears to be a practical way to model scenarios in which there are indeed parts of the building where communication is not possible. The protocol works also with complete spatial coverage, however. The assumption of incomplete spatial coverage simplifies also the handling of mobile computers that are temporarily switched off in order to save batteries. For instance, if communication with a given group member appears to be impossible, it can be assumed that the group member happens to be in an uncovered area although it might actually be switched off.

In short, the proposed protocol tackles a number of combined problems: mobility of group members, dynamic membership, incomplete spatial coverage of the wireless links, total order of multicast deliveries, high cost of wireless links and limited resources of group members. It appears reasonable to claim that, in spite of all these problems, the resulting protocol is simple. Moreover, it does not "constrain" movements of group members in the sense

that group members may move at any time, irrespective of which part of the protocol is being executed. The correctness of the protocol depends only on *global* assumptions, e.g., of the form "a group member does not move very fast all the time". These assumptions will be stated precisely along the paper.

1.1. Related work

Transport-level mechanisms able to maintain "connections" to mobile computers [10,15] could be a simple way for supporting multicasting to mobile recipients, e.g., through multiple "unicasts". However, their use is not appropriate for the scenario considered in this paper:

(i) Maintaining a separate connection for each group member is costly and increases unnecessarily the load on each wireless link; for instance, this strategy would not exploit the broadcast capabilities of the wireless medium when many group members belong to the same cell [1].

(ii) Mobile computers could remain outside of all cells for a time that is "long" if compared to the timeouts used at the transport level. Similar arguments apply to services for routing datagrams to mobile computers [8], with the additional consideration that these services would leave up to the protocol the job of managing lost messages.

We employed a "two-tier" architecture [1,4] and shifted most of the communication, computation and storage cost on the computers connected to the wired network. However, we retained on mobile computers the responsibility for detecting duplicates, for detecting missing multicasts and for signaling that a cell contains group members. In the scenario considered in this paper, shifting also these aspects to the wired network would let the protocol become extremely complicated. A decision similar to ours has been (implicitly) taken in [9].

The protocol proposed in [1] (hereinafter the AB-protocol, after the names of its authors) supports multicast communication within a *static* group of mobile recipients. Wireless links provide *complete* spatial coverage and multicasts are not totally-ordered. The AB-protocol maintains on the wired network chains of pointers along which messages have to be forwarded to the current location of each group member. When a group member switches between cells, a fairly complicated *hand-off* procedure updates these pointers and moves to the new location on the wired network the messages directed to the group member and not delivered yet. It is assumed that a mobile computer does not switch between cells while hand-off is in progress. Although this hypothesis could not be difficult to satisfy in practice, we believe that the correctness of a protocol should not depend on the fact that users do not move while a certain part of the protocol is in progress. In our case users may move at any time, irrespective of what is going on at the protocol level.

Our protocol does not maintain any chain of forwarding pointers in the wired network and, consequently, it does not employ any notion of hand-off. As pointed out in [13], hand-off does not appear to be suitable for picocellular networks, because of the large amount of information that need to be forwarded. On the other hand, the hand-off employed in [1,9] guarantees that a group member never receives duplicates of multicasts already delivered. In our protocol, instead, a mobile computer may occasionally receive such unnecessary messages, that will be discarded. This aspect will be discussed in more detail in section 3.4.

The AB-protocol decreases the number of messages on the wired network by involving only those computers in the wired network whose cells actually contain group members. The set of these computers is described in a data structure, called *location view*, that is replicated at all computers in the set. Updates to the location view are serialized by a coordinator and they involve updating each replica. Hand-off must be enriched by the actions necessary for location view management, in particular, when a group member moves to an empty cell or when the last group member leaves a cell. An optimization of this procedure is employed in [9], where, essentially, neighboring cells are collectively managed as a single logical cell, so that the number of replicas of the location view is smaller and updates become less frequent. Our protocol exploits a similar idea in the sense that it also maintains the knowledge of which cells contain at least one group member. However, this information is centralized, thus its updating is potentially faster, simpler and requires fewer messages on the wired network. There may be transitory periods in which this information is not completely accurate, but these periods do not affect the correctness of the protocol and their length is of the order of the time for delivering a message on the wired network.

Totally-ordered multicasts in groups with mobile members in considered in [18]. This protocol addresses failures of stationary computers and allows mobile computers to communicate in a variety of ways, including through TCP and through e-mail. However, this protocol appears to be tailored to scenarios in which a mobile computer is either "firmly" connected to the wired network or it is disconnected. Our protocol allows mobile computers to continuously move, "appear" and "disappear" and gives special emphasis on the limited resources of mobile computers and the high cost of wireless communication.

The proposed protocol borrows several ideas from the techniques used in "stationary" distributed computing for implementing *reliable* multicasts. These are multicasts that provide strong delivery guarantees even in spite of failures of computers and/or communication links [14]. In particular, the similarities include: the use of a centralized sequencer for totally ordering multicasts and for storing multicasts that have not been acknowledged yet [16]; the total ordering of membership changes with respect to the multicasts generated by applications [5]; the independence between the propagation of acknowledgments and of multicasts generated by applications [6,7]. Essentially,

both reliable multicast in stationary distributed systems and multicast in mobile computing suffer of the same crucial problem: group members could fail to receive a multicast. Although in reliable multicast this problem is caused by failures, it turns out that the related techniques are useful also when the problem is caused by movements of group members.

2. Overview

2.1. System model

The system is composed of two kinds of computers: *stationary* computers and *mobile* computers. Communication is through message-passing. Stationary computers are connected to a wired network that provides reliable and FIFO-ordered communication. Mobile computers may move and communicate through wireless links. Some stationary computers, called *gateways*, may communicate also through wireless links. Each gateway defines a spatially limited area that is called *cell* and is covered by a wireless link. A gateway may broadcast messages to all mobile computers in its cell and send messages to a specific mobile computer in its cell, whereas a mobile computer may only send messages to the gateway of the cell where it happens to be located. Communication within a cell is FIFO-ordered but unreliable, e.g., messages may be lost. Furthermore, the union of all cells does not cover the entire area where mobile computers may be located, that is, mobile computers may roam in areas where communication is not possible. Figure 1 shows a system with 5 stationary computers and 4 mobile computers. Two stationary computers are gateways and one of the mobile computers happens to be located outside of both cells. We emphasize that we do not assume any network support for routing messages to a specific mobile computer. Finally, we assume that computers do not fail. Mobile computers may be switched off for saving batteries but only if they are not handling protocol messages, e.g., not at arbitrary times (section 4.2).

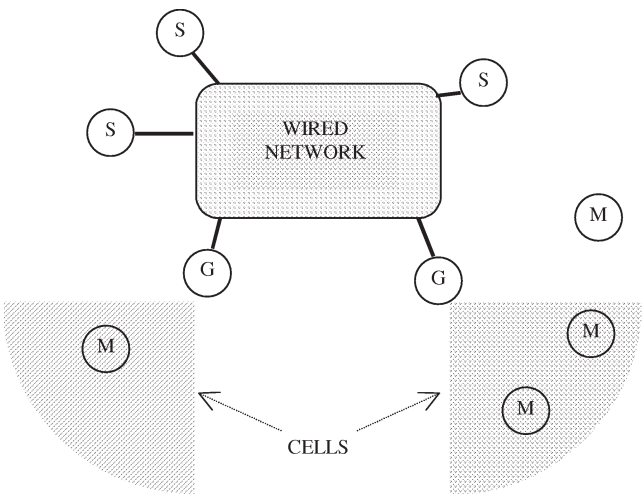

Figure 1. Example of system.

The proposed protocol employs *FIFO-multicast* among stationary computers [14], that is, a multicast that guarantees the following property: let C1 be a computer that sends a message *m1* and then a message *m2*, both through a FIFO-multicast; any computer that delivers both *m1* and *m2* delivers *m1* before than *m2*. The implementation of FIFO-multicast among stationary computers is independent of the topic of this paper. We will assume that such a functionality is exported by the layer below the protocol and we will not enter into details about how to implement it. We only observe that, in our system model, FIFO-multicast among stationary computers could be implemented simply by means of a separate unicast per recipient. The number of messages actually required may decrease if the network provides dedicated support [11].

