A scalable causal broadcast that tolerates dynamics of mobile networks

Daniel Wilhelm, Luciana Arantes, Pierre Sens

A scalable causal broadcast that tolerates dynamics of mobile networks

Daniel Wilhelm
Sorbonne University, CNRS
Inria, LIP6, Paris, France
daniel.wilhelm@lip6.fr

Luciana Arantes
Sorbonne University, CNRS
Inria, LIP6, Paris, France
luciana.arantes@lip6.fr

Pierre Sens
Sorbonne University, CNRS
Inria, LIP6, Paris, France
pierre.sens@lip6.fr

***Abstract*—Many distributed applications and protocols require causal broadcast. Various existing algorithms ensure causal order of broadcast messages, but they are either not scalable, or do not take into account the characteristics of mobile networks, such as nodes mobility, message losses, or limited capacity of nodes. This paper proposes a causal broadcast algorithm suitable for mobile networks since it copes with the dynamics, constraints, and specifications of such networks. Control information included in each message, and maintained on each node, is of small size and the algorithm handles message losses. Performance evaluation of experiments conducted on Omnet++ confirms the effectiveness of our causal broadcast protocol.**

I. Introduction

Causal Broadcast is a fundamental group communication service used by many distributed applications, such as distributed databases, publisher/subscribe systems, collaborative applications, or distributed social networks. It ensures that messages are delivered to all nodes (processes) only once, preserving causal relation of broadcast messages, i.e., the delivery of broadcast messages must respect Lamport's happened-before relationship [9]: if the broadcast of a message m precedes the broadcast of a message m′, then every process that delivers these two messages must deliver m before m′.

In this paper, we are particularly interested in providing a causal broadcast service for wireless mobile networks [8], composed of mobile nodes and reliable support stations. The dynamics of such networks where mobile nodes can move, leave/join the system, and fail, poses new challenges for the implementation of the group communication service. For instance, if a mobile node joins an executing distributed application where other nodes have already delivered and broadcasted some messages, the new node should not be blocked, waiting for these messages, if it will never receive them. Furthermore, the protocol must deal with message losses, the low memory capacity of mobile nodes, and, depending on the system, a high number of mobile nodes.

Many approaches have been proposed in the literature that guarantees and implements a message causal order. The two most well-known ones are (1) the piggybacking of causal per node information in each message, such as logical vector clocks [6][10], and (2) flooding through FIFO links [7], where messages are systematically forwarded at first reception. The dissemination pattern through FIFO links ensures that there exists no path between two nodes over which messages are sent out of causal order. The first approach is not suitable for tackling the dynamics and scalability issues of mobile networks, because the size of causal information depends on the number of nodes of the system. Therefore, we have chosen the second approach to implement our causal broadcast protocol. However, the latter, which was proposed by Friedman et al. [7], only offers causal order over static distributed systems where network topology does not change.

The authors in [13] have extended Friedman et al.'s broadcast protocol to dynamic systems, using data structures that do scale. On the other hand, their solution to cope with system dynamics does not address the issue of free mobile nodes movement since the network overlay must always be connected through links previously initialized by a particular handoff procedure. Therefore, a path of initialized links must always exist between each pair of nodes. Moreover, links are all supposed to be FIFO, reliable, and initialized in both directions. These characteristics are not realistic for mobile networks, which make [13] not suitable for such networks.

Our causal broadcast algorithm is designed for mobile networks, taking into account their intrinsic characteristics and constraints. Mobile nodes can join/leave the system, move, and temporarily fail. They are connected to support stations through a wireless network, which is neither reliable nor FIFO. Our algorithm renders them FIFO and reliable by applying message retransmission, sequence number assignment, and message reception acknowledgment. On the other hand, since support stations are connected by a wired network, existing protocols, such as TCP, ensure reliable FIFO communication among them. It is worth emphasizing that messages piggyback few control information, and memory usage complexity is low for mobile nodes, while, for support stations, it grows linearly with the number of local connected mobile nodes. Hence, our broadcast requires less control information than vector clocks and does not make the constraining assumptions of the flooding approach [7][13][12]. Performance evaluation results of experiments conducted over the simulator OMNeT++/INET [18] confirm the advantages of our solution.

The rest of the paper is organized as follows. Section II gives some background on causal broadcast. Section III presents the system model. In Section IV, we describe our proposed causal broadcast protocol. Section V presents evaluation of results on OMNet++. Section VI discusses related work and, finally, Section VII concludes the paper.

II. Background

Mobile Networks are usually composed of a huge number of nodes, which render not sustainable full system membership knowledge by nodes. Instead, they have just a local partial view of the system, which usually contains much fewer nodes than the whole system, and only communicate with the nodes, denoted neighbors, that belong to this partial view. Messages are, therefore, disseminated transitively through an overlay network built with the local view of nodes: nodes send received messages to their respective neighbors, which, in their turn, also forward them.

In this work, we are interested in providing a group communication service which, besides the primitives for joining and leaving the system (Join() and Leave() respectively), offers to the application the primitives CoBroadcast(m), that broadcasts the message m to all nodes, and CoDeliver(m), that delivers m to the application, respecting the causal order of messages. Causal order ensures that sent messages are delivered while respecting the causal relation between them, based on the happened before relation [9] introduced by Leslie Lamport. (see Definition 1 bellow). Therefore, the delivery of received messages might be delayed until they respect causal order. We thus distinguish the reception of a message from its delivery. Note that due to re-transmissions a node might receive multiple times the same message, but the latter is delivered only once.

Definition 1 (Happened before). *The happened before relation, denoted* $\rightarrow$*, partially orders events in a distributed system. Considering two events* e_1 *and* e_2, $e_1 \rightarrow e_2$ *iff: (a)* e_1 *and* e_2 *occurs on the same process and* e_1 *precedes* e_2 *or (b) for a message m* e_1=*send(m) and* e_2=*deliver(m) or (c) there exists an event* e_3 *such that* $e_1 \rightarrow e_3$ *and* $e_3 \rightarrow e_2$ *(transitivity).*

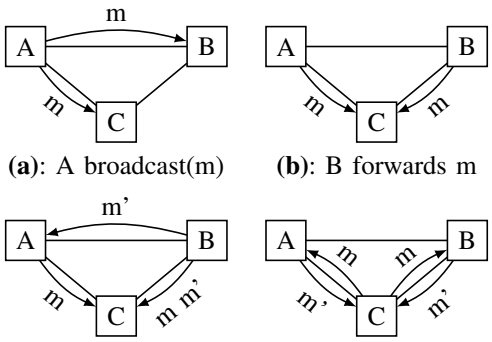
Following (b) and (c) of Definition 1, causal order between two messages is formally defined as: $\forall$m, send(m) $\rightarrow$ send(m$'$) $\Rightarrow$ deliver(m) $\rightarrow$ deliver(m$'$). By extending the above definition to broadcast and deliver of messages, we have: $\forall$m, broadcast(m) $\rightarrow$ boradcast(m$'$) $\Rightarrow$ deliver(m) $\rightarrow$ deliver(m$'$).

We consider a dynamic system in which nodes can join/leave the system during execution. Furthermore, messages delivered by all nodes are discarded, and nodes that join the system will, therefore, never receive these messages. For this reason, we apply the following definition of causal broadcast in our work [11]:

Definition 2 (Causal Broadcast). $\forall$ *messages* m_1, m_2, *broadcast(*m_1*)*$\rightarrow$*broadcast(*m_2*)* $\Rightarrow$ *deliver(*m_2*)*$\not\rightarrow$*deliver(*m_1*)*

With such a definition, if broadcast(m_1) $\rightarrow$ broadcast(m_2) and if m_1 is not available anymore in the system, then m_2 can still be delivered without blocking forever waiting for m_1. However, m_1 is never delivered after m_2.

For implementing causal order of broadcast messages, our algorithm exploits the principle of message forwarding over reliable FIFO links, as proposed in [7]. The scenario of Figure 1 explains such an approach. It consists of three nodes connected by reliable FIFO links. First, node A broadcasts m



(a): A broadcast(m) **(b)**: B forwards m

(c): B send(m') **(d)**: C deliver(m)

Fig. 1: Causal broadcast by FIFO forwarding [7]

(a). Once B received m, B delivers and sends it to C (b). Then B broadcasts m$'$ (c). Finally, C delivers m and sends it to A. (d) shows that m$'$ cannot be received before m by any node.

III. Model

We consider a mobile network composed of Mobiles Hosts (*h* nodes) and Mobile Support Stations (*s* nodes). Nodes communicate exclusively through message passing. Group communication primitives are called by applications running on *h* nodes while *s* nodes deal with message loss and guarantee that messages reach their destination. Every *h* and *s* node is uniquely identified by an *id*.

