



**HAL**  
open science

# A scalable causal broadcast that tolerates dynamics of mobile networks

Daniel Wilhelm, Luciana Arantes, Pierre Sens

► **To cite this version:**

Daniel Wilhelm, Luciana Arantes, Pierre Sens. A scalable causal broadcast that tolerates dynamics of mobile networks. 23rd International Conference on Distributed Computing and Networking (ICDCN), Jan 2022, New Delhi / Virtual, India. hal-03524944

**HAL Id: hal-03524944**

**<https://inria.hal.science/hal-03524944>**

Submitted on 13 Jan 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# A scalable causal broadcast that tolerates dynamics of mobile networks

Daniel Wilhelm

Sorbonne Université, Inria, LIP6  
Paris, France

Luciana Arantes

Sorbonne Université, Inria, LIP6  
Paris, France

Pierre Sens

Sorbonne Université, Inria, LIP6  
Paris, France

## ABSTRACT

Causal broadcast is at the core of collaborative applications, distributed databases, conferencing, or social networks. Existing causal broadcast algorithms are either not scalable or cannot be implemented on mobile networks because they do not take into account the features of these networks: limited capacities of nodes (computation, storage, energy), unreliable communication channels, and the dynamics of connections due to node mobility, node failure, and join/leave of nodes. This work presents a causal broadcast algorithm for mobile networks. The algorithm is scalable: control information piggybacked on messages and maintained on nodes is of small size. Experiments conducted on OMNeT++, a realistic network simulator, confirms the effectiveness of our causal broadcast protocol, rendering causal broadcast affordable in mobile networks.

## 1 INTRODUCTION

Causal broadcast is a fundamental group communication service largely used by many applications such as distributed databases, publish/subscribe systems, collaborative applications, or distributed social networks. It ensures that messages are delivered to all nodes exactly once, preserving causal relation between messages, i.e., the delivery of a message respects Lamport's happened-before relationship [11]: if the broadcast of a message  $m$  precedes the broadcast of a message  $m'$ , then no process delivers  $m'$  before  $m$ .

Due to scalability issues, changes in network topology, and node resource limitations [10], the implementation of a causal broadcast service for mobile networks composed of mobile hosts and static support stations is a challenge. In such networks, stations are connected through a wired network, while a station communicates through the wireless network with hosts inside its geographic coverage area, denoted cell. Hosts move between cells and only communicate with the station of their current cell. They have memory, computational power, and battery life limitations [10][6][17][2] and are subject to transient or permanent failures when, for instance, its battery becomes flat, or a hardware failure occurs. Interferences cause message losses on the wireless network. To the best of our knowledge, no causal broadcast solution handling those characteristics has been proposed.

Existing implementations of causal broadcast either piggyback information on messages [14][17][8][13] or organize nodes in logical topologies in which messages are disseminated through reliable FIFO channels [9][16]. The first approach is not suitable for mobile networks that contain many mobile hosts, due to scalability issues, because the size of information piggybacked on messages either grows with the number of nodes [8][13], or with the message load [14][17]. The second approach [9] [16] ensures that messages are implicitly causally ordered at reception. Therefore, no information

is piggybacked on messages, making that approach scalable. However, [9] considers a static network, and the dynamic model of [16] is not applicable to mobile networks. Moreover, [9][16] require reliable FIFO channels, which are not provided by mobile networks, even with TCP, because a host moving to a new cell drops the connection with its previous cell, and pending messages of that connection are lost.

Several works address causal multicast in mobile networks [6][17][2][3][12]. However, they usually make unrealistic assumptions such as reliable [6][17] and/or FIFO [3][12] channels, or reliable host connection protocols. In addition, they do not address the problem that hosts might fail to connect to a cell's station before moving to another cell, which might lead to many concurrent connection initializations. Furthermore, they usually consider that hosts are reliable, i.e., they never fail, and the proposed solutions do not scale. Finally, causal multicast requires additional information to handle multicast groups, not necessary for causal broadcast.

Stations handle interferences on the wireless network by caching messages for retransmission. Discarding cached messages once they become obsolete is an important issue to deal with. Some existing implementations, such as [6][3], propose costly centralized solutions since the source station that initially broadcast a message coordinates the protocol for discarding it from all other stations, inducing a high overhead of message traffic (acknowledge messages) and memory storage (a message is cached at all stations even if only one station requires it).

This work presents a scalable causal broadcast algorithm designed for mobile networks. Hosts can join/leave the network and fail, permanently or transiently, at any time. They move freely and might be temporarily disconnected from the network when out of range of all stations. We assume no reliable connection protocol, and the algorithm handles multiple concurrent connections by the same host. Resource limitations of hosts are handled by keeping causal information at stations, while hosts only keep very little control information. Messages piggyback only a few integers as control information. Contrarily to existing centralized solutions [6][3], the discarding of obsolete messages cached at stations is decentralized: a station discards a message once all hosts connected to it acknowledged the message. Consequently, stations only cache necessary messages, and no extra messages are exchanged for message discarding.

Summing up, the proposed causal broadcast algorithm is scalable in both the number of hosts and stations, has a low message traffic and storage overhead, while handling mobile network dynamics without the constraining assumptions of FIFO wireless links [9][16][15] or causal multicast approaches [6][3].

Experiments were conducted over OMNeT++/INET [21]. The proposed algorithm is compared to [6], a causal multicast algorithm for mobile networks which we extended to causal broadcast. Experiments confirm that our algorithm outperforms the latter.

The paper is organized as follows. Section 6 discusses related work. Section 2 presents the background and Section 3 the model. Section 4 describes the proposed causal broadcast algorithm. Section 5 presents the experimental results. Section 7 concludes the paper.

## 2 BACKGROUND

Causal order ensures that nodes deliver messages while respecting the causal relation between them, as defined by the happened-before relation [11] introduced by Leslie Lamport:

**Happened-before relation:** *Considering two events  $e_1$  and  $e_2$ ,  $e_1$  causally precedes  $e_2$ , or  $e_1 \rightarrow e_2$  iff: (a)  $e_1$  and  $e_2$  occur on the same process and  $e_1$  precedes  $e_2$  or (b) for a message  $m$   $e_1 = \text{send}(m)$  and  $e_2 = \text{deliver}(m)$  or (c) there exists an event  $e_3$  such that  $e_1 \rightarrow e_3$  and  $e_3 \rightarrow e_2$ .*

Causal broadcast is a group communication service that provides the application processes with two primitives *co-broadcast*( $m$ ), that broadcasts a message  $m$  to all nodes, and *co-deliver*( $m$ ), that delivers  $m$  to the application. Causal broadcast is defined by the following properties [15]:

**Causal broadcast: Validity.** If a process  $q$  co-delivers a message  $m$  from a process  $q$ ,  $q$  previously co-broadcasts  $m$ .

**Integrity.** A process co-delivers a message  $m$  at most once.

**Causal order.** The delivery order of messages follows the happened-before relationship:  $\text{co-broadcast}(m) \rightarrow \text{co-broadcast}(m') \Rightarrow \nexists$  process  $q \mid \text{co-deliver}(m') \rightarrow \text{co-deliver}(m)$ .

**Termination.** A message co-broadcasted by a correct process is co-delivered by all correct processes.

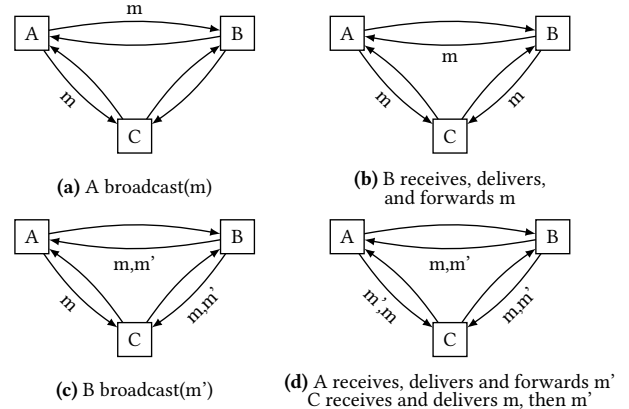
The delivery of a message is delayed until all messages that causally precede it are delivered. Usually, in mobile networks, stations discard messages once all hosts delivered them. Hosts that join the network will, therefore, not receive and deliver these messages.