2.2. Basic structure of the protocol

The protocol supports multicast communication within a group with dynamic membership. We will make a few assumptions that have the only purpose of simplifying the presentation and that can be easily removed:

(i) We refer to a single group.

(ii) All group members run on mobile computers.

(iii) There is at most one group member at each computer.

The protocol guarantees the following *Total Order* property [14]: If any two group members M1 and M2 deliver multicasts *m1* and *m2*, then they deliver these multicasts in the same order, e.g., either they both deliver *m1* before *m2*, or they both deliver *m2* before *m1*. The fact that M1 and M2 indeed deliver *m1* and *m2* is guaranteed under reasonable assumptions on their physical movements, that are detailed in section 3.4. Moreover, each multicast is delivered exactly once, e.g., duplicates are discarded.

In the description of the protocol we will assume that the first field of each message is a value of an enumerated type called *tag*. Tags are indicated in SMALLCAPS and are used for determining the actions to be executed upon receiving the corresponding message. We shall associate a different meaning with the terms “receiving” and “delivering” a message, in particular, at group members. A computer C *receives* a message*m* when *m* arrives at the protocol at C. Upon receiving *m*, the protocol may perform any of: (i) buffer *m*; (ii) discard *m*; (iii) pass *m* up to the application. In case (iii), C *delivers* the message *m*. The basic structure of the protocol is as follows. A computer wishing to join the group sends a message tagged JOIN to a dedicated stationary computer, called the *coordinator* and denoted as S_C. The problem of determining the identity of S_C for a given group is irrelevant to our discussion, so we will not mention this issue any further. A group member wishing to leave the group sends a LEAVE message to S_C. Group members may generate multicasts that will be delivered by all group members according to the Total Order property. Messages to be multicast in the group are tagged

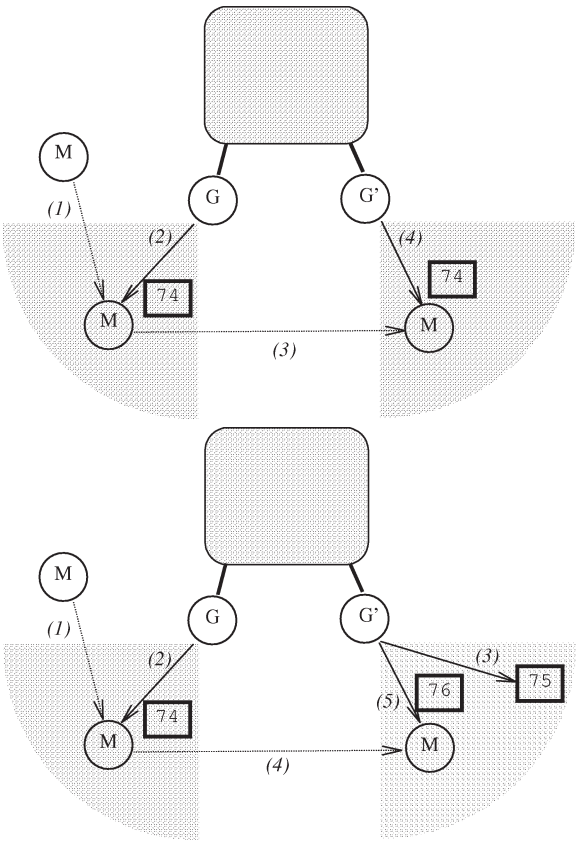

Figure 2. Examples of duplicates (left) and of missing multicasts (right). Boxes indicate sequence numbers.

NEW and are directed to S_C, that processes incoming NEW messages in sequence. For each such message, S_C changes the tag to NORMAL, appends to the message an increasing *sequence number*, and transmits the resulting message *m* to gateways through a FIFO-multicast. Gateways broadcast *m* to group members in the respective cells. Due to its movement across cells, any group member M could receive duplicates or could miss multicasts (figure 2). By maintaining a history of the received sequence numbers, M discards duplicates in the former case and sends to the gateway a proper NACK message in the latter. Upon receiving this NACK, the gateway will relay to M a copy of the missing multicasts, through a sequence of TRANSFER messages. We say that a multicast is *stable* iff it has been delivered by all group members [6]. S_C stores a copy of each multicast previously sent until it knows that the multicast is stable, which guarantees that a copy of any missing multicast is always available at S_C. For efficiency, however, gateways maintain locally a cache of previous multicasts. In other words, if a group member M remains unreachable "for a while" and then enters a cell C(G), the gateway G will use its local cache to bring M up-to-date and, in case of "long" disconnections, it will perhaps fetch "old" multicasts from S_C.

Group members piggyback acknowledgments for multicasts already delivered into NACK messages. Gateways extract this information and forward it to S_C within STABINFO messages. One such message contains a group member identifier and the largest sequence number that has certainly be delivered by that member. S_C thus acquires stability information about a group member whenever that member discovers that it missed a multicast. The fact that acknowledgments are piggybacked into other protocol messages that are needed anyway is beneficial for reducing power consumption on the mobile computers and for decreasing the aggregate load on the wireless links. Other protocols instead acknowledge every multicast individually [1,9].

2.3. Fundamental problems

The combination of mobility and incomplete spatial coverage makes communication between mobile computers and gateways intrinsically unreliable. In this section we will describe how the protocol copes with the related problems. The actions in this section are part of the protocol and, for simplicity, they will not be repeated in the detailed description of the next sections.

(A1) Whenever a mobile computer M sends a message *m* to a gateway, M periodically re-sends a copy of *m* (after expiring a specified time-out) until receiving the "matching" acknowledgment *ack*(*m*)[1]; furthermore, M does not send any new message m' until receiving *ack*(*m*).

(A2) When a gateway G receives a message *m*, it responds to the sending mobile computer with the related acknowledgment *ack*(*m*).

(A3) A mobile computer discards duplicate acknowledgments[2].

Let G be a gateway that has received a message *m* from a mobile computer:

(A4) G processes *m* locally only if *m* carries a NACK *tag*, otherwise it forwards *m* to S_C.

As a result of (A1)–(A2), a copy of *m* could arrive at multiple gateways and a given gateway could receive the same message multiple times. However, the protocol is structured so that this scenario is not a problem.

The above actions make communication from a mobile computer M to S_C reliable, in the sense that messages are not lost. However, S_C may receive *duplicates* and *out-of-order* messages, as the following example shows. M sends *msg1* to S_C through a gateway G; after a specified time-out, M decides to re-send *msg1*, but this time *msg1* happens to travel through a different gateway G'; S_C receives the copy of *msg1* from G'; S_C receives "many" further messages from M through G'; S_C receives the copy of *msg1* that M

[1] If M happens to be located outside of all cells, it postpones retransmissions until entering a cell again.

[2] As an example of these duplicates, suppose M sends multiple times a given message *m* until receiving the matching *ack*(*m*) and then sends a further message m': while M is waiting for *ack*(m'), it could receive another copy of *ack*(*m*).

sent "a long ago" through G. In short, S_C could receive a duplicate of *any* arbitrary past message sent by M. The related problems are solved as explained below.

When a computer M wishes to join the group, it selects an identifier, called *join-id*, and encloses it in the JOIN message directed to S_C. This *join-id* must be different from any other *join-id* that M selected in the past. Similar assumptions are often necessary in distributed computing and, in practice, *join-id* may be implemented by means of a 32-bit random pattern [2,17]. The membership of the group is maintained by S_C as a set of *member identifiers*. Each member identifier is a pair (M, *join-id*), where M is a computer identifier and *join-id* was selected by M upon joining the group. As will be shown in the next sections, a group member encloses its member identifier in each message that it sends. This information is used as follows:

(A5) S_C discards any received message tagged either NEW or LEAVE and carrying a member identifier that is not part of the current group membership.