All *s* nodes are reliable and static, i.e., they do not move, join, or leave the system, neither fail. They are connected by a high speed wired network, whose links are reliable and FIFO, and over which we build a static logical — tree-based — overlay network. They communicate with each other exclusively through this overlay by using the TCP protocol. Every *s* node antenna has the same fixed transmission range, which defines its respective cell to which *h* nodes, close to it, connect themselves. Furthermore, *s* nodes hold most of the consistency and causal order information of the protocol since they have much more memory and computing power than *h* nodes, and no energy limitation.

On the other hand, *h* nodes can move, join, or leave the system, and are subject to temporary failures. The latter happens when a node crashes and then re-joins the system, recovering its last saved state. *h* nodes communicate with the *s* nodes of their respective cells through a wireless network where interferences can lead to message losses, but not message corruption. A *s* node acts as a relay, forwarding the broadcasted messages of the *h* nodes of its cell. Wireless links are not supposed reliable, nor FIFO. Note that a *h* node may be temporarily disconnected from the system if no cell covers its position. Furthermore, an *h* node can be within the transmission range of two *s* nodes simultaneously, but it is connected to at most one *s* node at a given moment, which is generally the closest one. Finally, unlike *s* nodes, *h* nodes have computing and energy limitations, thus maintaining only a small data structure.

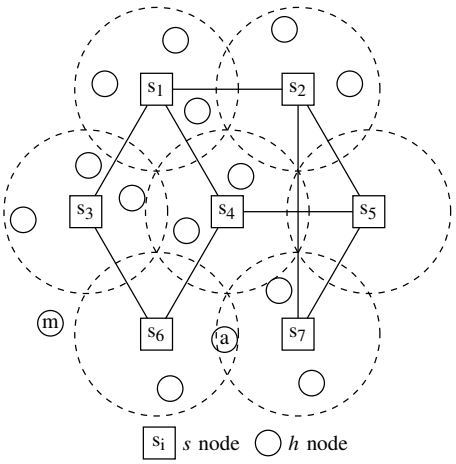


Fig. 2: Network topology

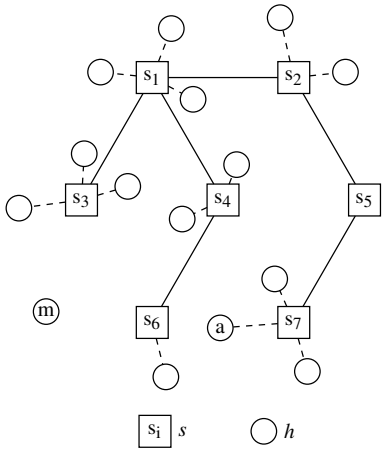


Fig. 3: Tree topology

We should point out that cells must overlap in order to ensure the covering of the whole area of the system, as shown in Figure 2.

Figure 2 shows an example of a network topology. Cells are represented by dashed circles. It is worth remarking that node *m* is within no cell, and that node *a* is within both s_6's and s_7's cells.

The same network of Figure 2 is represented in Figure 3 but *s* and *h* nodes are logically organized in a tree-based overlay. The wired and wireless networks are respectively represented by solid and dashed lines. Some wired links from Figure 2 have been removed, and *h* nodes are connected to at most one *s* node (e.g., *a*). *h* nodes are leaves of the tree since they only communicate with their respective *s* node. The latter can also be a leaf, provided that no *h* node is connected to it. Note that *m* that is within no cell, is temporarily disconnected from the tree.

IV. Causal broadcast algorithm

Our causal broadcast algorithm, presented in Algorithm 1 to 4, consists of three parts: the dissemination of application messages, the handling of join/leave, and mobility operations of *h* nodes.

Each line is preceeded by a symbol (*,#, or +), corresponding to the part of the algorithm to which the line is related. Lines preceeded by * are those related to the dissemination of application messages, those preceeded by # are those related to the join and leave of *h* nodes, and those preceeded by + are those related to the mobility of *h* nodes.

Algorithm 1 presents the tasks executed by *h* nodes. Algorithm 2 to 4 present the tasks executed by *s* nodes. Algorithm 2 handles the reception of messages sent by *h* nodes, Algorithm 4 handles the reception of messages sent by *s* nodes, and Algorithm 3 the periodical sending of *ack* messages by *s* nodes. An application running on a *h* node can call the following four functions provided by the algorithm:

- **Join()**: whenever the *h* node wants to join the system.
- **Leave()**: whenever the *h* node wants to leave the system.
- **CoBroadcast**(m): for broadcasting the message m.
- **CoDeliver()**: delivering a message, if available.

A. Data structures and message types

Specific structures are kept by *h* and *s* nodes to guarantee the causal delivery of application messages.

First, *h* nodes maintain variables to identify messages, to manage application messages, and to manage the connection with their cell's *s* node.

A *h* node piggybacks its id, id_{h}, on sent messages to identify them. It also checks that received messages come from its cell by comparing their attached id to the one of its cell, id_{C}, since *h* nodes may be in reach of several cells.

Moreover, a *h* node maintains some variables to manage application messages: Two sequence number counters, seq_{h} and seq_{C}, the first to stamp new broadcasted application messages, the latter to contain the sequence number of the next application message to deliver. Additionnaly, two buffers, *SBuffer* and *RBuffer*, the first stores unacknowledged sent application messages, the second contains received application messages until they are FIFO ordered.

Finally, a *h* node uses some variables to manage its connection: Two session number counters are used to identify connections, *Ses* and Ses_{LC}, the first contains the current session number, the latter the session number of the latest session in which the connection was established, i.e., where the node received a reply from the *s* node to which it tried to connect, confirming the reception of the connection request. A variable *state* is also used to identify the node's current connection state. A node can be in four states: *init, join, conn, estab*.

Secondly, *s* nodes also maintain variables to manage their messages, as well as a structure for each connected *h* node h_{i}.

A *s* node maintains its cell's id, id_{C}, a sequence number counter for new broadcasted messages seq_{C}, and a buffer which stores unacknowledged broadcasted application messages, denoted *SBuffer*.

On the one hand, the structure associated to h_{i} contains some variables to manage the connection. The structure associated to h_{i} is identified with h_{i}'s id, id_{h}. Two sequence number

counters are used, seq_{h} and seq_{ACK}, the first contains sequence number of the next message of h_{i} to re-broadcast, the second the sequence number of the most recent application message h_{i} acknowledged. The session number of the connection with h_{i} is stored in *Ses*. A buffer, *RBuffer*, contains received application messages of h_{i} until they are FIFO ordered.

On the other hand, some variables of the structure are only used during the handoff process. H_{lock} locks the structure if a handoff is in progress, to ensure that handoffs concerning h_{i} are done sequentially. seq_{Co} saves the state of *SBuffer*. m_{nd} contains messages discarded by the *s* node which h_{i} did not delivered at the previous *s* node to which it was connected. *CoRequest* stores the the most recent pending Req_1 request message received during the current handoff.

Messages are divided into three groups: the first handles the dissemination and acknowledgment of application messages, the second the join/leave of *h* nodes, and the third the mobility of *h* nodes.

The first group contains:

- Application messages from node type A to B: <**App**$_{\mathrm{h\_s}}$,data,$\mathrm{id_h}$,$\mathrm{seq_h}$>, <**App**$_{\mathrm{s\_s}}$,data,$\mathrm{id_h}$,$\mathrm{seq_h}$> and <**App**$_{\mathrm{s\_h}}$,data,$\mathrm{seq_C}$,$\mathrm{id_C}$,$\mathrm{M_d}$>
- Acknowledge messages of *h* nodes <**ack**$_{\mathrm{h}}$,$\mathrm{id_h}$,$\mathrm{seq_C}$,Ses> and of *s* nodes <**ack**$_{\mathrm{C}}$,$\mathrm{id_C}$,vSeq>.

The second group contains:

- <**join**,$\mathrm{id_h}$,Ses> sent by *h* nodes to connect to the system.
- <**init**$_{\mathbf{ACK}}$,$\mathrm{id_h}$,$\mathrm{seq_h}$,$\mathrm{seq_C}$,Ses>: sent by *s* nodes to conclude the connection phase.
- <**leave**,$\mathrm{id_h}$,Ses> sent by *h* nodes to leave the system.
- <**leave**$_{\mathbf{ACK}}$,$\mathrm{id_h}$,Ses> acknowledge the reception of *leave*.
- <**Delete**,$\mathrm{id_h}$,Ses> sent to *s* nodes to delete the *h* node $\mathrm{id_h}$ if registered.