Our algorithm extends the FIFO dissemination approach proposed in [9] to mobile networks. The algorithm of [9] ensures causal order through FIFO dissemination in static networks where reliable nodes are connected by reliable FIFO channels, thereby ensuring that no path exists over which messages travel out of causal order. For example, in Figure 1  $A$  broadcasts  $m$ , which causally precedes  $m'$  broadcast by  $B$ . All nodes receive  $m$  before  $m'$  since all nodes forward  $m$  at reception on all their channels.

## 3 MODEL

Mobile networks are composed by mobile hosts, denoted *host*( $s$ ), and static support stations, denoted *station*( $s$ ). Hosts and stations communicate through message passing. Applications running on hosts use a group communication service to join and leave the network, as well as to broadcast messages to all hosts and deliver them in causal order.

The characteristics of *stations* are the following:



**Figure 1: Causal broadcast by dissemination [9]**

- Each station is at the center of a cell, corresponding to the area covered by its antenna's transmission range.
- Stations are supposed reliable and do not move or leave the network, because a failing station would disconnect the area covered by it, and cell overlapping should be avoided (even though it cannot be realistically completely prevented) because of interferences on the wireless network. Hence, we assume hardware replication for stations.
- Stations do not have energy limitations and have much more storage and computational capacity than hosts.

The characteristics of *hosts* are the following:

- A host is connected to at most one station (generally the closest one) at any given moment, and communicates with the system through that station, by sending messages on the wireless network.
- A host can join and leave the network at any moment. A host that joins the network will not deliver those messages that the station to which it connects has previously discarded.
- Hosts move freely inside and outside cells.
- Hosts might temporarily or permanently fail. For example, a host is temporarily faulty until its battery is recharged, or is permanently faulty if it has a hardware failure. A faulty host stops sending, receiving, and processing messages, and loses all variables stored in volatile memory.
- Hosts have computational, storage, and energy limitations.

The wireless network is unreliable, due to interferences that lead to message losses while stations are connected by a high-speed wired network, which is FIFO and reliable, and over which is built a static logical tree-based overlay.

Hosts can be in the states *up* or *down*. A station controls the state of hosts connected to it: it considers a connected host as down if the host leaves the system, or if it receives no message from the host for a given interval of time (assume a failure). Otherwise, the station considers the host as *up*.

A host joining the system does not deliver those messages already discarded by the station that acknowledges its system join. Hence, we modify the *termination* condition of **causal broadcast**:

**Termination.** Note  $s_{j,k}$  the station from which host  $k$  received the system join acknowledgment. A message  $m$  co-broadcasted by an

up process is co-delivered by all up processes  $k$  for which  $s_{j,k}$  did not discard  $m$  prior to  $k$ 's connection.

## 4 CAUSAL BROADCAST ALGORITHM

This section presents our causal broadcast algorithm, denoted *WAS*, which extends the FIFO dissemination approach [9] to mobile networks, where hosts are not static, neither reliable, and communicate through unreliable wireless channels with stations which, in their turn, communicate among themselves through reliable channels.

The algorithm is divided into three parts: (1) dissemination of application messages, (2) join/leave operations, and (3) handoff procedure to handle hosts moving between cells. The proof of the algorithm can be found in our technical report [22].

### 4.1 Dissemination of application messages

Hosts are the source of application messages, and stations ensure that all hosts deliver them causally. A host broadcasts an application message by sending it to the station to which it is connected. The station then forwards the message to the hosts of its cell through the wireless network, as well as to other stations through the wired network. Each station ensures that the hosts connected to it deliver application messages in causal order.

On the wireless network, message loss and FIFO ordering are handled by assigning sequence numbers to messages, and storing them in buffers for retransmission until destination nodes acknowledge them. A host uses variables  $seq_h$  to order messages it sends, and  $seq_{NC}$  to order messages it receives. A station uses  $seq_C$  to order messages it broadcasts, and stores  $seq_h$  and  $seq_{NC}$  of each host connected to it. In both stations and hosts, a local buffer, *RBuffer*, stores received messages until they are FIFO ordered, and *SBuffer* stores sent messages until they are acknowledged by all destination(s). A host keeps a message in *SBuffer* until its cell's station acknowledges the message. A station keeps a message in *SBuffer* until all hosts connected to it acknowledge the message.

Every application message is uniquely identified by  $(id_h, seq_h)$ , where  $seq_h$  is the sequence number that host  $id_h$  associated to the message. Moreover, since cells may overlap, every message sent over a wireless network cell also piggybacks the id of this cell ( $id_C$ ). Upon reception of the message, hosts and stations verify  $id_C$  and only take into account messages sent inside their cell.

Nodes (hosts and stations) regularly send acknowledge messages, containing ranges of sequence numbers to acknowledge application messages whose sequence numbers are in these ranges. Hosts do not send acknowledgments during handoffs, since they would acknowledge application messages received from another station, which might order messages differently.

Application messages which are delivered by all hosts are considered obsolete and should then be discarded by stations. Existing algorithms use a centralized deletion of obsolete messages [6], which has an overhead both in terms of stored messages in station *SBuffer* and the number of messages sent over the wired network: each application message  $m$  is managed by a station called  $MSS_{init}$ , and acknowledge messages for  $m$  from each host are forwarded to  $MSS_{init}$ . Once  $MSS_{init}$  received an acknowledge message from each host, it broadcasts a delete message to all stations in order to discard  $m$ . To overcome such an overhead, we have introduced a

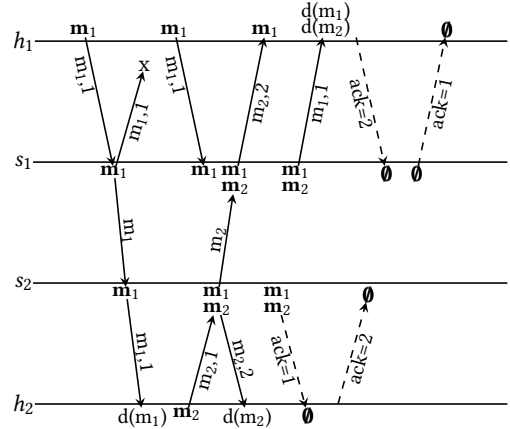


Figure 2: Broadcast of  $m_1$  and  $m_2$

decentralized mechanism where stations discard messages using only local information: a station discards a message as soon as all hosts connected to it have acknowledged the message. Therefore, our approach does not require message exchanges between stations and a station only caches messages required by hosts of its cell.

Figure 2 shows the broadcast and delivery of two messages causally related.  $h_1$  is connected to  $s_1$  and  $h_2$  to  $s_2$ . Stations  $s_1$  and  $s_2$  are connected by a wired channel. Hosts (resp. stations) piggyback  $seq_h$  (resp.  $seq_C$ ) in application messages. *SBuffers* are represented in bold. First,  $h_1$  broadcasts  $m_1$ . Upon reception,  $s_1$  forwards  $m_1$  to  $s_2$  and broadcasts it in its cell, which contains  $h_1$ . Upon reception,  $s_2$  forwards  $m_1$  to  $h_2$ .  $h_2$  receives and delivers  $m_1$ , then broadcasts  $m_2$  (co-broadcast( $m_1$ )  $\rightarrow$  co-broadcast( $m_2$ )).  $h_2$  (resp.  $s_2$ ) stops transmitting  $m_1$  (resp.  $m_1$  and  $m_2$ ) upon reception of the acknowledge message regularly sent by  $s_2$  (resp.  $h_2$ ). Suppose  $h_1$  did not receive neither  $m_1$  the first time  $s_1$  broadcast it due to interferences, nor its acknowledgment. Hence,  $h_1$  retransmits  $m_1$ .  $s_1$  ignores the second reception of  $m_1$  since it already received  $m_1$ . Upon reception of  $m_2$ ,  $s_1$  broadcasts it. Then  $h_1$  receives and buffers  $m_2$  because the sequence number that  $s_1$  attached to  $m_2$  is equal to 2, and  $h_1$  awaits a message with  $seq=1$ . Eventually,  $s_1$  broadcasts  $m_1$  again and, upon reception,  $h_1$  delivers  $m_1$  then  $m_2$ . Finally,  $h_1$  (resp.  $s_1$ ) acknowledges  $m_1$  and  $m_2$  (resp.  $m_1$ ). Hence,  $m_1$  and  $m_2$  are completely discarded from the network, i.e., removed from the buffers of all nodes.