This simple strategy causes S_C to ignore any NEW or LEAVE message sent by a computer M that either (i) is not a group member anymore; or (ii) is a group member, but it sent the message *before* its last joining of the group. For instance, suppose that M sent a message *m* while it was a group member. Case (i) is when *m* arrives at S_C after that M has left the group, whereas case (ii) is when *m* arrives at S_C after that M has left *and re-joined* the group (perhaps multiple times). In particular, (A5) eliminates the problem of duplicate LEAVE messages: when S_C receives the first LEAVE it removes the sending group member from the group, thus any subsequent copy of the same LEAVE will be discarded. Concerning NEW messages, (A5) discards duplicates of "old" messages but not the duplicates sent by current group members. These duplicates are detected by a proper sequence number that the sending group member encloses in each NEW message (section 3.1).

Clearly, (A5) cannot be applied to JOIN messages because otherwise no computer would become a group member. The handling of duplicates for JOIN messages is performed as follows. S_C maintains a cache of recently-used member identifiers called *membership cache*. An entry for member identifier *mid* is inserted into the membership cache as soon as *mid* has left the group. This entry will be removed after a time sufficiently long to ensure that no messages related to *mid* are still in transit (once again, assumptions of this kind are practically reasonable and are often necessary in distributed computing [17]).

(A6) S_C discards any JOIN message carrying a member identifier that either is part of the current membership or is contained in the membership cache.

The membership cache allows detecting duplicates that arrive at S_C when the sending computer has already joined *and left* the group. For instance: (i) M repeatedly sends a ⟨JOIN, *join-id1*⟩ and is admitted into the group; (ii) M sends a ⟨LEAVE, *join-id1*⟩ and is removed from the group; (iii) S_C receives a copy of the ⟨JOIN, *join-id1*⟩. Without the membership cache, S_C would not be able to detect that the message received at (iii) is a duplicate, thus, it would erroneously let M enter the group again.

It shall be observed that although JOIN and LEAVE may arrive at S_C out-of-order, S_C processes them without reordering. The protocol in [9] does not exhibit this feature and instead assumes that these "control" messages are processed in the same order as they were generated. In particular, if one such message, e.g., a LEAVE, is sent while their hand-off procedure is still in progress, they assume that the message is processed *after* hand-off is completed. They do not give the related details for fulfilling this assumption but acknowledge that this re-ordering is inefficient.

2.4. Beaconing

The protocol code running at each gateway makes use of a set of mobile computer identifiers called `local` and of a set of group member identifiers called `local-m`. `Local` identifies the mobile computers located in the corresponding cell. `Local-m` contains the member identifier of each computer in `local` that is a group member. Both data structures are maintained with the help of a *beaconing protocol*. Every T_B seconds, the gateway broadcasts a "beacon" within its cell. Upon receiving a beacon, a mobile computer announces its presence in the cell by sending back a response "greeting" message that contains its computer identifier. On the basis of the greetings received, each gateway dynamically updates `local` accordingly. For brevity, we will not enter into the details of this protocol [10,13], since it does not provide any insight to our discussion. We shall only assume that it satisfies the following informal properties:

(B1) If a mobile computer M enters a cell C(G) and remains in C(G) for a sufficiently long time, then M will be included in `local` at the gateway G.

(B2) If a mobile computer M leaves C(G) and remains outside of C(G) for a sufficiently long time, then M will be removed from `local` at G.

(B3) Let $C(G_1), \ldots, C(G_n)$ be a set of cells that spatially overlap; a mobile computer that remains in the overlapping area for a sufficiently long time belongs to `local` of only one gateway in $G_1, \ldots, G_n$.

The "sufficiently long time" is, essentially, T_B plus the timeout set by the gateway for receiving the corresponding responses. The actual value of T_B depends on the speed of user movements and on the cell size. In case of "picocells" (e.g., ten meters) and users that move at 3 m/s it could be 100 ms [13]. Property (B3) may be enforced by mobile computers, for instance, on the basis of the "strength"

of the beacon. We shall assume that each group member piggybacks in its greeting messages a field stating "*I am a group member and mid is my member identifier*". On the basis of this field, whenever a gateway updates `local`, it updates also `local-m`, if necessary. To simplify the presentation, we shall assume that when a gateway receives a message sent by a mobile computer M, M has already been inserted in `local` (and in `local-m`, if necessary). This assumption may be easily satisfied in practice. Modifying the protocol so as to remove this assumption is not difficult, however.

It is worth emphasizing that:

(i) When a group member switches between cells, the related gateways do not exchange any state information about this computer, but they simply update *autonomously* the respective `local` and `local-m`; this feature contributes substantially to keeping the entire protocol simple.

(ii) We allow messages exchanged between *mid* and G to be lost even while *mid* continues to belong to `local-m` at G; for instance, there could be a physical obstruction in the related cell C(G) or *mid* could leave and re-enter C(G) so quickly that its movement is not tracked by the beaconing protocol.

(iii) *mid* could (temporarily) belong to `local-m` at multiple gateways, because, for instance, the execution of the beaconing protocol triggered by a movement of *mid* from cell C(G) to cell C(G′) cannot complete simultaneously at G and G′.

In short, we interpret the content of `local` and `local-m` only as "*hints*" about the actual location of the related mobile computers. As will be clear, we use `local` and `local-m` only for allocation of data structures at gateways and for involving in the protocol only those gateways whose cells actually contain group members. In particular, we do *not* use them for building chains of pointers along which to forward messages waiting for delivery.

3. The protocol

The protocol is described partly in pseudo-code and partly in textual form. Messages are processed in sequence, in the order they are received. The actions to be executed for a given message are determined by the tag of that message. These actions are enclosed, in the pseudocode, within an `Upon` statement. We omit the obvious synchronization actions among flows of execution that share memory. Tables with one entry for each group member are accessed associatively, that is, `t`[*mid*] denotes the entry of table `t` associated with group member *mid*. We shall say that a (mobile) computer M sends a message *m* "reliably" to mean that M keeps on re-transmitting *m* until receiving the matching acknowledgment *ack*(*m*) from a gateway (section 2.3). Finally, we shall say "multicast with sequence number s" (or "sth multicast") to mean "NORMAL or TRANSFER message with sequence number s".

3.1. Coordinator

The coordinator S_C maintains the following data structures.

(i) `members`: the set of identifiers of group members;

(ii) `seqnum`: the sequence number of the last NORMAL message sent;

(iii) `norm-buffer`: a table with one entry per each multicast that is not known to be stable; each entry contains a copy of the multicast and a set (`pending`) of group member identifiers, one for each group member that might not have delivered the multicast yet;

(iv) `first`: a table with one entry for each group member; `first`[*mid*] contains the *lowest* sequence number among the multicasts that *mid* must deliver;

(v) `last`: a table with one entry for each group member; `last`[*mid*] contains the *highest* sequence number among the multicasts that *mid* must deliver, or 0, if this number is not known yet (e.g., S_C has not received yet a LEAVE message from *mid*);

(vi) `stable`: a table with one entry for each group member; `stable`[*mid*] contains a sequence number having the following interpretation: *mid* has certainly delivered all multicasts with sequence number $s \in$ [`first`[*mid*], `stable`[*mid*]]; `stable`[*mid*] is 0 if S_C does not know which multicasts have been delivered by *mid*;

(vii) `recv`: a table with one entry for each group member; `recv`[*mid*] contains the sequence number of the last in-order NEW message that the coordinator has received from *mid*.

Initially, `members` is empty, `seqnum` is 0, and `norm-buffer` is empty. S_C may receive from gateways requests to provide the copy of one or more messages contained in `norm-buffer`. We do not give the obvious details for brevity.

S_C may receive JOIN and LEAVE messages, whose handling is in figure 3. In a message ⟨JOIN, M, *join-id*⟩, M is the identifier of a computer that wants to join the group and *join-id* is the number selected by M to this purpose. A message ⟨LEAVE, *mid*⟩ is sent by a group member *mid* that wants to leave the group. Figure 4 shows examples of the data structures at the coordinator upon the joining (left) and the leaving (right) of a member *mid*. In the right figure, the actual values of `stable`[*mid*] and `recv`[*mid*] depend on the messages STABINFO and NEW that S_C has received at that moment (see below).