The third group contains:

- <**init**,$\mathrm{id_h}$,$\mathrm{seq_C}$,Ses,$\mathrm{Ses_{LC}}$> sent by *h* nodes to change cell.
- Messages exchanged between *s* nodes during handoffs to ensure causal order for moving *h* nodes: <**Req$_1$**,$\mathrm{id_h}$,$\mathrm{seq_C}$,$\mathrm{Ses_{LC}}$,Ses>, <**Rsp$_1$**,$\mathrm{id_h}$,$\mathrm{seq_C}$,$\mathrm{m_{nd}}$,Ses>, <**Req$_2$**,$\mathrm{id_h}$,$\mathrm{msg_{req}}$,Ses>, <**Rsp$_2$**,$\mathrm{id_h}$,msg,$\mathrm{msg_{rcv}}$,Ses>
- <**App$_{\mathbf{Co}}$**,data,$\mathrm{id_C}$,$\mathrm{seq_C}$,$\mathrm{id_{h_{dest}}}$,Ses> application messages sent during the connection phase to a specific *h* node.

We define several functions to make the algorithm more easily readable. Messages are sent with the **broadcast(type,...)** function, whose behavior and arguments change in function of the message *type*.

h nodes use the **broadcast** function to sent messages (**App**$_{\mathrm{h\_s}}$, **ack**$_{\mathrm{h}}$, **join**, **leave**, **init**) on the wireless network to their cell's *s* node.

When used by *s* nodes, the behaviour of the **broadcast** function changes according to the message type:

- **App**$_{\mathrm{S\_h}}$, **ack**$_{\mathrm{C}}$, **init**$_{\mathrm{ACK}}$, **App**$_{\mathrm{Co}}$, **leave** and **leave**$_{\mathrm{ACK}}$ messages are sent on the wireless network.
- **App**$_{\mathrm{S\_S}}$ messages are forwarded on the wireless network and to the neighbor *s* nodes, except the one which sent them.
- **Req$_1$**, **Rsp$_1$**, **Req$_2$**, **Rsp$_2$** messages are forwarded if the *s* node is not the destination of them. In this case, they are forwarded to the neighbor *s* nodes, except the one which sent them.
- **Delete** messages are forwarded to the neighbor *s* nodes, except the one which sent them.

Moreover, we define some other functions. **chooseC()** returns the position of the nearest *s* node, and **minSeq()** returns the sequence number of the oldest message of *SBuffer* (or $\mathrm{seq_C}$ if *SBuffer* is empty).

B. Dissemination of application messages

Similarly to [7] and [13], the dissemination mechanism is based on flooding over an overlay network. Nodes are logically organized in a tree, like the one of Figure 3.

A *h* node calls **CoBroadcast(m)** in order to broadcast an application message *m*. All *h* nodes of the system should deliver *m*, respecting causal order of messages. On the other hand, *s* nodes are responsible for the dissemination of application messages. A *s* node re-broadcasts to the *h* nodes of its cell every application message it receives from a *h* node of its cell. It also sends the message to its *s* node neighbors of the overlay through the wired network. An application message received by a *s* node, sent by a second *s* node, is forwarded in the same way, except that it is not sent back to the sender.

A *h* node includes in every application message it broadcasts both its *id* and the *id* of the current cell to which it is connected. Since wireless links are neither FIFO nor reliable, our protocol needs to detect out of order messages as well as losses. To this end, a *h* node associates a sequence number value *seqh* to every new message it broadcasts, by keeping a local sequence number counter variable which is incremented at every new broadcast. A *s* node also has its own sequence number counter variable used to timestamp (*seqs*) every new message it re-broadcasts. It increments the counter at every re-broadcast of a different message and controls, for each connected *h* node $\mathrm{h_i}$, which is the *seqs* of the last message that $\mathrm{h_i}$ delivered.

Both *h* and *s* nodes maintain two types of local buffers: (1) *RBuffer*, which stores received application messages until they are FIFO ordered and thus delivered (2) *SBuffer* which keeps *pending* messages sent over wireless networks, i.e., those messages that have not been acknowledged yet by all the receivers.

A *s* node keeps one *RBuffer* per connected *h* node. It uses the *seqh* value included in application messages sent by a given *h* node, h_{i}, to detect out of order message receptions: if a message *m* with $seqh_{\mathrm{m}}$ sent by h_{i} is received by s_{i} but the latter has not received yet all messages from h_{i} whose *seqh* value is smaller than $seqh_{\mathrm{m}}$, then *m* is inserted in the *RBuffer* that s_{i} associates to h_{i}. Similarly, h_{i} uses the sequence number values (*seqs*) of the messages received from s_{i} to order them, i.e., h_{i} delivers them in ascending order of *seqs*, temporally keeping those which are out of order messages in its *RBuffer*.

Regarding *SBuffer*, a message broadcasted by a *h* node is considered *pending* by this *h* node till it receives an acknowledge from its respective cell's *s* node while a message

Algorithm 1: Tasks of h_i

Join

1# $seq_{h_i}=seq_{C_i}=Ses_i=Ses_{LC_i}=0$
2# $SBuffer_i=RBuffer_i=\emptyset$
3# id_{C_i}=chooseC()
4# $state_i$=join
5# broadcast(<join,id_{h_i},0,0,id_{C_i},0>)

Upon changing cell

6+ id_{C_i}=chooseC()
7+ $state_i$=($state_i$==join ? join : init)
8+ Ses_i++
9+ broadcast(<$state_i$,id_{h_i},seq_{C_i},Ses_i,id_{C_i},Ses_{LC_i}>)

Upon calling CoBroadcast(m)

10* msg=<App_{h_S},m,id_{h_i},seq_{h_i}++>
11* broadcast(msg)
12* $SBuffer_i$.insert(msg)

Leave

13# broadcast(<leave,id_{h_i},Ses_i>)

upon reception of m=<type,...> **from a *s* node**

14* **switch** (m)
15* **case** <App_{S_h},data,seq_C,id_C,M_d> **:**
16* **if** seq_C>seq_{C_i} **then**
17* $RBuffer_i$.insert(m)
18* **else if** seq_C==seq_{C_i} **then**
19* seq_{C_i}++
20* **if** $id_{h_i} \notin M_d$ **then**
21* deliver(d)
22* FIFOdeliver()
23+ **case** <App_{Co},data,id_C,seq_C,$id_{h_{dest}}$,Ses>**:**
24+ **if** $id_{h_{dest}}$==id_{h_i}∧Ses==Ses_i **then**
25+ **if** $state_i$==init **then**
26+ seq_{C_i}=0 ; Ses_{LC_i}=Ses ; $state_i$=connecting
27+ **if** seq_C>seq_{C_i} **then**
28+ $RBuffer_i$.insert(m)
29+ **else if** seq_C==seq_{C_i} **then**
30+ seq_{C_i}++
31+ deliver(d)
32+ FIFOdeliver()
33* **case** <ack_C,vSeq>**:**
34* $SBuffer_i$\ = {∀m′ ∈$SBuffer_i$,m'$_{seq}$<vSeq[id_i]}
35* seq_{C_i}=vSeq[id_i]
36# **case** <$init_{ACK}$,id_h,seq_h,seq_C,Ses>**:**
37# **if** id_{h_i}==id_h∧$state_i$≠estab∧Ses_i==Ses **then**
38# **if** !ackTimeout **then**
39# StartAckTimeout()
40# **if** initTimeout **then**
41# stop initTimeout
42# seq_{C_i}=seq_C ; Ses_{LC_i}=Ses ; $state_i$=estab
43+ clear($SBuffer_i$,seq_h)
44+ FIFODeliver()
45# **case** <$leave_{ACK}$,id_h>**:**
46# **if** id_{h_i}==id_h **then**
47# leave()

upon expiration of timeout of M at a *h* node

48* **if** M==ackTimeout **then**
49* broadcast(<ack_h,id_{h_i},seq_{C_i},Ses_i>)
50* **else**
51* broadcast(<M>)
52* setTimer(M,calcTimeout())