### 4.2 Join/leave the network

**Joining the network.** Hosts can join the network during execution. Host  $h_j$  joins the network by sending a *join* message to station  $s_i$  which then replies with the corresponding *connect<sub>ACK</sub>* message.  $h_j$  regularly retransmits *join* until receiving *connect<sub>ACK</sub>*. Message exchanges between  $h_j$  and  $s_i$  can result in one of the three scenarios presented in Figure 3: (a) both the *join* and *connect<sub>ACK</sub>* message are received; or due to interferences (b) the join message or, (c) the *connect<sub>ACK</sub>* message, is lost. Since  $h_j$  cannot distinguish between (b) and (c), it cannot determine if  $s_i$  has received *join*, or not, until receiving the corresponding *connect<sub>ACK</sub>* message. Hence,  $h_j$  cannot determine if  $s_i$  registered it before receiving *connect<sub>ACK</sub>*.

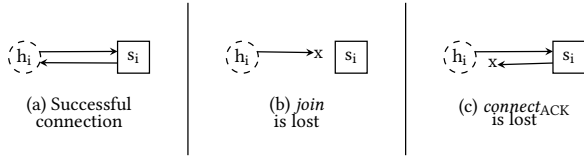


Figure 3: Host connection scenarios

Host  $h_i$  might also move to another cell  $s_j$  before receiving  $connect_{ACK}$  from its previous station  $s_i$ . In this case,  $h_i$  will send a  $join$  message to  $s_j$ , despite the fact that  $s_i$  might have also received a  $join$  message, and therefore registered  $h_i$ . However, a host should be connected to only one station, which is the one that received the latest  $join$  message sent by the host, in this case  $s_j$ .

In order to handle its multiple registrations,  $h_i$  keeps the list PS which contains the stations that might have registered its connection attempts. It identifies each of these attempts with a connection sequence number, denoted  $Ses$ , which it increments when changing cells. Thus, when  $h_i$  changes cells during the connection  $Ses$ , before receiving a  $connect_{ACK}$  message from  $s_i$ ,  $h_i$  saves the tuple  $(s_i, Ses)$  in PS. Host  $h_i$  then attaches PS to the  $join$  message sent in the new cell. Every station also keeps the  $Ses$  value corresponding to the latest connection of every host connected to it.

Station  $s_j$  replies to  $join$  with a  $connect_{ACK}$  message which contains  $seq_C$ , the sequence number of the oldest message  $s_j$  caches in  $SBuffer$ . It also registers  $h_i$  and sends a  $Delete$  message for each tuple  $(s_i, Ses)$  in PS. Upon receiving a  $Delete$  message,  $s_i$  unregisters  $h_i$ , if it has been registered in the connection  $Ses$ . Such a procedure ensures that eventually  $h_i$  is only registered at  $s_j$ .

Upon reception of  $connect_{ACK}$ ,  $h_i$  completes the connection initialization by updating  $seq_{NC}$  to  $seq_C$ , and PS to  $(s_j, Ses)$ , since  $s_j$  will be the station that will control causal information related to it.  $h_i$  will then deliver messages based on  $seq_{NC}$  sequence number.

**Leaving the network.** A host leaves the network by sending a  $leave$  message containing PS, until a station, regardless of which one, acknowledges it. The station which receives  $leave$  sends a  $Delete$  message to each station  $s$  of the tuples  $(s, Ses)$  of PS, which will unregister  $h_i$  if registered.

### 4.3 Handoff procedure

A station maintains causal information related to those hosts connected to it. Hence, when a host  $h_i$  changes cell, its causal information must be transferred from its previous cell's station  $s_p$  to its new cell's station  $s_n$ . Existing handoff procedures [6][2][3] make several unrealistic assumptions, such as reliable wireless channels or that a moving host always succeeds in connecting to its cell's station before moving to another cell.

Moreover, they usually identify messages on the wired network by vector clocks, whose size is the number of stations. Hence, they are not scalable in terms of stations. The handoff procedure of our algorithm does not make such assumptions and its dissemination approach scales. It handles message losses on the wireless network as well as simultaneous connection attempts of the same host, without doing assumptions about the success of those connection attempts. Our dissemination approach scales, but requires an extra control to compare the message ordering at stations. Finally, our algorithm implements a decentralized discard approach of obsolete

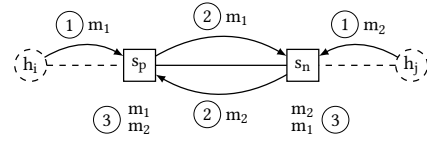


Figure 4: Sequence number assignment by stations

messages at stations, removing the message and memory overhead of centralized approaches, but also requiring an extra control when hosts change cells.

We start the handoff section by discussing the problems that the handoff procedure faces (handoff challenges), then we describe the handoff procedure algorithm itself, and finally, we give a handoff execution example.

**4.3.1 Handoff challenges.** We highlight the following points:

**Host connections.** In a real configuration, a moving host  $h_i$  cannot determine which station keeps its causal information, since it cannot distinguish between cases (b) and (c) of Figure 3. Furthermore, it might connect to several stations in a short time interval, which might lead to simultaneous  $h_i$ 's handoff instances. Thus, simultaneous handoff procedures of host  $h_i$  must be handled, by ensuring that the latest handoff procedure will correctly be executed and that previous handoff procedures are ended/aborted.

**Message ordering.** In the dissemination phase, stations assign local sequence numbers to messages as they are received. For example, in Figure 4, at (1),  $h_i$  and  $h_2$  broadcast  $m_1$  and  $m_2$  respectively. At (2),  $s_p$  receives  $m_1$  and forwards it to  $s_n$ . Similarly,  $s_n$  forwards  $m_2$  to  $s_p$ . At (3),  $s_p$  and  $s_n$  respectively adopt  $[m_1, m_2]$  and  $[m_2, m_1]$  orders. Consequently, if  $h_i$  delivers  $m_1$  while connected to  $s_p$  and then connects itself to  $s_n$ ,  $s_n$  cannot assign to  $h_i$  the sequence number  $seq_{NC}=1$ , because  $h_i$  would then never deliver  $m_2$  and deliver  $m_1$  twice. In fact,  $seq_{NC}$  is only meaningful for the last initialized connection of  $h_i$ , which is identified by  $Ses_{SLC}$ , the last connection number in which  $h_i$  initialized  $seq_{NC}$ . To determine which messages  $h_i$  has delivered,  $s_n$  must therefore exchange information with  $s_p$ .

**Decentralized discard mechanism of obsolete messages.** A station discards a message once all hosts connected to it have acknowledged the message. This local deletion approach may lead to different  $SBuffer$  states of  $s_n$  and  $s_p$ : when  $h_i$  connects to  $s_n$ ,  $s_n$  might have already discarded messages that  $s_p$  still caches, and that  $h_i$  did not deliver yet. For instance, in Figure 4, suppose that  $h_i$  connects to  $s_n$  before delivering  $m_2$ , but that  $s_n$  already discarded  $m_2$  when  $h_i$  connects to it.  $s_n$  must then recover  $m_2$  from  $s_p$ , and  $h_i$  should deliver  $m_2$  before delivering messages currently broadcasted by  $s_n$ .