S_C may receive messages of the form ⟨NEW, b, s, *mid*⟩. One such message *m* informs S_C that the sending group member *mid* wants to multicast b and that *m* is the sth

```
1  Upon receiving <JOIN, M, join-id>
2     mid := (M, join-id);
3     insert mid in members;
4     allocate entry for mid in first,last,stable,recv;
5     seqnum := seqnum + 1;
6     first[mid] := seqnum;
7     last[mid] := 0;
8     stable[mid] := 0;
9     recv[mid] := 0;
10    msg := <NORMAL, seqnum, "mid has joined">;
11    FIFO-multicast msg to all gateways;
12    insert msg in norm-buffer with pending field
                        equal to members;
13 Upon receiving <LEAVE, mid>
14    remove mid from members;
15    seqnum := seqnum + 1;
16    last[mid] :=seqnum;
17    msg := <NORMAL, seqnum, "mid has left">;
18    FIFO-multicast msg to all gateways;
19    insert msg in norm-buffer with pending field
                        equal to members and mid;
20 Upon receiving <LEAVEOK, mid>
21    remove entry for mid from first,last,
                stable,recv, if present;
22    foreach entry g in norm-buffer {
23       remove mid from g.pending, if present;
24       if (g.pending is empty) then
25          remove g;
26    }
27    FIFO-multicast <CLEANUP, mid> to all gateways;
```

Figure 3. Actions at the coordinator.

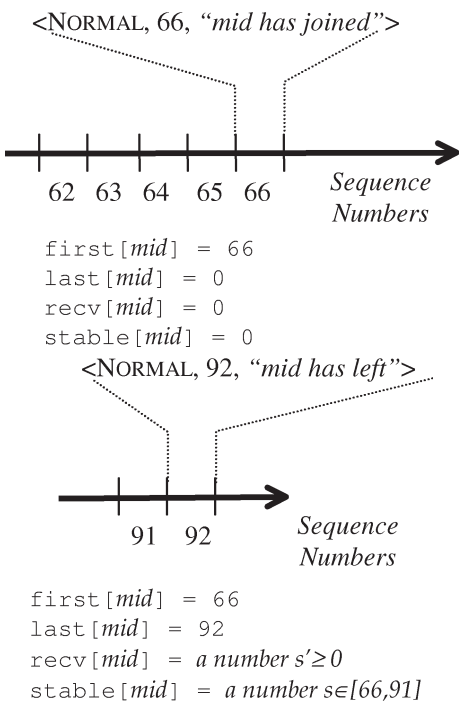

Figure 4. Example of data structures at the coordinator upon joining (left) and leaving (right) of *mid*.

NEW message sent by *mid*. Upon receiving *m*, S_C performs the following steps:

(1) If $s =$ `recv`[*mid*] $+ 1$, perform the following steps.

(2) Increment `recv`[*mid*] and `seqnum`.

(3) Build a message $m' = \langle$NORMAL, `seqnum`, $b\rangle$.

(4) Transmit m' to all gateways through a FIFO-multicast.

(5) Insert m' into `norm-buffer`, with pending set to the current value of members.

If $s <$ `recv`[*mid*] $+ 1$, S_C discards *m* since it is a duplicate. If, finally, $s >$ `recv`[*mid*] $+ 1$, S_C buffers *m* until steps 1–5 can be applied to it, that is, until the missing NEW messages originated by *mid* and still in transit through the wired network have arrived at S_C. We do not provide the details of this buffering for brevity.

S_C may receive messages of the form $\langle$STABINFO, s, *mid*$\rangle$, where *mid* denotes a group member and s is a sequence number such that *mid* has certainly delivered all multicasts with sequence number up to s (included). Upon receiving one such message *m*, S_C performs the following steps:

(1) If `members` does not contain *mid* or `stable`[*mid*] $\geqslant s$, skip the remaining steps.

(2) Assign `stable`[*mid*] equal to s.

(3) For each entry `g` in `norm-buffer`, if `g.seqnum` $\leqslant$ `stable`[*mid*] then remove *mid* from `g.pending` and, if `g.pending` has become empty, remove entry `g`. This step discards copies of multicasts that have become stable.

S_C may receive messages of the form $\langle$LEAVEOK, *mid*$\rangle$, where *mid* denotes a mobile computer that is leaving the group (section 3.3). A LEAVEOK acknowledges the delivery of all messages that *mid* was supposed to deliver (e.g., those in [`first`[*mid*], `last`[*mid*]]). The handling of LEAVEOK (figure 3) is similar to the handling of a $\langle$STABINFO, `last`[*mid*], *mid*$\rangle$, except that in this case: (i) *mid* is not contained in `members`; (ii) S_C releases all resources held by *mid* (lines 21, 23) and instructs the gateways to do the same (line 27). The clause "if present" at lines 21 and 23 is necessary because a LEAVEOK might be a duplicate.

3.2. Gateway

Each gateway G maintains the following data structures:

(i) `local-m`: the sets of identifiers of group members in the cell (section 2.3);

(ii) `first`, `last`: two tables with the same meaning as at S_C;

(iii) `norm-cache`: a cache of NORMAL messages recently received from S_C;

```
1  Upon receiving <NORMAL, seq, body>
2    if (body = "mid has joined") then {
3      allocate entry for mid in first, last;
4      first[mid] := seq;
5      last[mid] := 0;
6      if (the related mobile computer ∈ local) then
7        insert mid in local-m;
8    } elseif (body = "mid has left") then {
9      last[mid] := seq;
10   }
11   broadcast received message in the cell;
12 Upon receiving <NACK, s1, s2, mid>
13   if (s1 < trans[mid]) then
14     discard received message and return;
15   if (no entry for mid exists in first) then
16     discard received message and return;
17   send <STABINFO, mid, s1> to coordinator;
18   execute StateTransferAbort(mid);
19   spawn flow of execution for executing
                          StateTransfer(mid,s1,s2);
20 Procedure StateTransfer(gm, low, high)
21   allocate entry trans[gm];
22   trans[gm] := max(low + 1, first[gm]);
23   num := trans[gm];
24   if (last[gm] ≠ 0) then
25     high := last[gm] + 1;
26   while (num < high) {
27     msg := copy of num-th message;
28     set tag of msg to TRANSFER;
29     send msg to gm;
30     num := num + 1;
31   }
32   remove entry trans[gm];
33 Procedure StateTransferAbort(gm)
34   if (∃ trans[gm]) then {
35     abort StateTransfer;
36     remove entry trans[gm];
37   }
38 Upon receiving <CLEANUP, mid>
39   remove entries for mid from first, last;
40   if (mid ∈ local-m) then
41     remove mid from local-m;
42   execute StateTransferAbort(mid);
```

Figure 5. Actions at each gateway.

(iv) `trans`: a table with one entry per each group member in `local-m` that is receiving from G copies of previous multicasts (see below).

Initially, `local-m`, `first`, `last`, `norm-cache` and `trans` are empty. The obvious actions for managing `norm-cache` (e.g., for inserting/purging NORMAL messages) will be omitted for brevity, since they do not affect the correctness of the protocol.

G may receive NORMAL messages from S_C. Their handling is detailed in figure 5: G inspects the payload of the message *m* (in practice, G will inspect only an additional dedicated field) for updating `first` and `last` as necessary and broadcasts *m* in the cell. The values of `first` and `last` will be used for determining which messages to relay upon receiving a NACK (see below).