Algorithm 2: upon reception of m=<type,...> from h_i at s_j

1* **switch** (m)
2* **case** <App_{h_S},data,id_h,seq_h>**:**
3* **if** $seq_{h_{j,i}}$ == seq_h **then**
4* Disseminate(data,id_h,seq_h)
5* $seq_{h_{j,i}}$++
6* FIFODisseminate()
7* **else if** $seq_{h_{j,i}}$ < seq_h **then**
8* $RBuffer_{j,i}$.insert(m)
9* **case** <ack_h,id_h,seq_C,Ses>**:**
10# **if** h[id_h]∧Ses==$Ses_{j,i}$ **then**
11* $seq_{C_{j,i}}$=seq_C
12* clear($SBuffer_j$,seq_C)
13+ **if** seq_C < minSeq() **then**
14+ clear($m_{nd_{j,i}}$)
15+ **if** $m_{nd_{j,i}}$==∅ **then**
16+ $seq_{C_{j,i}}$=calcSeq(id_h)
17+ broadcast(<$init_{ACK}$,id_h,$seq_{C_{j,i}}$,Ses>)
18# **case** <join,id_h,Ses>∨<init,id_h,seq_C,Ses,Ses_{LC}> **:**
19# **if** h[id_h] **then**
20+ **if** $H_{lock_{j,i}}$ **then**
21+ **if** Ses≠$Ses_{j,i}$ **then**
22+ **if** Ses_{LC}==$Ses_{j,i}$ **then**
23+ clear($m_{nd_{j,i}}$,seq_C)
24+ $seq_{C_{j,i}}$=seq_C
25+ $Ses_{j,i}$=Ses
26+ update($m_{nd_{j,i}}$,Ses,seq_C)
27+ **if** $m_{nd_{j,i}}$ == ∅ **then**
28+ $seq_{C_{j,i}}$=calcSeq(id_h,seq_C)
29+ **if** $m_{nd_{j,i}}$ == ∅ **then**
30# broadcast(<$init_{ACK}$,id_h,$seq_{C_{j,i}}$,$Ses_{j,i}$>)
31# **else**
32# h={id_h,0,minSeq(),Ses,∅,false,0,∅}
33# **if** type == join **then**
34# $H_{lock_{j,i}}$=true
35# broadcast(<$init_{ACK}$,id_h,minSeq(),Ses>)
36# broadcast(<Delete,id_h,Ses>)
37# **else**
38+ broadcast(<Req_1,id_h,seq_C,Ses_{LC}>,Ses)
39# **case** <leave,id_h,Ses>**:**
40# **if** h[id_h] **then**
41# delete(h[id_h])
42# broadcast(<Delete,id_h,Ses>)
43# broadcast(<$leave_{ACK}$,id_h>)

Function: Disseminate(data,id_h,seq_h)

44* broadcast(<App_{S_h},data,seq_{C_j},id_{C_j},∅>)
45* $SBuffer_i$.insert(<App_{S_h},data,seq_{C_j},id_{C_j},∅,id_h,seq_h>)
46* broadcast(App_{S_S},data,id_h,seq_h)
47* seq_j++

Algorithm 3: upon expiration of ackTimeout at *s* nodes

```
48: vSeq={seq_i,∀ connected h_i}
49: broadcast(ack_S, vSeq)
50: setTimer(ackTimeout,calcAckTimeout())
```

re-broadcasted by a *s* node remains *pending* till it receives acknowledges (ack messages) from all connected *h* nodes of the cell. In both cases, as soon as a message is not *pending* anymore, it is removed from the *SBuffer*. On the other hand, every *pending* message is periodically retransmitted within a time interval whose duration, recalculated at each retransmission, depends on the number of pending messages in the *SBuffer*.

A *s* (resp., *h*) node regularly sends an ack message (timeout mechanism), confirming the reception (resp., delivery) of those application messages whose sequence number value is smaller or equal to the one included in the *ack* message in question. A final remark is that a *h* node delivers a message it has broadcasted only after receiving this same message from its cell's *s* node.

Figure 4 shows the broadcast of two messages: h_1 broadcasts m_1 and h_2 broadcasts m_2 after delivering m_1 (broadcast(m_1) $\rightarrow$ broadcast(m_2)). The notation of the messages also includes their sequence number. Pending messages in *SBuffer* (bold) and non FIFO ordered ones in *Rbuffer* (italic) are also shown. h_1 is connected to s_1 and h_2 to s_2 while s_1 and s_2 are neighbors.

Upon reception of m_1, s_1 sends it to s_2 and also broadcasts it within its cell, which contains h_1. When receiving m_1, s_2 broadcasts it in its cells. Note that s_2 does not send m_1 back to s_1. Node h_2 receives and delivers m_1. Then, h_2 broadcasts m_2, which is disseminated like m_1. Remark that h_2 delivers m_2 only after receiving m_2 from s_2, confirming the reception of m_2. At expiration of a timeout, s_2 (resp., h_2) sends an ack message to h_2 (resp., s_2) with seq=1 (resp., seq=2) to acknowledge m_2 (resp., m_1 and m_2). h_2 and s_2 then stop sending m_2 and clear their respective *SBuffer*.

Node s_1 re-broadcasts m_1 within its cell after receiving it, but it is lost. Thus, at the next timeout expiration, h_1 re-broadcasts m_1, because s_1 did not acknowledged it. However, s_1 has received m_1 and, therefore, ignores m_1's second reception. Upon receiving m_2 from s_2, s_1 broadcasts it within its cell. On the other hand, for h_1, m_2 does not respect FIFO order, because it awaits a message with $seqs_1 = 1$ and m_2 has $seqs_1 = 2$. Hence, h_1 stores m_2 in its *RBuffer*. At expiration of the timeout related to m_1, s_1 broadcasts it again, and, upon reception, h_1 delivers the two messages in FIFO order and remove them from its *Rbuffer*. Finally, h_1 and s_1 send *ack* messages at their next timeout expiration, which are both received. Hence, they remove both messages from their respecting *SBuffer*. Note that all copies of m_1 and m_2 have been deleted from all node buffers.

Algorithm 4: upon reception of m from *s* nodes at s_j

```
51* switch (m)
52*   case <App_S_S,m,id_h,seq_h>:
53*     Disseminate(m,id_h,seq_h)
54+   case <Req_1,id_h,seq_C,Ses_LC>:
55+     if Ses_j,i<Ses∧!H_lock_j,i then
56+       if CoRequest_j,i.Ses<Ses then
57+         CoRequest_j,i=m
58+     else
59+       if Ses_j,i>Ses then
60+         broadcast(<Delete,id_h,Ses>)
61+         return
62+       seq_Co_j,i=seq_C_j
63+       if Ses_LC == Ses_i then
64+         m_nd={id_m_k,∀m_k∈SBuffer∪m_nd_j,i∧
              id_m_k∉M_m_k∧seq_m_k>seq_C}
65+       else
66+         m_nd={id_m_k,∀m_k∈SBuffer∪m_nd_j,i∧id_m_k∉M_m_k}
67+       broadcast(<Rsp_1,id_h,seq_h_j,i,m_nd,Ses>)
68+   case <Rsp_1,id_h,seq_h,m_nd,Ses>:
69+     if Ses≠Ses_j,i then
70+       broadcast(m)
71+     seq_h_j,i=seq_h
72+     seq_Co_j,i=seq_C
73+     m_nd_j,i=m_nd
74+     msg_req={m∈m_nd,m∉SBuffer_j}
75+     broadcast(<Req_2,id_h,msg_req,Ses>)
76+   case <Req_2,id_h,msg_req,Ses>:
77+     if Ses≠Ses_j,i then
78+       broadcast(m)
79+     msg={m'∈m_nd∪SBuffer_j,m'∈msg_req}
80+     m_rcv={m'∈SBuffer_j,m'_seq>seq_Co_j,i}
81+     delete(h[id_h])
82+     clear(SBuffer_j)
83+     broadcast(<Rsp_2,id_h,msg,m_rcv,Ses>)
84+   case <Rsp_2,id_h,msg,msg_rcv,Ses>:
85+     if Ses≠Ses_j,i then
86+       broadcast(m)
87+     ∀m'∈SBuffer_j\{msg_rcv∪m_nd_j,i}, m'_seq<seq_Co_j,i,
          m'_M_d∪={id_h}
88+     H_lock_j,i=true
89+     BroadcastCo(msg)
90+     if CoRequest_j,i then
91+       Receive(CoRequest_j,i)
92+     if msg==∅ then
93+       seq_C_j,i=calcSeq(id_h)
94+       broadcast(<init_ACK,id_h,seq_C_j,i,Ses_j,i>)
95#   case <Delete,id_h,Ses>:
96#     if h[id_h] then
97#       if Ses>Ses_j,i then
98#         if CoRequest_j,i then
99#           broadcast<CoRequest_j,i>
100#         delete(h[id_h])
101#     broadcast(<Delete,id_h,Ses>)
```

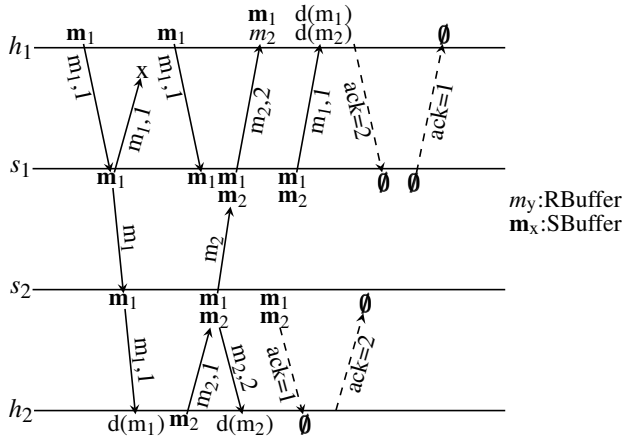


Fig. 4: Broadcast of m_1 and m_2

C. Join/leave the system

An extra control is necessary to identify the connection in which *h* nodes are, because each *s* node has its own local sequence number (*seqs*), messages may be lost, and *h* nodes can move and change cells.