Another point that makes the comparison of the  $SBuffer$  states of  $s_n$  and  $s_p$  difficult, is that  $s_p$  does not maintain information about messages it has already discarded. For example, in Figure 4, if  $s_p$  deletes  $m_1$  once  $h_i$  acknowledged it, then when  $h_i$  moves to  $s_n$ ,  $s_p$  cannot inform  $s_n$  that  $h_i$  has already delivered  $m_1$ , since it caches no information about  $m_1$ . Moreover, since stations receive messages at different times,  $s_n$  might receive messages that  $s_p$  only receives later.  $s_n$  must distinguish between messages that  $s_p$  receives during the handoff from messages that  $s_p$  already discarded (i.e.  $m_1$ ).

---

**Task of  $h_i$  :**

---

**Moving from  $s_p$ 's cell to  $s_n$ 's cell**

- 1: **if** did not receive  $connect_{ACK}$  from  $s_p$  **then**
  - 2:  $PS \cup = \{s_p, Ses\}$
  - 3:  $Ses++$
  - 4: send( $\langle connect, id_h, seq_{NC}, Ses, Ses_{LC}, PS \rangle$ ) to  $s_n$
  - receive**  $\langle connect_{ACK}, seq_{h_m}, seq_{C_m}, Ses_m \rangle$ :
  - 5: **if**  $Ses == Ses_m$  **then**
  - 6:  $seq_{NC} = seq_{C_m}; seq_h = seq_{h_m}; Ses_{LC} = Ses_m; PS = \{s_n, Ses_m\}$
- 

---

**Task of  $s_n$  :**

---

**receive**  $\langle connect, id_h, seq_{NC}, Ses, Ses_{LC}, PS \rangle$  **from host  $h_i$ :**

- 1: **if**  $id_h$  registered  $\wedge$  handoff over  $\wedge$   $Ses \geq Ses_i$  **then**
  - 2: **if**  $Ses_{LC} == Ses_i$  **then**
  - 3: use  $seq_{NC}$  as acknowledgment
  - 4:  $Ses_i = Ses; PS_i = PS$
  - 5:  $seq_{C_i} =$  determine  $seq_C$  attributed to  $h_i$
  - 6: **if** no message discarded from  $SBuffer$  to deliver **then**
  - 7: send( $\langle connect_{ACK}, id_h, seq_{C_i}, Ses_i \rangle$ ) to  $h_i$
  - 8: **else**
  - 9: Register( $id_h$ )
  - 10: send( $\langle Req_1, id_h, seq_{NC}, Ses_{LC}, Ses \rangle$ ) to stations of PS
  - receive**  $\langle Rsp_1, id_h, seq_h, m_{nd}, Ses \rangle$  **from station  $s_p$ :**
  - 11:  $seq_{h_i} = seq_h$
  - 12:  $msg_{req} =$  message of  $m_{nd}$  that  $s_n$  already discarded
  - 13: send( $\langle Req_2, id_h, msg_{req}, Ses \rangle$ ) to  $s_p$
  - receive**  $\langle Rsp_2, id_h, msg, msg_{rcv}, Ses \rangle$  **from station  $s_p$ :**
  - 14: piggyback  $id_h$  on messages  $h_i$  has delivered at  $s_p$
  - 15: send to  $h_i$  messages already discarded by  $s_n$  ( $\in msg$ )
  - 16: **if**  $msg == \emptyset$  **then**
  - 17:  $seq_{C_i} = \min(m.seq, m \in s_n.SBuffer \text{ not delivered by } h_i)$
  - 18: send( $\langle connect_{ACK}, id_h, seq_{C_i}, Ses_i \rangle$ ) to  $h_i$
  - 19: send( $\langle Delete, id_h, PS[i], Ses \rangle$ ) to stations of  $PS_i$
  - 20:  $rcv(Req_1)$  for stored  $Req_1$  with highest  $Ses$
- 

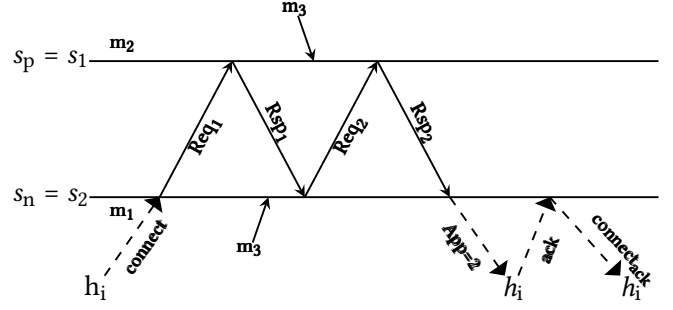
---

**Task of  $s_p$  :**

---

**receive**  $\langle Req_1, id_h, seq_{NC}, Ses_{LC}, Ses \rangle$ :

- 1: **if** current handoff and  $Ses > Ses_i$  **then**
  - 2: save  $Req_1$
  - 3: **else if**  $Ses > Ses_i$  **then**
  - 4: **if**  $Ses_{LC} == Ses_i$  **then**
  - 5: use  $seq_{NC}$  as acknowledgment
  - 6:  $m_{nd} = \{(id_h, seq_h) \text{ of messages } h_i \text{ has not delivered}\}$
  - 7:  $Ses_i = Ses$
  - 8: send( $\langle Rsp_1, id_h, seq_{h_i}, m_{nd}, Ses \rangle$ ) to  $s_n$
  - receive**  $\langle Req_2, id_h, msg_{req}, Ses \rangle$ :
  - 9:  $msg =$  messages  $s_n$  requests in  $msg_{req}$
  - 10:  $m_{rcv} =$  messages  $s_p$  received since receive( $Req_1$ )
  - 11: unregister( $id_h$ )
  - 12: send( $\langle Rsp_2, id_h, msg, m_{rcv}, Ses \rangle$ ) to  $s_n$
- 

**Figure 5: Handoff procedure**

4.3.2 *Handoff procedure description.* When  $h_i$  moves to  $s_n$ 's cell, it sends a *connect* message to  $s_n$  that contains: the connection sequence number  $Ses$ , the connection sequence number  $Ses_{LC}$  of its last acknowledged connection, the sequence number  $seq_{NC}$  of the last message  $h_i$  delivered, and the list  $PS$  containing the stations that might register  $h_i$ .

The pseudo-code of the handoff procedure is given by Tasks  $h_i$ ,  $s_n$ , and  $s_p$ .

Upon reception of a *connect* message,  $s_n$  verifies if it has already registered  $h_i$  (line  $s_n.1$ ) which happens if (1)  $h_i$  was previously connected to  $s_n$ , then tried to connect to another station but the *connect* message was lost (Figure 3.b), and then  $h_i$  tried to re-connect to  $s_n$ ; (2)  $s_n$  already received the *connect* message but the *connect<sub>ACK</sub>* message was lost (Figure 3.c). Station  $s_n$  distinguishes between (1) and (2) with the help of  $Ses_i$ , since in case (1),  $Ses_{LC}$  is equal to  $Ses_i$ . In this case,  $h_i$  might have delivered messages without acknowledging them and, therefore,  $s_n$  takes  $seq_{NC}$  of  $h_i$  as an acknowledgment. In both cases, if  $h_i$  has no message to deliver which  $s_n$  discarded from its  $SBuffer$ , then  $s_n$  sends a *connect<sub>ACK</sub>* message to initialize the connection on  $h_i$ 's side ( $s_n.7$ ). On the other hand,  $s_n$  registers  $h_i$  if not registered (lines  $s_n.9 - s_n.10$ ) and starts the handoff procedure.