G may receive messages of the form ⟨NACK, *s1*, *s2*, *mid*⟩ (figure 5). One such message informs G that the sending group member *mid* has delivered all multicasts with sequence number up to s_1 (included) but that it missed those with sequence number $s \in [s_1 + 1, s_2 - 1]$. Accordingly, G starts transferring to *mid* a copy of the missing multicasts (line 19) and records in trans that a transfer to *mid* is in progress (lines 21–23)[3]. The copies at line 27 are obtained from either the `norm-cache` or the `norm-buffer` at the coordinator. Lines 15–16 handle the case in which the NACK arrives before than G has received from S_C the "*mid has joined*" message. Finally, when a group member *mid* leaves `local-m`, G executes StateTransferAbort(*mid*) (line 33).

3.3. Group member

Each group member *mid* maintains the following data structures:

(i) `mystable`: a variable containing the sequence number of the last multicast delivered;

(ii) `mynew`: the sequence number of the last NEW message sent;

(iii) `mybuf`: the set of multicasts that have been received but cannot be delivered yet;

(iv) `mylow`: the smallest sequence number among the multicasts in `mybuf`, or 0, if `mybuf` is empty;

(v) `mynack`: a copy of the last NACK message sent, or a null message (see below).

Initially, `mystable` and `mynew` are 0, `mybuf` is empty and `mynack` contains a null message.

A group member *mid* that wants to multicast *b* into the group performs the following steps:

(1) Increment `mynew`.

(2) Generate a message $m = \langle$NEW, *b*, `mynew`, *mid*$\rangle$.

(3) Send *m* reliably.

A group member receives multicasts by means of messages tagged either NORMAL or TRANSFER (figure 6, *my-mid* at lines 15 and 20 indicates the identifier of the executing group member). A multicast is delivered only if it carries a sequence number equal to `mystable` + 1 (lines 4–7, 8–13). Upon delivering a multicast, *mid* increments `mystable` and attempts to deliver multicasts previously

[3] The case $s_1 =$ `trans`[*mid*] must satisfy the check at line 13 because if *mid* sends a ⟨NACK, *s1*, *s2*, *mid*⟩ and then misses the first TRANSFER message in the corresponding sequence, then *mid* sends a further identical NACK that must indeed be handled.

```
1  Upon receiving <NORMAL or TRANSFER, seq, body>
2    if (seq < mystable + 1) then
3      discard received message and return;
4    if (seq-th element ∉ mybuf) then {
5      insert received message in mybuf;
6      update mylow, if necessary;
7    }
8    while ((mystable + 1)th element ∈ mybuf) {
9      extract (mystable + 1)th element from mybuf;
10     update mylow, if necessary;
11     deliver body of the element;
12     mystable := mystable + 1;
13   }
14   if (mynack = null message) and (mylow ≠ 0) then {
15     mynack := <NACK, mystable, mylow, my-mid>;
16     send mynack reliably and set timer;
17   }
18 Upon timer expiring
19   if (mylow ≠ 0) then {
20     mynack := <NACK,mystable,mylow,my-mid>;
21     send mynack reliably and set timer;
22   } else
23     mynack := null message;
```

Figure 6. Actions at each group member.

buffered (8–13). *Mid* realizes that it has missed one or more multicasts upon receiving a message that shall be buffered[4]. In this case, it notifies the gateway by means of a message $m = \langle$NACK, `mystable`, `mylow`, *mid*$\rangle$. Obviously, *mid* shall not send a NACK upon every insertion in the buffer, because it could have just sent one. On the other hand, the fact that *mid* has sent a NACK does not imply that it will receive all required multicasts, because some TRANSFER messages could be lost. For this reason, we have employed the following strategy. Upon sending a NACK, *mid* sets a timer (independent of the timer of section 2.3 for sending the NACK reliably). While the timer is set, *mid* does not send any further NACK. When the timer expires, *mid* sends a further Nack only if it is still aware that it missed a multicast (18–23). The correspondence between these steps and figure 6 is obvious (retransmission is postponed when *mid* happens to be located outside of all cells).

A computer M that wants to *join* the group performs the following steps:

(i) Start listening to messages directed to the group.

(ii) Assign `mystable` equal to the sequence number of the first NORMAL message received.

(iii) Send a $\langle$JOIN, M, *join-id*$\rangle$ reliably.

(iv) Start executing the protocol of figure 6, except that line 11 (delivery) is skipped.

Let *m0* denote the NORMAL message with body "(*M,join-id*) *has joined*" and let s_0 denote the sequence number of *m0*. As soon as M receives *m0*, M executes the following actions and then switches to the "ordinary" protocol of figure 6 (that is, it stops skipping line 11):

(v.i) Set `mystable` equal to $s_0 - 1$.

(v.ii) Remove from `mybuf` all elements with sequence number $s < s_0$.

A group member *mid* that wants to *leave* the group performs the following steps:

(i) Send a $\langle$LEAVE, *mid*$\rangle$ reliably.

(ii) Keep on executing the protocol until both the following conditions hold:

  (a) *mid* has received a message $m' = \langle$NORMAL, s, "*mid has left*"$\rangle$;

  (b) *mid* has delivered all multicasts up to sequence number $s - 1$ (included).

(iii) Send a $\langle$LEAVEOK, *mid*$\rangle$ reliably.

It could be possible to employ a looser semantics and allow a group member *mid* to leave the group as soon as it completes step (i) (S_C would discard all resources held by *mid* upon receiving the LEAVE). In this case, the highest sequence number among the multicasts delivered by *mid* could be *any* number in [`first`[*mid*], `last`[*mid*]]. A similar semantics is employed in [9].

3.4. Example

As an example, consider a group member *mid* that enters a cell C(G) and receives a NORMAL message with sequence number s from the gateway G. If $s =$ `mystable` $+ 1$, then *mid* delivers the message and increments mystable. If $s <$ `mystable` $+ 1$, *mid* discards the message. If $s >$ `mystable` $+ 1$, *mid* buffers the message and responds with $m = \langle$NACK, `mystable`, `mylow`, *mid*$\rangle$. Upon receiving *m*, G transmits to *mid* a copy of the multicasts with sequence number $s' \in$ [`mystable` $+ 1$, `mylow` $- 1$] by means of a sequence of TRANSFER messages. Upon receiving each message in the sequence, *mid* delivers the message and increments `mystable`. Concurrently with these steps, *mid* may receive further NORMAL messages. These will be either discarded or buffered by the same reasoning as above. Buffered messages are delivered, if possible, when *mid* delivers a message.

If *mid* moves to another cell before having received all TRANSFER messages a similar scenario will be repeated in the new cell, except that `mystable` may be greater, if *mid* delivered at least a message while in C(G). If *mid*

[4] If *mid* wanted to discover more promptly whether it missed a multicast upon entering a cell, it could autonomously send a NACK to the gateway. The same effect can be achieved more efficiently (at the expense of making the protocol less modular) if *mid* piggybacks into the message of the beaconing protocol the sequence number of the last message delivered [9]. Both optimizations require small changes to the actions performed at each gateway, that we do not detail for brevity.

leaves and re-enters the *same* cell C(G), G will abort the state transfer upon *mid*'s leaving and will spawn a new instance upon receiving the next NACK sent by *mid*. If the leaving and re-entering of *mid* is "so fast" that it is not reflected in the `local-m` at G, then G will not spawn a further state transfer and some TRANSFER messages could not arrive at *mid*. By the same reasoning as for NORMAL messages, however, *mid* is able to decide whether it shall send a further NACK. Upon receiving this NACK, G will abort the state transfer possibly in progress and will spawn a new instance, as appropriate.

Let *m0* denote the message with body "*mid has joined*" and let s_0 denote the sequence number of *m0*. At the beginning, we have implicitly assumed that when *mid* enters C(G) it has already received *m0*. If this was not the case the discussion remains unmodified, except that *mid* drops any message with sequence number smaller than s_0. The fact that *mid* will indeed receive *m0* follows by the same reasoning as above, having observed that the initial value for `mystable` at *mid* is smaller than s_0.