To solve this problem, we have introduced the concept of *session* sequence number that uniquely identifies a wireless network connection between a *h* node and a cell's *s* node. For this purpose, every *h* node keeps the following variables whose values are also included in some messages of the protocol, whenever necessary:

- *Ses* identifies h_i's current connection. It is incremented when h_i changes its current cell.
- Ses_{LC} identifies the last established connection.

The **Join()** primitive, called by h_i, chooses a *s* node s_j and sends it a *join* message which includes the *id* and *Ses* of h_i. In its turn, if h_i is not already connected to it, node s_j associates a structure to h_i to control the connection. Otherwise, h_i's session number is updated in order to take into account that h_i might have tried to connect to another *s* node without succeeding. In both cases, s_j sends an $init_{ACK}$ message to h_i, in order to inform it about the sequence number of the oldest stored message in its *SBuffer*. Moreover, a *Delete* message, including the *join*'s session number of h_i, is disseminated to the other *s* nodes so that they delete the structure they might have associated with h_i if they store a session number which is lower than the one of the *Delete* message, i.e., if *Delete* concerns a more recent connection. We should emphasize that a *h* node can join the system at any moment, but it will not deliver those messages which were discarded by the *s* node to which it connects before its connection.

h_i can leave the system at any time by calling the **Leave()** primitive which will re-broadcasts a *leave* message until it is received by some *s* node. The latter forwards the *leave* message to the other *s* nodes. Those that have associated a structure to h_i will delete it at reception of *Delete*. h_i leaves the system once receiving an acknowledge to its *Leave* request.

D. Handoff procedure

The handoff procedure ensures the causal order delivery of application messages when *h* nodes move between cells after having joined the system. We denote s_p and s_n the previous and the new cell *s* node of h_i respectively. Basically, the handoff procedure consists of a set of messages exchanged between the moving node h_i and s_n, as well as between the latter and s_p, as shown in Figure 5. The handoff procedure must cope with the two following constraints.

a) Single delivery: *s* nodes do not necessarily assign sequence numbers in the same order to those application messages which are not causally related, i.e., which have been concurrently broadcasted. For example, let's suppose that s_p receives *m* then *m'* while s_n receives *m'* then *m* and that h_i delivers *m* when connected to s_p, moves, connects to s_n, and then delivers *m'*. Without any extra control, h_i would deliver *m* again, since s_n has ordered *m* after *m'*. In order to avoid these multiple deliveries, a *s* node assigns a small set, denoted M_d, to every application message it broadcasts. The M_d of a message contains the *id*s of all *h* nodes of the cell that have already delivered the message in another cell. By exchanging messages with s_p, s_n acquires knowledge about which are the pending messages of its *SBuffer* which h_i have already delivered, including then h_i's id to the M_d of each of these messages. h_i will not deliver those messages whose M_d set contains its *id*. For every one of these messages, it just updates the related sequence number (*seqs*). Note that the size of M_d is quite small as, at a given moment, few *h* nodes are changing cells.

On the other hand, a *s* node discards a message m, i.e., removes it from its *SBuffer*, once all connected *h* nodes have acknowledged *m*. This message deletion procedure renders more difficult the comparison of the *SBuffer* of s_n and s_p to find which messages h_i has not delivered, since the *SBuffer* of one *s* node may contain messages removed from the other one, and messages received by one may not have been received yet by the other. Such a conflict is handled by message exchanges over the wired network connecting the *s* nodes.

b) Session consistency: Messages may be lost on the wireless network. In this case, nodes cannot determine if their handoff messages are received nor to which connection received messages belong. This uncertainty is handled by including the *h* node's session number *Ses* in every handoff message, identifying, therefore, the connection in which it is sent. Moreover, a *h* node is unable to know which *s* node holds its latest connection information, because its connection requests may be lost. In order to tackle this problem, handoff messages are propagated over the wired network to all *s* nodes. A third remark is that due to its movement, a *h* node may try to connect to different *s* nodes in a short time interval, starting, therefore, several handoff procedures simultaneously. It happens, for instance, if h_i tries to connect to a second *s* node just before connecting to s_n. Hence, during the handoff procedure with h_i, s_n may receive old connection requests from h_i. The *s* nodes manage these concurrent requests sequentially in increasing order of *Ses*.

Handoff principle: h_i starts the handoff procedure when it moves to s_n's cell, by sending to s_n an *init* message which contains the sequence number (*seqs*) of the latest message it delivered, as well as the request's session number *Ses* and the session number of the last established connection Ses_{LC}.

Upon reception of h_i's *init* message, if h_i is already registered in s_n and no handoff procedure is in progress for h_i, then s_n updates the stored session number *Ses* value it associated to h_i, so that messages related to h_i's previous connection attempts to other *s* nodes, i.e., those with lower *Ses* values, will be discarded. Furthermore, if the last established connection of h_i (Ses_{LC}) was with s_n itself, it considers the *seqs* value of the *init* message as an acknowledge (*ack* message) because, in this case, h_i may have received and delivered messages during that Ses_{LC} connection. On the other hand, if h_i is not already connected to s_n, then the latter disseminates the handoff request Req_1 message including *seqs* and Ses_{LC} to its *s* node neighbors in the tree overlay that will forward it to their neighbors and so on.

As previously explained, only one handoff procedure per node is executed at a given time, following the *Ses* value of Req_1, even if h_i has tried to connect to several *s* nodes in a short time interval. Req_1 messages related to more recent connections of h_i, and received during the execution of another handoff procedure, are handled only after the latter ends. s_n discards every Req_1 message concerning an older connection than the current one and propagates a *Delete* message containing the request's *Ses* value, so that the *s* node which sent the Req_1 in question deletes the structure it associated with h_i. If no handoff procedure is currently in progress and Req_1's *Ses* value is higher than the one s_p stores, then s_p replies to Req_1 with Rsp_1 which contains the list of id's of the messages which h_i has not delivered. Moreover, if the last established connection (Ses_{LC}) of h_i was with s_p, then s_p considers Req_1's sequence number as an acknowledgment.

When receiving Rsp_1, s_n sends a Req_2 to s_p, asking for messages it deleted among those of Req_1's list. s_p replies with a Rsp_2 that includes two lists: (1) the list of requested messages that s_n has discarded; (2) the list of messages that s_p has received since the sending of *(Rsp_1)*. Finally, s_p deletes the structure associated to h_i and removes those messages from *SBuffer* whose only missing acknowledge was the one from h_i.

The handoff protocol ensures that h_i delivers all messages exactly once (at least once and at most once), respecting the causal order of them.

At least once: s_n keeps all messages received after the reception of the *init* message and sends them to h_i. Moreover, due to the FIFO channels between *s* nodes, the messages received by s_n before *init* are received by s_p before it receives Req_1. Thus, h_i always delivers these messages, since s_n requests them to s_p in Req_2, if it has discarded them.

At most once: h_i might deliver messages from s_p until it tries to connect to s_n by sending an *init* message. Since the wired network is FIFO, and s_n sends Req_1 after receiving *init*, the messages that h_i might have already delivered are those s_n receives before the reception of Rsp_1. Among these messages which are still in its *SBuffer*, s_n determines which ones have not been delivered yet by h_i: (1) messages identified by s_p in the Rsp_1 message (m_1 in Figure 5); (2) messages that s_p received between the sending of Rsp_1 and reception of Req_2 (m_3 in Figure 5). Due to FIFO channels, s_p has received these messages before the reception of Req_2 and includes them in the Rsp_2 message. Undelivered messages are then identified by s_n that also adds the id of h_i to the M_d of all the other messages received before Rsp_1 since they were delivered.

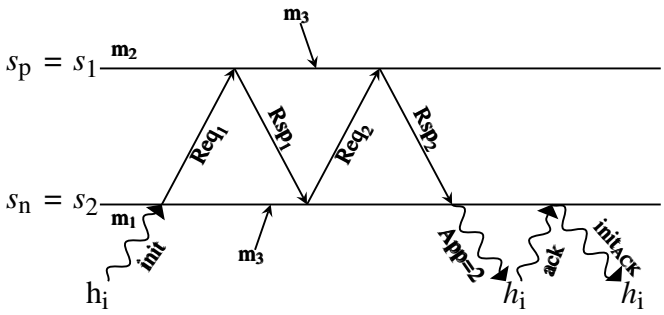


Fig. 5: Handoff procedure

s_n and h_i exchange messages to conclude the handoff procedure. h_i must first deliver the messages it has not delivered but which s_n has discarded (m_2 in Figure 5), before delivering s_n's pending messages (m_3 in Figure 5). These messages have a sequence number and are thus delivered respecting causal order. They contain h_i's id to identify the handoff. s_n concludes the handoff by sending an $init_{ACK}$ message to h_i once the latter confirms the delivery of the messages s_n have deleted. It assigns to h_i the sequence number (seqs) of the oldest pending message of its *SBuffer* which was not delivered by h_i. Finally, h_i updates its sequence (seqs) and session (Ses_{LC}) numbers, and removes from its *SBuffer* the messages that $init_{ACK}$ have acknowledged.