Figure 5 shows a handoff procedure, which is a message exchange composed of three phases:

- Phase 1: detection by  $s_n$  of discarded messages that  $h_i$  has not delivered. Messages  $Req_1$  and  $Rsp_1$  are exchanged between  $s_p$  and  $s_n$  in this phase.
- Phase 2: detection by  $s_n$  of messages that  $h_i$  has not delivered among messages that  $s_n$  caches. Messages  $Req_2$  and  $Rsp_2$  are exchanged between  $s_p$  and  $s_n$  in this phase.
- Phase 3: initialization of the connection between  $s_n$  and  $h_i$ .

**Phase 1:**  $s_n$  starts this phase by sending  $Req_1$  to the previous stations included in  $PS$  that might register  $h_i$  (line  $s_n.10$ ). Note that (1) all messages not delivered by  $h_i$  that  $s_n$  discarded were received by  $s_n$  before  $send(Req_1)$ , because  $s_n$  discards no message after the reception of  $h_i$ 's *connect* message unless  $h_i$  acknowledged it; (2) the FIFO dissemination approach on the wired network ensures that, upon reception of  $Req_1$ ,  $s_p$  receives all messages that  $s_n$  received before sending  $Req_1$ . Upon reception of  $Req_1$ , station  $s_p$  uses  $seq_{NC}$  piggybacked onto  $Req_1$  to identify the messages that  $h_i$  has not delivered yet (line  $s_p.8$ ), and sends their respective  $id$  via  $Rsp_1$ .

The FIFO dissemination approach on the wired network ensures that, upon receiving  $Rsp_1$ ,  $s_n$  received all messages that  $h_i$  could have delivered in previous connections.  $s_n$  might have already discarded

some of those messages.  $s_n$  identifies the messages it discarded but that  $h_i$  has not delivered (lines  $s_n.12$ ), and requests them in  $Req_2$  message. Then,  $s_p$  piggybacks these messages onto the  $Rsp_2$  message, keeping the same order as in its  $SBuffer$  (line  $s_p.13$ ).

Simultaneous handoff procedure instances for  $h_i$  are handled sequentially. If  $Req_1$  messages are received during a handoff execution, the one with the highest  $Ses$  value is kept pending provided that it is newer than the current handoff's  $Ses$  value. The station will handle that  $Req_1$  at the end of the current handoff (line  $s_p.20$ ).  $Req_1$  messages with lower  $Ses$  values concern handoffs of previous connection attempts and are discarded. At the end of the handoff  $s_n$  sends a *Delete* message containing the handoff's  $Ses$  value (line  $s_p.2$ ) to the stations of PS so that those stations unregister  $h_i$ .

**Phase 2:** The FIFO dissemination approach on wired channels guarantees that messages that  $h_i$  could deliver when connected to  $s_p$  are received by  $s_n$  at latest at  $Rsp_1$ , since  $Rsp_1$  was sent after that  $s_n$  sent *connect*, and  $h_i$  delivers no message from  $s_p$  after changing cell (after sending *connect*). Messages not delivered by  $h_i$  that  $s_n$  receives before sending  $Req_1$  are identified by  $s_p$  in  $Rsp_1$ . Phase 2 identifies messages that  $s_n$  receives between *send*( $Req_1$ ) and *receive*( $Rsp_1$ ) that  $h_i$  has not delivered.

The FIFO dissemination on wired channels also guarantees that messages received by  $s_n$  between *send*( $Req_1$ ) and *receive*( $Rsp_1$ ) are surely received by  $s_p$  between *receive*( $Req_1$ ) and *receive*( $Req_2$ ).  $s_p$  saves the id of such messages and sends this list of ids to  $s_n$  in  $Rsp_2$  message (lines  $s_p.9 - s_p.12$ ).

Upon reception of  $Rsp_2$ ,  $s_n$  determines which messages  $h_i$  has not delivered among those received before  $Rsp_1$ . The messages are those (1)  $s_p$  identified as not delivered by  $h_i$  (line  $s_p.8$ ), (2)  $s_p$  received between the receptions of  $Req_1$  and  $Req_2$  (line  $s_p.12$ ). On the other hand,  $h_i$  has already delivered all the other messages  $s_n$  received before  $Rsp_1$ . Moreover, messages that  $s_n$  receives after  $Rsp_1$  are not delivered by  $h_i$  because the FIFO dissemination approach on wired channels ensures that all messages received by  $s_p$  before it sent the  $Rsp_1$  message were also received by  $s_n$  before *receive*( $Rsp_1$ ).

**Phase 3:** Assigning a sequence number to  $h_i$  is not sufficient to identify all messages already delivered by  $h_i$ . For example, in Figure 4, if  $h_i$  delivered  $m_1$  before connecting to  $s_n$  but did not deliver  $m_2$ , then  $s_n$  must give to  $h_i$  the sequence number 0 ( $seq_{NC} = 0$ ), so that it will deliver  $m_2$ . However, without additional control,  $h_i$  would then also deliver  $m_1$  again. To prevent  $h_i$  of delivering the same messages twice,  $s_n$  adds  $h_i$ 's id to the messages  $h_i$  already delivered but which are still cached in  $s_n$ 's  $SBuffer$  (line  $s_n.14$ ), and, upon receiving them,  $h_i$  will only increment  $seq_{NC}$  and not deliver them again.

Moreover,  $s_n$  must send to  $h_i$  messages identified at Phase 2 that it already discarded but that  $h_i$  did not deliver.  $s_n$  sends these messages to  $h_i$ , ordered as in  $msg$ , also piggybacking ( $id_{h_i}, Ses_i$ ) that identify the handoff procedure instance. Once  $h_i$  acknowledged all of them, or if  $h_i$  has no message to deliver that  $s_n$  discarded ( $s_n.16$ ), then  $s_n$  concludes the handoff by sending a *connect*<sub>ACK</sub> message to  $h_i$  with  $seq_{C_i}$ , the sequence number of the oldest message in  $s_n$ 's  $SBuffer$  that  $h_i$  has not delivered (line  $s_n.18$ ). At its side,  $h_i$  ends the handoff by setting  $seq_{NC}$ ,  $seq_h$ , and  $Ses_{LC}$  (line  $h_n.4$ ).

## 4.4 Handoff example

Figure 5 presents the handoff executed when  $h_i$ , previously connected to  $s_p$ , tries to connect to  $s_n$  in the configuration of Figure 4. For better readability, we assume that no other handoff procedure takes place simultaneously for  $h_i$ . At the beginning of the handoff  $s_p$  has discarded  $m_1$ ,  $s_n$  discarded  $m_2$ , and  $h_i$  delivered  $m_1$ . Both stations receive  $m_3$  during the handoff.

Host  $h_i$  sends a *connect* message to  $s_n$ , which contains  $seq_{NC}=2$ , since  $h_i$  delivered  $m_1$ . Upon reception of the *connect* message,  $s_n$  sends  $Req_1$  to  $s_p$  with  $seq_{NC}=2$ . When receiving  $Req_1$ ,  $s_p$  concludes that  $h_i$  has not delivered  $m_2$  and replies with  $Rsp_1 = \{\{id(m_2)\}, seq_h = 1\}$  ( $seq_h=1$  since  $h_i$  already broadcasted  $m_1$ ).  $s_n$  requests then  $m_2$  via  $Req_2$ , since it has already discarded  $m_2$ .  $s_p$  replies with  $Rsp_2$  containing the requested message ( $\{m_2\}$ ) and the list of messages received between  $Rsp_1$  and  $Rsp_2$  ( $\{id(m_3)\}$ ). Finally,  $s_p$  unregisters  $h_i$ .  $s_n$  determines with  $Rsp_2$  which messages of its  $SBuffer=\{m_1, m_3\}$   $h_i$  has delivered.  $s_n$  received  $m_1$  before  $Rsp_1$ , and  $s_p$  did not identify  $m_1$  as not delivered by  $h_i$ . Hence,  $h_i$  already delivered  $m_1$  and  $s_n$  adds  $h_i$ 's id via  $m_1$ . In addition,  $h_i$  must first deliver  $m_2$  before delivering  $m_3$ . Thus,  $s_n$  resends  $m_2$  received from  $s_p$  with  $seq_C=0$  to  $h_i$ . Finally,  $s_n$  discards  $m_2$  once  $h_i$  acknowledged it, assigns  $seq_{C_i}=3$  to  $h_i$ , since  $h_i$  delivered  $m_1$  and  $m_2$ , and sends *connect*<sub>ACK</sub> to  $h_i$ . Upon reception of *connect*<sub>ACK</sub>,  $h_i$  sets  $seq_{NC}=3$ .