Observe that when group members move across cells, *no* message need be exchanged in the wired network. Not only this feature contributes to keeping low the traffic on the wired network, it also contributes to decreasing the computing load on gateways and, consequently, the delay by which they process each protocol message. It may occasionally happen that *mid* receives a multicast that has already delivered. This event may typically happen within a "short" time window after switching between cells, in particular, in the following scenario (figure 2, left). Suppose *mid* delivers the sth multicast while in the cell C(G), then switches to cell C(G′). Due to delays on the wired network, G′ receives the sth multicast "much later" than G and transmits this multicast when *mid* is in C(G′). Other protocols [1,9] insist on preventing similar scenarios (at the expense of requiring a hand-off upon each cell switching) on the grounds that duplicates may introduce a cost in terms of wireless bandwidth and battery power at group members. We preferred a different approach because fixed costs of this nature are introduced at lower protocol levels anyway, given the broadcast nature of the wireless medium. For instance, suppose that a given message *m* has to be sent to a group member mid_0 and that other group members in the same cell have already delivered *m*. Even if the layer that implements the multicast protocol attempted to avoid duplicates by addressing *m* only to mid_0, there would be no real saving in terms of wireless bandwidth and battery power, because *m* has to be handled anyway by the lower protocol layers of *all* computers in the cell. As an aside, we observe also that:

(i) With our protocol a group member *may* receive unnecessary messages, but this does not happen *necessarily*;

(ii) the energy for *receiving* a message is much less than the energy for *sending* a message (up to one order of magnitude) [12];

(iii) while in a cell, a mobile sends and receives "many" messages anyway, because of the beaconing protocol.

3.5. Correctness

The correctness of the protocol is established through the following properties:

P1 (Total order). Any two group members that deliver two multicasts, deliver them in the same order.

P2 (Duplicates). A group member does not deliver duplicate multicasts.

P3 (Membership). Any computer that wants to be a group member eventually will become a group member; moreover, any group member that wants to leave the group eventually will leave the group.

P4 (Non-triviality). A group member *mid* delivers all multicasts with sequence number in [`first`(*mid*), `last`(*mid*)].

P5 (Validity). If a group member wants to multicast *m*, then all group members eventually will deliver *m*.

P6 (Integrity). Any multicast or membership change delivered by a group member has been originated by a group member.

P7 (Garbage collection). All resources allocated to a computer that has left the group are freed[5].

Properties P1, P2 are the essence of the protocol: if group members deliver multicasts then they deliver them in the desired total order and with no duplicates. However, these properties are clearly not sufficient since they would be satisfied even by a protocol in which group members do not deliver any multicast and/or no computer ever manages to become a group member. Similar trivial solutions are ruled out by properties P3–P5: there are indeed group members (P3), group members indeed deliver multicasts (P4), group members can indeed generate multicasts (P5). Finally, P6 guarantees that neither multicasts nor membership changes are generated capriciously. In the remaining part of the section we shall prove the above properties. We shall assume that any computer M is able to send reliably a message *m* to the coordinator S_C. This assumption can be easily satisfied unless M *always* enters and leaves cells so quickly that *m* never arrives at any gateway or *ack*(*m*) never arrives at M.

The following lemmas are valid for any group member *mid*. More in detail, for any computer that has completed step (iii) of the actions in section 3.3 for becoming a group member (e.g., that has initialized `mystable` and has sent reliably the JOIN message) and that has not sent the pertinent LEAVEOK yet.

[5] Except for the entry in the membership cache (section 2.3).

L1 *mid* delivers multicasts in the order of increasing sequence numbers[6].

L2 *mid* updates `mystable` only upon delivering a multicast and each update is an increment.

L3 `mylow` $\neq 0 \Rightarrow$ `mylow` $>$ `mystable`, *mid* has received a multicast with sequence number `mylow` and no multicast with sequence number in [`mystable` $+ 1$, `mylow` $- 1$].

L4 `mystable` $= s$ and *mid* has a copy of the $(s + 1)$th multicast $m \Rightarrow$ *mid* delivers m and increments `mystable`.

L5 *mid* transmits a $\langle$NACK, h_0, $h_0 + k_0$, *mid*$\rangle$ ($h_0 > 0$, $k_0 > 1$) while in cell C(G) and remains in C(G) for a sufficiently long time $\Rightarrow$ eventually G will spawn a state transfer procedure for transferring to *mid* multicasts in $[k_{\text{MIN}}, k_{\text{MAX}}]$, where

$$k_{\text{MIN}} = \max\big(h_0 + 1, \textit{first}(\textit{mid})\big),$$
$$k_{\text{MAX}} = \min\big(h_0 + k_0 - 1, \textit{last}(\textit{mid})\big).$$

L6 `mystable` $= h_0$ and `mylow` $= h_0 + k_0$ ($h_0 > 0$, $k_0 > 1$) $\Rightarrow$ eventually *mid* will receive all multicasts with sequence number in $[k_{\text{MIN}}, k_{\text{MAX}}]$, where

$$k_{\text{MIN}} = \max\big(h_0 + 1, \textit{first}(\textit{mid})\big),$$
$$k_{\text{MAX}} = \min\big(h_0 + k_0 - 1, \textit{last}(\textit{mid})\big).$$

Lemmas L1–L4 follow immediately from the pseudocode in figure 6 and the textual description in section 3.3. As for L5, it must be `mylow` $\neq 0$ (figure 6, lines 14–15 and 19–20) and we obtain from L3 that *mid* has missed at least a multicast in $[h_0 + 1, h_0 + k_0 - 1]$. It follows again from figure 6 and L3 that *mid* will re-transmit a NACK until receiving the missing multicasts. On the other hand, *mid* can obtain the missing multicasts only if the local gateway G spawns a state transfer procedure for *mid*, that is, only if G receives a NACK that is not discarded at lines 14 and 16 of figure 5. The check at line 13 certainly fails because `mystable` never decreases (L2). The check at line 15 will certainly fail after G has received the "*mid has joined*" message from S_C, but since *mid* is a group member this message will certainly arrive. The values for k_{MIN} and k_{MAX} follow from lines 22–25 of figure 5.

The proof of L6 follows. *Mid* has received a multicast *m0* with sequence number $h_0 + k_0$ and no multicast with sequence number in $[h_0 + 1, h_0 + k_0 - 1]$ (L3); suppose that, upon receiving *m0*, `mynack` is a null message; *mid* sets `mynack` to $\langle$NACK, h_0, $h_0 + k_0$, *mid*$\rangle$, sends `mynack` reliably and sets a timer (figure 6, lines 14–17); when the timer expires, there are two cases to consider (lines 18–23):

(i) if `mylow` has become 0 then `mybuf` is empty by definition, thus the $(h_0 + k_0)$th multicast has been delivered; the proof then follows from L1 and L2;

(ii) otherwise, *mid* updates `mynack` with the current values of `mystable` and `mylow`, sends `mynack` reliably and sets the timer again.

Let $\langle$NACK, h_i, $h_i + k_i$, *mid*$\rangle$ denote the message sent upon the ith expiration of the timer. It is $h_{i+1} \geqslant h_i$ because `mystable` never decreases (L2). When the movement of *mid* allows applying L5 and *mid* does not miss all copies sent by the local gateway, it is $h_{i+1} > h_i$. Therefore, unless the movement of *mid* is such that these conditions are *never* satisfied, eventually either case (i) will apply or there will exist a j: $h_j \geqslant k_{\text{MAX}}$. If `mynack` was not a null message upon receipt of the initial message *m0*, then the reasoning is identical except that the $\langle$NACK, h_0, $h_0 + k_0$, *mid*$\rangle$ will be sent after the first expiration of the timer (it follows easily from figure 6 that `mynack` $\neq$ null message $\Rightarrow$ timer is set). The values for k_{MIN} and k_{MAX} follow from figure 5.

We are now able to prove P1–P7:

P1 and P2 follow immediately from L1 and the fact that sequence numbers are chosen by S_C.

P3: let M be a computer that wants to become a group member. Since we assume that M is able to send messages reliably, M will indeed become a group member because S_C eventually will process the relevant JOIN. The reasoning required when M wants to leave the group is identical, except that Leave messages have to be considered.