Handoff example: Figure 5 shows a simple example of a handoff procedure when h_i moves from cell 1 to cell 2 but that covers the general case. Three messages m_1, m_2, and m_3, are broadcasted: broadcast(m_1) is concurrent to broadcast(m_2) and broadcast(m_2) $\rightarrow$ broadcast(m_3). h_i has already delivered m_1. We consider that s_1 has discarded m_1 and stores m_2, while s_2 has discarded m_2 and stores m_1 ($SBuffer_1 = \{m_2\}$ and $SBuffer_2 = \{m_1\}$). Both *s* nodes receive m_3 during the handoff.

The handoff procedure starts when h_i sends an *init* message to s_2. The *init* message contains seq=1, since h_i delivered m_1. We assume that no other handoff takes place for h_i simultaneously. Upon reception of *init*, s_2 sends Req_1 message to s_1 with *seqs*=1 and Ses_{LC} which contains the session number of the connection with s_1.

When receiving Req_1, s_1 learns that h_i has not delivered m_2 since *seqs*=1. It thus replies with $Rsp_1 = \{\{id(m_2)\}, seq_h = 0\}$, where seq_h=0 because h_i did not broadcast any message.

Upon reception of Rsp_1, s_2 requests m_2 (Req_2) since it discarded m_2. s_1 replies with Rsp_2 message which contains the list of requested messages ($\{m_2\}$) and the list of received

messages since s_1 sent Rsp_1 ({id(m_3)}). Moreover, s_1 deletes the structure associated to h_i.

Based on Rsp_2, s_2 can determine which messages of its *SBuffer*={m_1, m_3} h_i has delivered: s_2 received m_1 before Rsp_2 and m_1 is not identified by s_1 as not delivered by h_i. Therefore, h_i already delivered m_1 and s_2 adds h_i's id to m_1's M_d. s_1 received m_3 between the send of Rsp_1 and the send of Rsp_2. Hence, h_i did not deliver m_2 and must deliver it before delivering s_2's pending messages. Thus, s_2 sends m_2 with seq=0 to h_i and discards m_2 once h_i acknowledged it.

Finally, s_2 assigns to h_i *seqs*=3, since h_i delivered m_1 and m_2, and sends an $init_{ACK}$ message to h_i, concluding the connection process. When receiving $init_{ACK}$, h_i sets *seqs*=3. Hence, m_3 is the next message h_i will deliver.

E. Fault resilience

h nodes are subject to transient faults. For recovery sake, *h* nodes save the following variables on persistent local storage: the sequence number of broadcasted and received messages, both session numbers, and the *SBuffer*. Note that our experiments show that *h* nodes' *SBuffer* are of small size. The *SBuffer* and its associated sequence number are saved whenever the *h* node broadcasts a message, while the sequence number of received messages is saved when it sends *ack* messages. Finally, *Ses* is saved each time it is incremented and Ses_{LC} whenever a new session connection is confirmed (reception of $init_{ACK}$ message). Upon recovering, a *h* node restores these variables and sends an *init* message, similarly to when it changes cells. Therefore, *h* node transient faults are tolerated with few persistent information.

On the other hand, permanent failures are not tolerated. A *s* node keeps a pending message in its *SBuffer* until all *h* nodes connected to its cell acknowledged it, move to another cell, or leave the system properly. Hence, *s* nodes would never discard unacknowledged messages in the presence of permanent failures. The memory footprint of *s* nodes would then grow infinitely, and the wireless network would rapidly be overloaded. In fact, all these unacknowledged messages would be periodically broadcasted on the cell's wireless network, increasing message loss rate until all messages on the wireless network would be lost due to interferences. Based on this observation, we point out that transient failures cannot last too long. However, this limitation is inherent to wireless networks' nature. No algorithm can safely discard messages and cope with interferences in such conditions. Nevertheless, transient failures of long duration, as well as permanent failures, could be handled with the assumption that a *h* node recovers and reconnects to the *s* node to which it was connected before failing within at least T seconds. In this case, a *s* node would delete the information it stores about a failed *h* node if no message is received from it during T seconds, considering this *h* node as a new one when it recovers.

V. Performance Evaluation

We have conducted experiments on OMNeT++, with the INET extension [18]. INET renders simulations more realistic by implementing communication layers (e.g., TCP/UDP/Ethernet/IPv4/MAC), node mobility, propagation delays, and wireless networks with interferences.

The wireless network's antennas have a communication range of 120m. The wired network has a bandwidth of 10Mb/s and a delay of 10ms between each link. The network contains 7 *s* nodes, configured as in Figure 2. Initially, 70 *h* nodes are placed randomly, connected to the closest *s* node.

Application messages have a fixed size of 100 bytes and are encapsulated into UDP/TCP/IPv4/MAC packets. Our algorithm uses TCP only on the wired network while UDP on the wireless network.

Experiments were executed several times, and the initial position of the *h* nodes is set randomly at each run. In the first experiment, we compare our algorithm with a TCP flooding one, considering that *h* nodes are static. We then evaluate our algorithm in a dynamic context where *h* nodes move but do not fail. We then extend this experiment where *h* nodes can fail and recover (transient failures). Finally, we analyze the memory footprint.

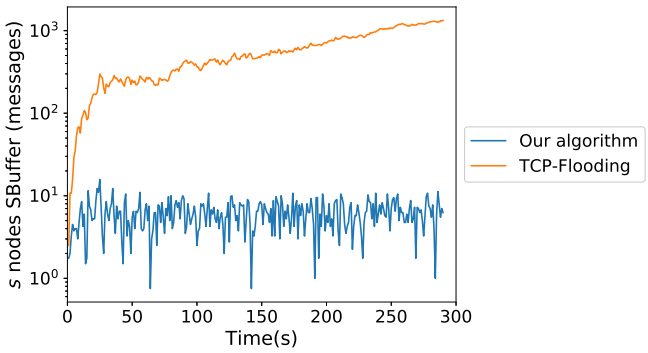
A. Static system

Figure 6 compares our algorithm with [13] (denoted *TCP-Flooding*), which is based on reliable FIFO channels implemented with TCP connections.

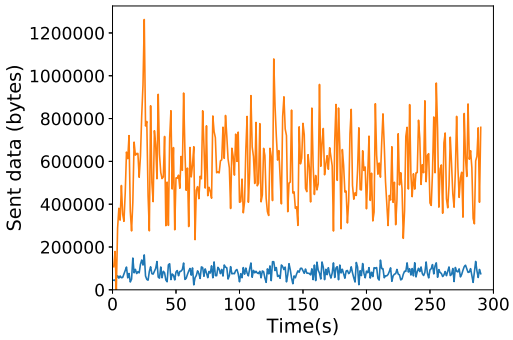
Each *h* node broadcasts, on average, an application message every 12.5 seconds. Since there are 70 *h* nodes, there are 70/12.5≈5.6 messages broadcasts per second. In *TCP-Flooding*, a *s* node disseminates an application message by sending it point-to-point to each of its connected *h* nodes. However, some application messages might be acknowledged right after reception, and the respective *ack* messages will then collide on the wireless network with the application messages sent to the other *h* nodes. Hence, in order to reduce these collisions, we have included a 5ms delay between every point-to-point sending of a given application message by a *s* node to each of its connected *h* nodes.

Figure 6a gives the average number of messages stored in *s* nodes' *SBuffer*. We use a logarithmic scale in the case of Figure 6a, due to the different order of magnitudes of *SBuffer* sizes. Our algorithm's *h* nodes' *RBuffer* are much smaller (0.5<) than TCP-Flooding's (<5). The size of *SBuffer* of *h* nodes and the *RBuffer* of *s* nodes are small, because *h* nodes only disseminate one message every 12.5 seconds. Therefore, we do not discuss them. On the other hand, the number of messages in TCP-Flooding's buffers is much higher than in our algorithm. Using TCP-flooding, *s* nodes' *SBuffer* stores many more messages because of the choice of the communication protocol itself and the congestion avoidance strategy.

For broadcasting an application message inside its cell, a *s* node needs to send the message to each *h* node of its cell using TCP (point-to-point communication) whereas our algorithm sends only one UDP message (broadcast function). Hence, until reception of the corresponding acknowledgment, a *s* node keeps in its *SBuffer*, on average, ≈70/7=10 messages per application message for *TCP-Flooding*, and only one for our algorithm.