## 4.5 Host failures

Hosts can fail temporarily or permanently and stations handle such failures. The duration of temporal failures should be bounded in time. In fact, a station does not discard a message until all hosts of its cell, including the faulty one, have acknowledged the message. Hence, a long temporal failure would increase the number of messages a station broadcasts, which in turn would increase the station's cell's message collision rate. The latter would then decrease the throughput, which could reach the point where messages are delivered more slowly than new messages are broadcasted, degrading the cell forever. Thus, stations control their cell's collision rate and a station unregisters a faulty host before this rate becomes too high. A station considers that a host is faulty when it does not receive any message from the host during a given time interval, or if the message collision rate becomes too high due to the lack of acknowledgments from the host.

Few variables of the host's persistent local storage should be restored when it recovers from a temporal failure:  $seq_h$ ,  $seq_{NC}$ ,  $Ses$ , and  $Ses_{LC}$ , and unacknowledged messages broadcasted by the host before its failure. Note that the number of these messages is quite small since messages are acknowledged very fast. A host saves  $seq_h$  (resp.,  $seq_{NC}$ ) when broadcasting an application (resp., acknowledge) message,  $Ses$  when changing cell, and  $Ses_{LC}$  when the host receives the confirmation that the station registered it. Upon recovering, the host restores these variables and sends a *recover* message to the station of its cell. However, since a faulty host can move during its temporal failure or stay faulty for a long time, this station might not keep the host's causal information. If the station still registers the host, it replies to the host with a *connect*<sub>ACK</sub> message. Otherwise it broadcasts a *recovery*<sub>req</sub> to stations, which reply with either the host's causal information, or a message that notifies that they do not store the host's causal information. If

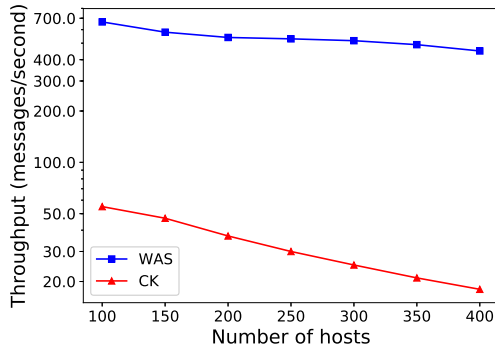


Figure 6: Throughput in function of hosts per cell

no station maintains the host’s causal information, the end of the recovery procedure is similar to the join procedure, otherwise to a cell changing.

Summing up, our algorithm tolerates permanent and transient failures, requiring few persistent information. Based on wireless message loss rate and timeouts, a station removes from its memory a host that permanently or temporarily failed. We point out that, inherent to wireless networks, interferences constrain the effectiveness of transient fault tolerance.

## 5 PERFORMANCE EVALUATION

**Experimental setup.** Experiments were conducted on INET, a network simulator implemented on OMNeT++ [21]. INET offers communication layers (e.g., TCP/UDP/Ethernet/IPv4/MAC), node mobility, node failures, and network interferences in wired and wireless networks.

Since no causal broadcast in mobile networks has been proposed yet, we compare our algorithm, denoted *WAS*, with the one proposed by Chandra and Kshemkalyani [6], denoted *CK*, which we have described in Section 6. We extended *CK*, which provides a causal multicast algorithm, to causal broadcast.

Stations are placed to ensure a complete area coverage with a minimum intersection of cells. Hosts are placed randomly in each cell at initialization. Antennas have a communication range of 120m and a bandwidth of 20Mb/s. Stations are connected by a wired network organized into a tree of degree 3 whose links have a bandwidth of 100Mb/s and a delay of 10ms. Application messages have a size of 100 bytes and are encapsulated in IPv4/MAC packets, whose header has 8 (resp., 20) bytes for UDP (resp., TCP). Therefore, an UDP (resp., TCP) packet has an overhead of  $20(\text{IPv4})+20(\text{MAC})+8(\text{UDP})=48$  (resp., 60) bytes. UDP is used for communication on the wireless network, while TCP on the wired network. Each host broadcasts application messages following a Poisson distribution. Hosts move in a straight line with a speed of  $5\text{km/h}\approx 1.39\text{m/s}$  inside the area covered by stations, and change direction every 5 seconds.

Experimental results concern three evaluations: (1) scalability comparison between *WAS* and *CK*, (2) comparison between a centralized and decentralized message discarding, (3) execution of *WAS* in a scenario with faulty hosts.

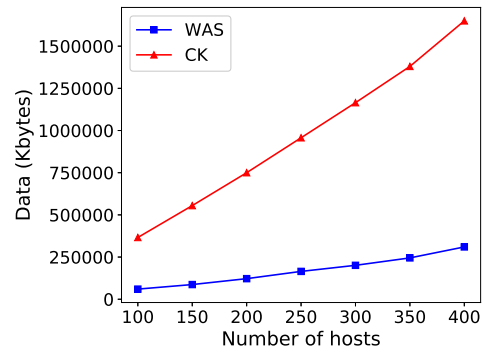


Figure 7: Sent data (Kbytes)

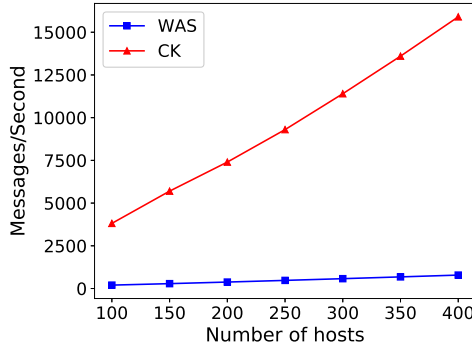
### 5.1 Scalability

**Throughput and delivery delay.** The first experiment, whose results are shown in Figure 6, evaluates the maximal throughput when the number of hosts per cell increases. The experiment contains 10 stations and a total number of hosts that varies from 100 to 400 (x-axis). Results, presented in a logarithmic scale, show that *WAS* has a much higher (x10-20) throughput than *CK*, and that the throughput of *CK* decreases faster than *WAS*. In a system containing 400 hosts, the maximal throughput of *WAS* is more than 20x higher than *CK*. The throughput of *CK* is bounded mostly by the fact that a station only sends an application message to a host once the latter has acknowledged all the message’s dependencies. Consequently, hosts send acknowledge messages very frequently, which negatively impacts performance because of a higher message collision rate on the wireless network. Moreover, *CK* has a delivery delay - the delay between broadcast(*m*) and deliver(*m*) - 2 times higher than *WAS*, because a station waits that a host acknowledges a message’s dependencies before sending it to the host.

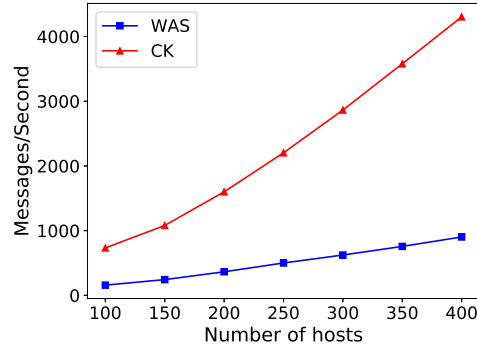
The second experiment measures the size and number of messages, evaluated at the IPv4 level, when the number of hosts increases. Results are shown in Figures 7 and 8 respectively. The number of stations is adapted to keep a ratio of 20 hosts/cell. 20 messages are broadcasted per second in the system.