P4: let M be a computer that wants to become a group member. M will eventually become a group member because of P3. Let *mid* denote the member identifier of M and let h_0 denote the sequence number of the first multicast received by M since its attempt to join ($h_0 =$ `mystable`, $h_0 < \textit{first}(\textit{mid})$). M will eventually receive a multicast with sequence number h_1 ($\textit{first}(\textit{mid}) < h_1$), unless M always enters and leaves cells so quickly that it misses all multicasts with such sequence numbers. It follows from L6 that M will eventually receive all multicasts in [*first*(*mid*), $h_1 - 1$] and from L4 that M will deliver all multicasts in [*first*(*mid*), h_1]. The proof is obtained by iterating this reasoning and observing that M leaves the protocol after receiving the message with sequence number *last*(*mid*).

P5: a group member multicasts m by enclosing m in a message m' tagged NEW; since m' is sent reliably, eventually it will be processed by S_C; S_C will then transform m' into a message m'', tagged NORMAL and associated with a sequence number; the proof now follows from P4.

P6: multicasts and membership changes delivered by group members are enclosed in messages tagged either NORMAL or TRANSFER and a TRANSFER message is identical to a NORMAL message except for the tag. It suffices then to observe that: (i) NORMAL messages are generated only by S_C; (ii) S_C generates one NORMAL message for each NEW, JOIN, or LEAVE message that it processes; (iii) S_C processes only the received messages that are not duplicates

[6] If mid is still executing step (iv) of the actions for becoming a group member (section 3.3), then the action of "delivering a multicast" must be replaced by the action of "discarding a multicast that should be delivered". The same rephrasing must be applied to L2 and L4.

(section 2.3); (iv) messages tagged NEW, JOIN or LEAVE are generated only by group members.

P7: A computer M that is leaving the group stops participating in the protocol after sending reliably a LEAVE and a LEAVEOK to S_C. S_C frees the pertinent entry in `members` upon receiving the former and in `first`, `last`, `stable`, `recv` upon receiving the latter. Moreover, S_C frees all pertinent entries in the `pending` fields of `norm-buffer` upon receiving the LEAVEOK. All resources at gateways are freed upon receiving the CLEANUP message that is sent by S_C upon receiving the LEAVEOK.

Observe that group members may move at *any* time and that we require assumptions only about their *global* movement, e.g., group members do not move "very fast all the time".

4. Extensions

4.1. Restricting the set of gateways involved in the protocol

The protocol described so far handles mobility of group members with a sort of "brute-force" approach: the coordinator blindly sends NORMAL messages to *all* gateways. The messages sent to gateways whose cells do not contain any group member (*empty* cells) are obviously an unnecessary cost. This cost is the price to pay for avoiding the additional complexity (and messages) required for keeping the coordinator aware of which cells actually contain group members. Whether this strategy is reasonable or not depends primarily on the ratio R between the number of group members and the number of gateways. Let us assume that group members are distributed uniformly across cells. If $R \gg 1$ then, on average, there are no empty cells and no unnecessary messages. If $R \ll 1$ there are, on average, many empty cells and thus there are many unnecessary messages. For those scenarios in which these unnecessary messages on the wired network are really an issue, the protocol may be extended as described below. In particular, this extension is useful for scalability in terms of number of gateways and does not require any change at group members.

The data structures at the *coordinator* S_C are augmented by a `subscr-set`. This is a set of gateway identifiers, one for each gateway whose cell contains group members, and is initially empty. S_C sends NORMAL messages *only* to gateways in the `subscr-set` rather than to *all* gateways. Upon receiving a SUBSCRIBE message, S_C inserts the sending gateway G in the `subscr-set` and responds to G with a SUBSCOK message containing a copy of `first` and `last`. Upon receiving an UNSUBSCRIBE message, S_C removes the sending gateway from the `subscr-set`. The data structures maintained at each *gateway* are augmented by a boolean variable called `subscribed`. This variable tells the gateway whether it belongs to the `subscr-set` at S_C and is initially False. The only changes with respect to the protocol of the previous section concern the

```
1  Upon change in local-m
2    if (not subscribed) then
3      if (local-m is not empty) then {
4        send SUBSCRIBE to S_C;
5        subscribed := True;
6        wait for SUBSCOK from S_C and buffer
           messages received from group members;
7        initialize first, last, according to the
           SUBSCOK;
8        process buffered messages;
9      }
10   else
11     if (local-m is empty) then {
12       send UNSUBSCRIBE to S_C;
13       subscribed := False;
14       abort in-progress state transfer (if any)
           and remove entries in trans;
15       remove entries in first, last;
16     } else
17       "ordinary actions of section 3.2"
```

Figure 7. Modified actions at each gateway upon changes in `local-m`.

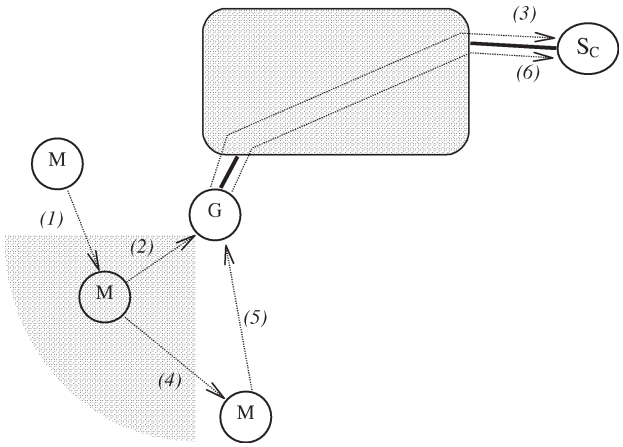

Figure 8. Start-up and shut down periods. Steps 2 and 5 denote the updating of `local-m`.

actions performed upon changes in `local-m`, that become as in figure 7. It can be seen that: (i) a gateway G enters the `subscr-set` when its cell starts containing group members; (ii) G leaves `subscr-set` when its cell does not contain any group member anymore; (iii) while in the `subscr-set`, G behaves as in the previous section.

Executions of the modified protocol are now characterized by two transitory periods, as follows. When a group member enters an empty cell of a gateway G not in the `subscr-set` (step 1 of figure 8), there is a transitory period during which G does not receive NORMAL messages from S_C. This *start-up* period ends when S_C processes the corresponding SUBSCRIBE message (step 3). When the last group member leaves the cell of a gateway G in the `subscr-set` (step 4), there is a transitory period during which G continues to receive (unnecessary) NORMAL

messages from S_C. This *shut down* period ends when S_C processes the UNSUBSCRIBE message (step 6).

During start-up, group members in cell C(G) can obviously miss multicasts. However, the *correctness* of the protocol is not affected because, informally, these group members can be considered as if they were still disconnected: They will recover the missing multicasts after an initial delay (e.g., once G has entered the `subscr-set`) but *automatically*. The duration of start-up is essentially the time it takes for a SUBSCRIBE message sent by a gateway to be processed by the coordinator. As long as this time is sufficiently small compared to the speed of users' movements, the resulting delay can be neglected in practice. As an aside, we observe that any multicast protocol that does not *always* send *all* multicasts to *all* gateways must exhibit a sort of start-up period when a group member enters an empty cell. For instance, the protocols presented in [1,9] require, among other things, the updating of a data structure replicated at a number of stationary computers. The *cost* involved in restricting the set of gateways that actually participate in the protocol is, in addition to the above delay, only *one* message on the stationary network per each change in the composition of the `subscr-set`. Moreover, these changes are requested by gateways only when they are really necessary.