(a) Average messages in SBuffer of s nodes



(b) Sent amount of data per second

Fig. 6: Experimental results in static configuration

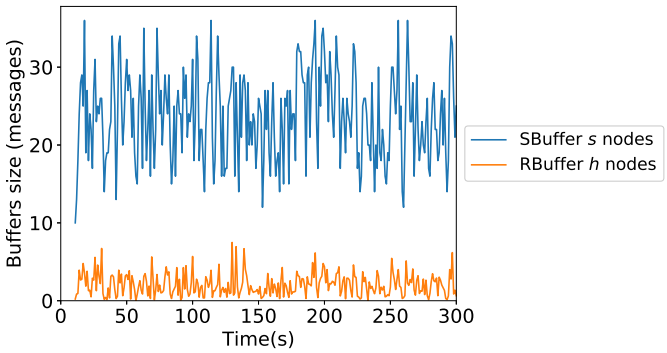
TCP's congestion avoidance strategy increases the time interval during which messages are stored in *SBuffer*. It consists of multiplying by 2 the retransmission delay (beginning at 0.2s) at each attempt. On the other hand, in our algorithm, retransmission delays start at 1s, and decrease with the number of sent messages, down to 200ms, aiming at delivering long outstanding messages faster. Moreover, TCP bounds the number of messages simultaneously sent to 46 (congestion window), while our algorithm bounds it to 150. Hence, *TCP-Flooding* takes longer to deliver messages, even though congestion avoidance reduces collisions (5% vs 10%).

A final remark is that we observe a low variation of buffer sizes in our algorithm, while *TCP-Flooding*'s buffer sizes vary a lot due to congestion avoidance (peaks) and fast retransmission (rapid decrease). The latter happens when outstanding messages are re-sent without waiting for the trigger of the corresponding timeouts upon detection of low network load. Our algorithm does not implement such mechanisms, and the buffer sizes are quite stable.

Figure 6b shows that *TCP-Flooding* sends more information than our algorithm. The former sends bigger messages than the latter, due to TCP's aggregation of messages. Hence, more data is lost and must be re-sent, even though fewer messages are lost. Moreover, *TCP-Flooding* sends more application messages, as well as more acknowledge messages (at least one/200ms vs one/500ms for our algorithm). Hence, the average ratio of messages sent per delivery, which stabilizes quickly for both algorithms, is much lower for our algorithm (0.4 msg/delivery) than *TCP-Flooding* (1.58 msg/delivery).

We evaluate the average message **delivery delay**, defined as the average time between the broadcast of an application message m (*coBroadcast(m)*) by a h node and the delivery of m (*coDeliver(m)*) by the h nodes. Our algorithm delivers messages with an average of 0.20s, much faster than *TCP-Flooding*, whose average delivery delay is 2s.

We also observe that *TCP-Flooding* hardly handles heavier **loaded networks**, because some h nodes will have great difficulty in receiving messages due to repeated collisions and the ensuring congestion avoidance strategy. Moreover, the establishment of TCP connections is sometimes long and even fails (exceeds 75 seconds). Consequently, we could not collect meaningful statistics for *TCP-Flooding* in heavier loaded scenarios. On the other hand, our algorithm tolerates it (up to 35 messages/second), with an average of s nodes' *SBuffer* size of 35 messages, an average delivery time of 0.31 seconds, *RBuffer* size of h nodes of 6 messages, an average of 700 sent messages/second, and, on average, 4300000 bytes of data sent/second.



(a) Average size of buffers

Fig. 7: Experimental results in a dynamic system

B. Dynamic system

In this section, we do not compare our algorithm with *TCP-Flooding* because the mobility model described in the article [13] is not comparable to ours: mobile nodes must always be connected to at least one base station preventing temporal disconnection. Furthermore, as we have shown in the previous section, the TCP channels approach used by the algorithm strongly degrades the performance of message flooding over mobile networks.

We keep the same network configuration, except that the h nodes move, with a velocity of 5km/h≈1.38m/s. Our algorithm handles h nodes moving outside the covered area, but the moving h node would then stop receiving and acknowledging messages, and the *SBuffer* of the s node at which it was previously registered would then grow until it reconnects to some s node. For the sake of clarity of results, h nodes move inside the area covered by the station cells.

Every h node sends, on average, an application message every 2.8 seconds, i.e., a total of 25 application messages are broadcasted per second. Note that the network load of this scenario is much higher (25 messages per second) than those of the static experiment (5.6 messages per second).

Figure 7a shows the size of s nodes' *SBuffers* and h nodes' *RBuffers*. We observe that network dynamics do not have a significant impact on performance. The variations of the buffer sizes are mostly due to the collision of messages (between 7.5-8.5%). The loss of a message delays the acknowledgment and the removal/delivery of all the messages whose sequence number is higher than the one of the lost message. On the other hand, when the latter is finally received, several messages are generally acknowledged. Hence, the buffer sizes first increase progressively, before decreasing abruptly. Moreover, the increasing peak of *RBuffer*'s size usually comprises the *RBuffer* of several h nodes of the same cell. In addition, h nodes in regions where cells overlap also have a bigger *RBuffer*, because of interferences caused by the collision of messages sent by the two s nodes whose cells are overlapping.

s nodes' *SBuffer*s usually contain no more than a few dozen messages. Moreover, h nodes mostly move between neighbor cells, which receive messages similarly, and which have a high probability to store the same messages. Hence, the list of discarded messages exchanged during the handoff is small or even empty and is quickly propagated among the neighbor h nodes. Some h nodes may not receive many messages when their cell is loaded, or when they are changing cell several times in a short time interval. Their cell's *SBuffer* and the list of discarded messages exchanged during the handoff then become much bigger (up to 200 messages).

We point out that the number of messages kept by *SBuffer* of h nodes and the *RBuffer* of s nodes are not shown in any figure since their respective size keeps very small during the whole execution (≈0.1-0.3 messages/node).

The average delivery latency is ≈0.3 seconds, slightly higher than in the static network (+0.05s), but keeps stable, even though buffers' size varies. Indeed, the majority of the h nodes receive and deliver messages at the first broadcast. Therefore, most of them are delivered quickly and with the same latency. The slight increase in delivery delay is due to the slight increase of collisions and the handoff procedures, where h nodes do not deliver messages, and messages they send are discarded.

C. Dynamic system with transient faults

We keep the dynamic configuration and inject transient faults on randomly chosen h nodes connected to the s node s_3. Beginning at t=15s of the experiment, a h node fails at every 30 seconds. The first failure lasts 3s and the duration of the failure increases by 1s at each new fault, i.e., the second lasts 4s, the third 5s, etc. A faulty node stops sending and receiving messages and comes back at its previous location.

Figure 8a shows the average size of all s nodes' *SBuffer*, and Figure 8b the size of s_3's *SBuffer*. The former shows that the main impacted *SBuffer* of s nodes is the one of the cell in which the fault occurs, which is s_3. Nevertheless, it drops quickly down to its previous size a few seconds after the faulty h node recovers.

We observe that the longer the failure duration, the bigger the *SBuffer* size of the s node to which the faulty h node was connected, and the longer the delay required for this h node to deliver the outstanding messages of the s node's *SBuffer* when it recovers. Thus, the maximum duration of faults that s nodes can cope with is bounded by the maximum number of simultaneously disseminated messages per second (supported network load), and the maximum number of messages that s nodes' *SBuffer* can keep. In our simulations, s nodes can send up to 400 messages simultaneously. Beyond it, messages will not be delivered fast enough to counterbalance new incoming messages, because of too many interferences. The number of outstanding messages of the s node's *SBuffer* to which the faulty h node was connected will then increase indefinitely, rendering the cell unstable. This instability will eventually propagate to the other cells.

Figure 8b also shows two special cases. First, the *SBuffer* of s_3 does not entirely recover between t=200 and 250s. A h node fails at t=195s and recovers at t=204s. s_3's *SBuffer* then begin to reduce, but a new fault occurs at 225s, before the *SBuffer* comes back to its original size. Secondly, between 165 and 180s and 225 and 250s the size of *SBuffer* of another station (resp. s_0 and s_5) also grows due to interferences when s_3 broadcasts many messages, or by the changing of cells of a recovering h node before it received all outstanding messages.