**Number of sent messages.** Figure 8 shows that *WAS* sends much fewer messages on both the wired and wireless networks. Moreover, the number of messages sent by *CK* increases much faster than the number of messages sent by *WAS*. *WAS* sends fewer messages on the wireless network because (1) hosts acknowledge messages less often than *CK* (20x), (2) hosts buffer messages at reception following *seq<sub>NC</sub>* piggybacked on messages, contrarily to *CK* where hosts do not buffer messages since stations only send a message *m* to a host once the host can causally deliver *m*. Hence, stations must retransmit messages less often than with *CK* (x5). On the wired network, *CK* sends more messages due to its centralized approach to discard obsolete messages. *WAS* implements a decentralized mechanism which requires no message exchange to discard obsolete messages. We point out that *CK* sends much fewer messages on the wired network than theoretically expected, because acknowledge messages of *CK* are small, and TCP groups many of them in a single packet.





(a) Messages sent over the wireless network



(b) Messages sent over the wired network

Figure 8: Sent messages in the network

**Amount of sent data.** Figure 7 shows that WAS sends a lower amount of data than CK. On the wireless network, this is mostly due to acknowledge messages. Even though these messages contain only a few integers, they have an additional size of 48 bytes because they are encapsulated in UDP/IPv4/Mac packets. However, only a few acknowledge messages can be grouped into one single packet since the station will not send the next messages to deliver until the current ones are acknowledged. On the wired network, acknowledge and *delete* messages are grouped by TCP, which mostly removes the encapsulation overhead. However, those many acknowledge and *delete* messages scale up fast. Moreover, with CK stations piggyback a vector of size  $N$  ( $N$ =number of stations) on application messages sent over the wired network, and that vector rapidly takes much space when the number of stations increases. WAS only piggybacks a few integers on application messages.

## 5.2 Decentralized discard mechanism

This section compares our decentralized discard approach, used by WAS, with the existing centralized discard approach [6].

In the third experiment, 200 hosts are distributed over 10 cells, the wireless network has a bitrate of 1Mb/s, and 35 messages are broadcasted per second for 300s. Figure 9 shows the number of messages that stations store in their respective SBuffer. Curve *Max* shows the maximum number of messages cached in a stations' SBuffer, i.e., approximately the number of messages each station would store with a centralized discard approach. Curve *Avg* shows the average number of messages a station stores in its sending buffer, and curve *Deviation* gives the standard deviation between the average and the number of messages each station caches. The curves *Avg* and *Deviation* do not take into account the SBuffer of the station that stores the most messages, in order to compare both curves with the *Max* curve.

The comparison of curves *Avg* and *Max* of Figure 9 shows that the number of messages cached by stations can vary a lot. Such a variation depends on the message loss rate in the station's cell: the higher the message loss rate, the longer a station caches a message, since lost messages must be retransmitted. The message loss rate depends on the number of messages to broadcast as well as the position of hosts in the network. The probability of message collision is higher in areas where two cells overlap because the

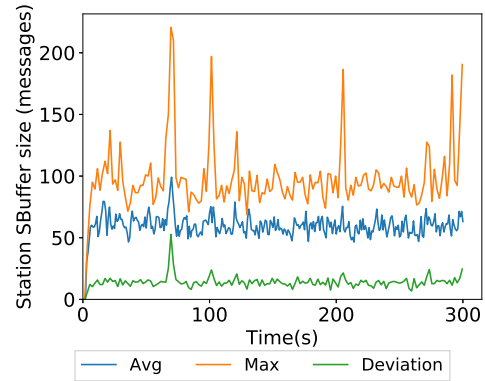


Figure 9: Messages in station send buffers

respective stations send messages over their cells that might collide. Similarly, areas with a high density of hosts have a higher message loss rate. The *standard deviation* is low, mostly around 10 messages, except for a short period around 70s where a heavily loaded cell degrades its adjacent cells. Hence, the number of messages a station stores in SBuffer is mostly close to the average for all stations, except for some stations whose local characteristics make their send buffer grow temporarily. In a decentralized message discard approach, message loss rate and failing hosts only have a local impact. The comparison of curves *Avg* and *Max* shows that with a decentralized message discard approach, stations store up to 4 times fewer messages than with a centralized one, and that, on average, stations store 40-50% fewer messages.

Finally, a host that fails stops acknowledging messages. Hence, the station which the host is connected to will stop discarding messages, and Figure 10 shows that, in presence of host failures, the station to which the faulty host is connected then caches many more messages than the other stations (8-10x more). Hence, the decentralized discard approach caches up to 8-10 times fewer messages on stations in presence of host failures.

### 5.3 Transient host failures

The last experiment, whose results are presented in Figure 10 measure the impact of transient host failures on WAS in a system containing 10 stations and 200 hosts (20 hosts per station), a wireless network bitrate of 1Mb/s, and where 15 messages are broadcasted per second. The first host fails at  $t=10s$  for 5 seconds, then each 30 seconds another host fails, and the fault duration increases by 2 seconds at each failure. In total, 9 hosts fail, the first failing at  $t=10s$  for 5s, the second at  $t=40s$  for 7s, and so on. When a host fails it stops acknowledging messages. The number of messages cached in the *SBuffer* of the station that registers the faulty host then grows. Hence, we measure the impact of transient failures through the number of messages cached in the *SBuffer* of that station.

Curve *Max* shows the maximum number of messages cached in a station's *SBuffer* which is, during failures, the number of messages cached by the station at which the faulty host is registered. Curve *Avg* shows the average number of messages a station stores in its *SBuffer*, and curve *Deviation* gives the standard deviation between the average and the number of messages each station caches. In order to evaluate the impact that a cell containing a faulty host has on the other cells, the former is not taken into account in the computation of *Avg* and *Deviation*. Vertical dashed lines represent a host crash.

During each failure, the number of messages cached by the station that registers the faulty host linearly increases. Those messages are also broadcasted by that station. Nevertheless, the number of cached messages sharply decreases once the host recovers, showing that, very fast, the host receives the missing messages and the cell rapidly reaches the same message load it had before the failure.

Curve *Avg* shows that, on average, a faulty host has a low impact in the number of messages stored by the other stations, except for the last failure occurring at  $t=255s$ .

Curve *Deviation* also shows that the increasing size of *SBuffer* of the faulty host's station has no impact on other cells, as long as the *SBuffer* does not become bigger than 150-200 messages. Once the *SBuffer* exceeds that size, the faulty host's station begins to degrade adjacent cells that overlap with it. In fact, the station broadcasts application messages and retransmits messages not acknowledged by the faulty host, thus increasing the number of messages sent by the station. This leads to an increasing number of message collisions in the cell, including the areas where the cell overlaps with other cells. Hosts in those overlapping areas, connected to other stations, will receive fewer messages, because of this higher message collision rate. Hence, they will deliver messages more slowly, increasing the number of messages broadcasted by the neighboring stations of the faulty host's cell. During the last crash occurring between  $t=250s$  and  $t=273s$ , the 3 adjacent cells of the faulty host's cell are impacted, explaining why the average size of the *SBuffer* increases. Moreover, contrarily to previous failures where the faulty host takes less than a second to acknowledge messages, the host takes 7 seconds to acknowledge all messages. Therefore, the failure of a host first and mostly has an impact in the cell in which it occurs and afterwards in adjacent cells when its cell's station stores more than 150-200 messages. Nevertheless, such an impact rapidly disappears once the host recovers.