4.2. Bounding the buffering space on mobile computers

Group members buffer received messages that cannot be delivered. Since the resources at mobile computers are costly, it may be practically useful to assume that the buffering space on the mobile computer is *bounded*, with "no buffering space" as an extreme case. To this end, it suffices to observe that the buffering of messages is not necessary for *correctness*, but it is merely an *optimization*. Consider a group member *mid* that sends a $\langle$NACK, $s-1$, $s+k$, *mid*$\rangle$, e.g., *mid* has received the $(s+k)$th multicast when it was waiting for the sth multicast. When *mid* has delivered all multicasts in $[s,\ s+k-1]$ (retransmitted by the gateway), it may deliver immediately also the $(s+k)$th multicast previously buffered. Suppose now that *mid* does not have any buffer space: Upon receiving the next multicast (e.g., the $(s+k+h)$th multicast), *mid* would simply request again the $(s+k)$th multicast. The protocol would still be correct, except that it would be less efficient. In other words, if *mid* did not have sufficient buffer space, it could send NACK's not because it really *missed* a certain multicast, but because it received that multicast "*too early*".

To enclose the consideration above in the protocol it suffices to: (i) maintain `mylow` as the lowest sequence number among the multicasts that have been received but not delivered yet (independently of the actual content of `mybuf`); (ii) buffer a multicast only if `mybuf` is not full; (iii) if `mybuf` is full and a received multicast *m* shall be buffered, discard *m* only if its sequence number is greater than all sequence numbers in `mybuf`, otherwise discard the multicast in `mybuf` with greatest sequence number and replace it by *m*. An implementation of the protocol for a given mobile computer may thus select the desired tradeoff between amount of buffering space and likelihood of unnecessary sending of NACK messages. The above considerations show also how to deal with mobile computers that are switched off for saving batteries (while they are not handling protocol messages). It suffices to save `mystable` and `mynew` on permanent storage before switching off and, when the computer will be switched on again, the data structures will be initialized as in section 3.3, except that `mystable` and `mynew` will be set to their previous values.

4.3. Skipping membership changes

Multicasts carrying membership changes are delivered to all group members. This feature may constitute an unnecessary cost for applications in which individual group members do not need to be kept informed of the current membership. When this cost is indeed an issue, it may be greatly reduced by modifying the protocol as explained in this section.

Each *gateway* broadcasts only those membership changes that involve group members located in its cell (by skipping lines 11 and 28 of figure 5 as necessary). In order to handle the related "holes" in the sequence numbers that can be observed by group members, each NORMAL message is augmented by an integer field called `prememb`. This field is filled by the *coordinator* S_C and contains the number of immediately preceding membership changes. In particular, `prememb` is 0 iff the previous multicast is not a membership change. In other words, let *m* denote the message being constructed and let *m0* denote the last NORMAL message that was not a membership change. Let s and s_0 denote the sequence number of *m* and *m0*, respectively. S_C assigns the prememb field of *m* equal to $s - s_0 - 1$. At *group members*, `mybuf` may now contain "dummy" elements. A dummy element is a special element that contains only a sequence number. Let *mid* be a computer that is neither joining nor leaving the group. The actions performed by *mid* upon receiving a message tagged either NORMAL or TRANSFER are modified with respect to figure 6 as follows (`prememb` and `seqnum` are fields of the received message): (i) if `prememb` $\neq 0$ then line 5 inserts in `mybuf` also a set of dummy elements, one for each sequence number in [`seqnum`-1, `seqnum`$-$`prememb`] that is not in `mybuf` already; (ii) if the element extracted from `mybuf` at line 9 is either a dummy element or a multicast carrying a membership change, then line 11 is skipped. In other words: S_C piggybacks into each message *m* a "compressed" description of the immediately preceding membership changes, if any; all recipients of *m* that are not interested in membership changes expand this description; holes in the received sequence numbers that are caused only by membership changes disappear automatically, with no additional messages.

5. Conclusions

We have presented a protocol for totally-ordered multicast communication within a dynamic group of mobile recipients. Group members are resource-poor computers that can move continuously across a number of wireless cells providing only incomplete spatial coverage. The protocol has been purposefully kept simple, because simplicity was considered as a powerful tool for tackling the many combined problems characterizing the overall scenario. From this point of view, the key aspects are that state information about the group is centralized and that movements of group members do not require interactions between gateways. These features also helped us in constructing the protocol upon practical mobility assumptions. In particular, in allowing group members to move at any time, irrespective of the part of the protocol that is being executed. The coordinator constitutes a single point of failure. However, computer failures were purposefully not addressed, since this simplification appeared to be a better starting point for capturing the essence of the problem. Moreover, the property that state information is centralized fits nicely and easily into the framework of established techniques for fault-tolerance such as, for instance, primary-backup replication and virtual synchrony in the context of process groups [3,5].

Acknowledgements

This work has been partially supported by the Italian Ministry of University, Research and Technology (M.U.R.S.T. 40%) and by the Commission of European Communities under Esprit Program Open Long Term Research project 20422 (Moby Dick). The author is grateful to Francesco Spadoni for identifying a mistake in the protocol.

References

[1] A. Acharya and B. Badrinath, A framework for delivering multicast messages in networks with mobile hosts, Mobile Networks and Applications 1(2) (1996).

[2] H. Attiya and R. Rappoport, The level of handshake required for establishing a connection, in: *Distributed Algorithms*, Lecture Notes in Computer Science, Vol. 857 (Springer, 1994) pp. 179–193.

[3] Ö. Babaoglu, A. Bartoli and G. Dini, Enriched view synchrony: a programming paradigm for partitionable asynchronous distributed systems, IEEE Transactions on Computers 46(6) (1997) 642–658.

[4] B. Badrinath, A. Acharya and T. Imielinski, Structuring distributed algorithms for mobile hosts, in: *Proc. of the 14th IEEE International Conference on Distributed Computing Systems* (June 1994) pp. 21–28.

[5] K. Birman, The process group approach to reliable distributed computing, Communications of the ACM 36(12) (1993) 36–53.

[6] K. Birman, A. Schiper and P. Stephenson, Lightweight causal and atomic group multicast, ACM Transactions on Computer Systems 9(3) (1991) 272–314.

[7] K. Birman and R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit* (IEEE Computer Society Press, 1994).

[8] K. Brown and S. Singh, M-UDP: UDP for mobile cellular networks, SIGCOMM Computer Communication Review (October 1996).

[9] K. Brown and S. Singh, RelM: Reliable multicast in wireless networks, Manuscript, Department of Computer Science, University of South Carolina (January 1996).

[10] R. Cáceres and L. Iftode, Improving the performance of reliable transport protocols in mobile computing environments, IEEE Journal on Selected Areas in Communications 13(5) (1995) 850–857.

[11] S. Deering and D. Cheriton, Multicast routing in datagram internetworks and extended LANs, ACM Transactions on Computer Systems 8(2) (1990) 85–110.

[12] G.H. Forman and J. Zahorjan, The challenges of mobile computing, IEEE Computer 27(4) (1994) 38–47.

[13] R. Ghai and S. Singh, An architecture and communication protocol for picocellular networks, IEEE Personal Communications (Third Quarter 1994) 36–46.

[14] V. Hadzilacos and S. Toueg, Fault-tolerant broadcasts and related problems, in: *Distributed Systems*, 2nd edition, ed. S. Mullender (ACM Press, 1993).

[15] D. Johnson and D. Maltz, Protocol for adaptive wireless and mobile networking, IEEE Personal Communications (February 1996) 34–42.

[16] F. Kaashoek and A. Tanenbaum, An evaluation of the Amoeba group communication system, in: *Proc. of the 16th IEEE International Conference of Distributed Computing Systems* (May 1996) pp. 436–447.

[17] B. Lampson, Reliable messages and connection establishment, in: *Distributed Systems*, 2nd edition, ed. S. Mullender (ACM Press, 1993).

[18] S. Maffeis, W. Bischofberger and K.U. Mätzel, A generic multicast transport service to support disconnected operation, Wireless Networks 2(1) (1996) 87–96.

Alberto Bartoli received a degree in electrical engineering, cum laude, in 1989, and a doctorate in computer engineering in 1994, both from the University of Pisa. In 1993 he joined the Dipartimento di Ingegneria dell'Informazione of the University of Pisa, where he is currently an Assistant Professor. His research interests include group-based computing, large scale distributed systems and mobile computing.
E-mail: alberto@iet.unipi.it