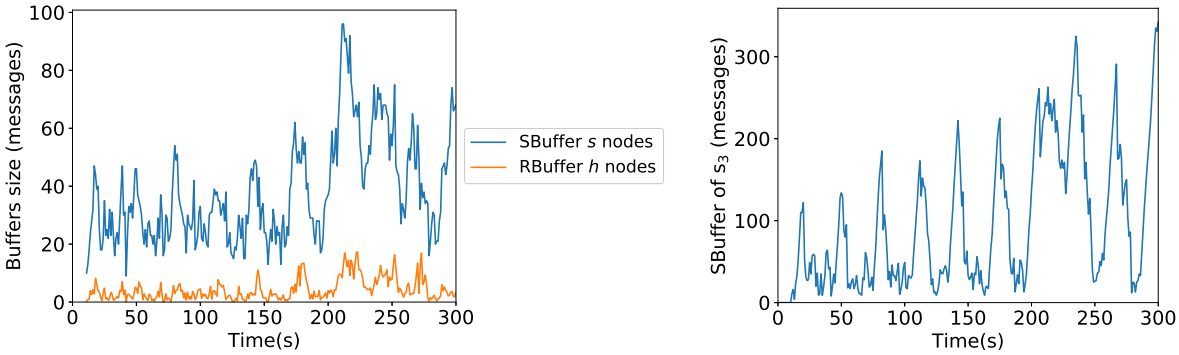
D. Scalability and memory analysis

The control information included in application messages concerns just some few integers while experiments show that the list of id's (M_d), piggybacked on s nodes' application messages usually contains one or no id.

The two lists of messages exchanged during the handoff between the s nodes contain at most the number of messages stored by the previous s node's *SBuffer* to which the moving h node was connected. Experiments show that the *SBuffer* of s nodes remains small, and practically very few messages are received by s nodes during the handoff procedure, since h nodes mostly move between adjacent s nodes. Hence, messages piggyback a few control information.

Additionally to the *SBuffer* and *RBuffer*, h nodes keep some integer variables, and s nodes some integers and a structure for each connected h node. Experiments show that all buffers, except s nodes' *SBuffer*, remain very small. s nodes' *SBuffer* contains only a few dozen messages except temporarly when some previously connected h node fails or if the s node's cell is overloaded. The structure s nodes keep for each h node contains only a few integers. The number of those structures a s node keeps grows linearly with the number of h nodes connected to the s node's cell.

In summary, the amount of control information required by the algorithm is low for h nodes and grows linearly in terms of locally connected h nodes for s nodes.



(a) Average size of buffers

(b) Average size of SBuffer of s_3

Fig. 8: Experimental results with transient failures

E. Experiments conclusion

We can draw the following conclusions from the results of our experiments:

Firstly, *TCP-Flooding* algorithm induces a high local control information overhead, increased transmission delays, and a high number and size of sent messages. Therefore, it is not suited for wireless networks, even in static systems.

Secondly, our algorithm shows good results in a dynamic network. Node mobility has a slight impact on the performance of causal broadcast, and performance is similar to those presented for static networks. On the other hand, transient faults degrade performance, particularly within the cell in which they occur. However, such a degradation disappears after a few seconds when the faulty *h* nodes recover. Hence, our algorithm tolerates high load, mobility, and transient faults.

Finally, in terms of scalability, very few control information is piggybacked on messages. Moreover, *h* nodes maintain few information, and *s* nodes' memory footprint grows linearly with the number of locally connected *h* nodes. Overall our algorithm has a small memory footprint.

VI. Related work

Several approaches have been proposed aiming at reducing the control information necessary to implement causal broadcast algorithms in distributed systems. However, they are not always suitable for tolerating the dynamics, and wireless interferences of mobile networks neither present solutions for discarding no longer usable messages.

Prakash and al. observed in [14] that, for controlling causal order among messages, it is sufficient to piggyback in a message just the information about its direct dependencies. Even though this approach copes with node dynamics (churn) of mobile networks, since it is based on message identifiers and not on node identifiers, every node maintains a matrix of size N^2, where N is the maximum number of nodes of the system. Additionally, in the worst case, the control information attached to messages presents a size of O(N).

Vector clocks [6][10] of size N are the smallest data structures which capture causality of events of a system with N nodes [5]. Nevertheless, to implement a causal broadcast algorithm that exploits vector clocks [16], each node keeps a local vector variable of size N, which is included in every broadcasted message. Hence, since the vector size is fixed to N, such clocks are not adjustable for systems where the number of nodes varies dynamically. Their size still grows linearly with N, even with solutions that compress them, such as [3]. Moreover, a garbage collector and additional control information are needed to delete obsolete messages.

By applying Bloom filters on messages, Ramabaja aims in [15] at reducing messages' piggybacked information. These filters have a much lower space complexity, but they can throw false positive (but not false negative) requiring, therefore, a mechanism to handle them. Furthermore, in order to limit the number of false positives, Bloom filter's size should increase proportionally to the number of nodes.

Both plausible [17] and probabilistic clocks [2] are vector clocks whose size is much smaller than the number of nodes in the system. The quality of the detected causality information is related to the vector size: the greater the size of the vector, the higher the accuracy of the captured causality. In plausible clocks, an entry of the vector clock is associated with several nodes. The probabilistic clocks extend the plausible clock by also associating several entries to a node. The use of these vector clocks by causal broadcast algorithms [2] strongly reduces the size of control information but they do only capture causality among broadcast messages with a high probability. An extra procedure is thus necessary to handle causally related messages that were not ordered. Moreover, contrary to our solution, the authors do not propose any mechanism to discard delivered messages.

Some works address scalability issues by organizing nodes in logical structures. In Adly et al. [1], nodes are grouped into clusters, logically organized into a tree. Thereby, a node needs to send and keep track of messages only to/from few nodes. However, at the presence of node churn, clusters often need to be reorganized. Moreover, the mobility of a node is restricted to its cluster. Hence, it is not suitable for mobile networks.

By organizing the nodes into an application-level tree on top of which messages are propagated, Blessing et al.'s causal broadcast algorithm [4] does not require that messages carry any causal information. However, the authors do not consider system dynamics.

Nédelec et al. [13][12] extend this approach to dynamic topologies. A node discards a message once it has received

it by each of its links. Links are FIFO, and they can be dynamically added or removed between nodes using handoff procedures, provided that the nodes of the system are always connected through initialized links. Therefore, dynamics are tolerated under certain conditions. Particularly, to add a new link between two nodes, an already initialized path must exist between them. Hence, the system can never be partitioned.

VII. Conclusion

We have presented in this article a causal broadcast algorithm that is tailored to the characteristics of mobile networks, such as nodes mobility, dynamic membership and connections, mobile nodes memory constraints, scalability issues, and wireless interferences. The size of required information piggybacked on messages is small, as well as mobile nodes' memory footprint, while mobile support stations' memory footprint grows linearly with the number of local connected mobile nodes. Simulation results on OMNet++ show that TCP induces a heavy message overhead in mobile networks and that our causal broadcast has good performance in both static and dynamic systems.

For future work, we aim to extend our causal broadcast algorithm in order to handle mobility of support stations. A second research direction will be a hybrid approach with FIFO ordering and probabilistic clocks [2], proposed in [12]. The latter partially restores causality tracking and thus allows the delivery of some concurrent messages without waiting for their FIFO ordering.

References

[1] N. Adly and M. Nagi. Maintaining causal order in large scale distributed systems using a logical hierarchy. In *Proc. IASTED Int. Conf. on Applied Informatics*, pages 214–219, 1995.

[2] Achour Mostéfaoui an Stéphane Weiss. Probabilistic causal message ordering. In Victor Malyshkin, editor, *Parallel Computing Technologies - 14th International Conference, PaCT 2017, Nizhny Novgorod, Russia, September 4-8, 2017, Proceedings*, volume 10421 of *Lecture Notes in Computer Science*, pages 315–326. Springer, 2017.

[3] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[4] Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. Tree topologies for causal message delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, page 1–10, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

[6] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988.

[7] Roy Friedman and Shiri Manor. Causal ordering in deterministic overlay networks. 05 2004.

[8] Tomasz Imielinski and B. R. Badrinath. Mobile wireless computing: Challenges in data management. *Commun. ACM*, 37(10):18–28, October 1994.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[10] Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[11] Achour Mostéfaoui, Matthieu Perrin, Michel Raynal, and Jiannong Cao. Crash-tolerant causal broadcast in $O(n)$ messages. *Inf. Process. Lett.*, 151, 2019.

[12] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. Causal broadcast: How to forget? In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, volume 125 of *LIPIcs*, pages 20:1–20:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[13] B. Nédelec, P. Molli, and A. Mostéfaoui. Breaking the scalability barrier of causal broadcast for large and dynamic systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60, 2018.

[14] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 744–751, 1996.

[15] Lum Ramabaja. The bloom clock. *CoRR*, abs/1905.13064, 2019.

[16] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Distributed Algorithms, 3rd International Workshop, Nice, France, September 26-28, 1989, Proceedings*, pages 219–232, 1989.

[17] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In Özalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms, 10th International Workshop, WDAG '96, Bologna, Italy, October 9-11, 1996, Proceedings*, volume 1151 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 1996.

[18] András Varga. The omnet++ discrete event simulation system. *Proc. ESM'2001*, 9, 2001.

[19] Daniel Wilhelm, Luciana Arantes, and Pierre Sens. A scalable causal broadcast that tolerates dynamics of mobile networks. Technical report, Sorbonne University , UPMC, May 2020.