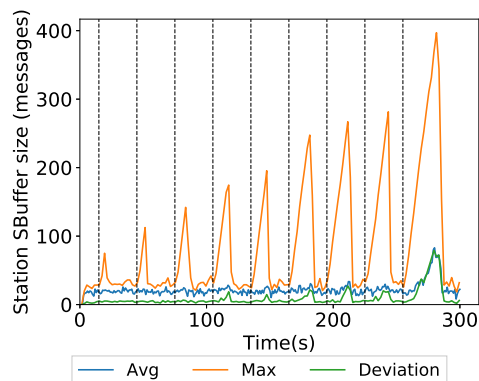


Figure 10: Messages stations cache when hosts fail

In conclusion, experiments confirm that WAS is more scalable than CK in terms of the number of hosts per cell, as well as in terms of total hosts and/or stations. WAS sends much fewer messages than CK both on the wired and wireless network, the amount of sent data is also much lower, and WAS has half the delivery delay of CK. Second, the decentralized message discard mechanism of WAS caches much fewer messages than CK, particularly during host failures. Finally, in WAS, the impact of a transient host failure is sharply absorbed after the faulty host recovers.

## 6 RELATED WORK

Charron-Bost proved [7] that without additional assumptions on the system, the size of the information required to causally order messages grows linearly with the number of nodes of the system. Therefore, solutions based on logical clock vectors [8][13][4] are not suitable to huge mobile networks since they are not scalable.

Plausible clocks [19], Bloom filters [18], and Probabilistic clocks [14] have constant size and ensure causal order with high probability. However, even though they greatly reduce causal control information in networks with a large number of nodes, messages might be delivered out of causal order.

Hierarchical approaches [1] group nodes into clusters and logically organizes them in a tree. Causal information is handled only at the cluster level. However, clusters must be reorganized in presence of nodes churn and a node moves only inside its own cluster. Hence, such a solution cannot be applied to mobile networks because hosts move between cells/clusters and join/leave the network.

Many causal dissemination protocols [5][16][15][20], exploiting message propagation and forwarding over reliable FIFO links on overlay networks, have been proposed in the literature. They do not require any causal information since messages are implicitly causally ordered at reception. A static tree is used in [5] to disseminate messages while, by using scalable data structures, [16][15] extend [5] protocol for dynamic systems. Nevertheless, the considered dynamics of the system cannot be applied to mobile networks, because every new channel must be initialized by message exchange through already initialized channels. Consequently, a path of initialized channels must always connect each pair of nodes, which is not always the case when a host moves to a new cell without having initiated the connection with the latter.

In [20], a reliable causal multicast is presented where nodes maintain only local metadata about their neighborhood, being, thus, scalable. The algorithm tolerates faults and outperforms other multicast causal algorithms with local views, but channels are assumed to be static.

Several works [6][17] [2] [3] [12] have proposed causal multicast for mobile networks.

Chandra and Kshemkalyani [6] consider a dynamic system composed of fixed stations and mobile hosts. For each connected host  $h$ , a station maintains information to control causal order on behalf of  $h$ . When a node  $h$  moves to a new station, its information is transferred from the previous station to the new one. A node can leave its previous station only after the new station has acknowledged the reception of such information which implies some constraints on the mobility model. Moreover, the algorithm implements a centralized message discarding of obsolete application messages, which has an overhead both in terms of stored messages in station  $SBuffer$  and number of messages sent over the wired network: each application message  $m$  is managed by a station called  $MSS_{init}$ , and acknowledge messages for  $m$  from each host are forwarded to  $MSS_{init}$ . Once  $MSS_{init}$  received an acknowledge message from each host, it broadcasts a delete message to all stations in order to discard  $m$ .

In [17], messages piggyback only direct dependencies. Such an approach copes with the dynamics of mobile networks since causal information does not depend on node identifiers. However, every node, including mobile hosts that have memory limitations, maintains a matrix of size  $N^2$ , where  $N$  corresponds to the maximum number of nodes in the system. Moreover, the information piggybacked on messages grows with the number of concurrent messages. In [2], the authors propose a multicast protocol that tolerates a dynamically changing system membership. On the other hand, host mobility is not supported by the algorithm, which does not implement any handoff procedure. Contrarily, [3] and [12] present causal multicast protocols that tolerate node mobility among stations, but both consider that wireless channels are FIFO and reliable, the connection protocol is reliable, and hosts do not fail, which are not realistic assumptions.

## 7 CONCLUSION

We have presented in this article a causal broadcast algorithm tailored to the features and dynamics of mobile networks. The latter include host mobility, dynamic host membership, unreliable dynamic wireless channels, memory and computing constraints of mobile hosts, scalability issues due to the high number of mobile hosts and stations, and mobile host failures. Messages piggyback few causal information. Mobile hosts have a low memory footprint while stations have a memory footprint that grows linearly with the number of locally connected mobile hosts. Stations discard obsolete messages with only local information, removing the message exchange between stations used by centralized message discarding approaches.

Performance results from simulations done on OMNeT++/INET show that our algorithm has a much lower message overhead, delivery delay, and caches fewer messages than a representative algorithm adapted to provide causal broadcast [6]. Furthermore, the

decentralized approach to discard obsolete messages of our algorithm results in much fewer messages cached by stations, as well as much fewer messages sent on the network compared to a centralized approach. Finally, performance results show that hosts that temporarily fail fastly catch up after recovering, and that the decentralized discard approach of obsolete messages mostly limit the impact of host failures to the cells in which those failures occur.

## REFERENCES

- [1] N. Adly and M. Nagi. 1995. Maintaining Causal Order in Large Scale Distributed Systems Using a Logical Hierarchy. In *IASTED Int. Conf. on Applied Informatics*. 214–219.
- [2] G. Anastasi, A. Bartoli, and F. Spadoni. 2001. A Reliable Multicast Protocol for Distributed Mobile Systems: Design and Evaluation. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1009–1022.
- [3] C. Benzaid and N. Badache. 2008. An Optimal Causal Broadcast Protocol in Mobile Dynamic Groups. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*. 477–484.
- [4] K. P. Birman, A. Schiper, and P. Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.* 9, 3 (1991), 272–314.
- [5] S. Blessing, S. Clebsch, and S. Drossopoulou. 2017. Tree topologies for causal message delivery. In *AGERE workshop*. 1–10.
- [6] Punit Chandra and Ajay D. Kshemkalyani. 2004. Causal Multicast in Mobile Networks. In *12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS*. 213–220.
- [7] B. Charron-Bost. 1991. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.* 39, 1 (1991), 11–16.
- [8] C. J. Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *11th Australian Computer Science Conference*.
- [9] R. Friedman and S. Manor. 2004. *Causal ordering in deterministic overlay networks*. Technical Report CS-2004-04. Technion - Computer Science Department.
- [10] Atta ur Rehman Khan, Mazliza Othman, Sajjad Ahmad Madani, and Samee Ullah Khan. 2014. A Survey of Mobile Cloud Computing Application Models. *IEEE Communications Surveys Tutorials* 16, 1 (2014), 393–413.
- [11] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. <https://doi.org/10.1145/359545.359563>
- [12] C. Li and T. Huang. 1999. A Mobile-Support-Station-Based Causal Multicast Algorithm in Mobile Computing Environment. *Proc. Natl. Sci. Coun. ROC(A)* 23, 1 (1999), 100–110.
- [13] F. Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel And Distributed Algorithms*. 215–226.
- [14] A. Mostéfaoui and S. Weiss. 2017. Probabilistic Causal Message Ordering. In *Parallel Computing Technologies - 14th International Conference, PaCT*. 315–326.
- [15] B. Nédelec, P. Molli, and A. Mostéfaoui. 2018. Causal Broadcast: How to Forget?. In *22nd International Conference on Principles of Distributed Systems, OPODIS*.
- [16] B. Nédelec, P. Molli, and A. Mostéfaoui. 2018. Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems. In *37th IEEE Symposium on Reliable Distributed Systems, SRDS*. 51–60.
- [17] R. Prakash, M. Raynal, and M. Singhal. 1996. An efficient causal ordering algorithm for mobile computing environments. In *16th International Conference on Distributed Computing Systems*. 744–751.
- [18] L. Ramabaja. 2019. The Bloom Clock. *CoRR* abs/1905.13064 (2019). <http://arxiv.org/abs/1905.13064>
- [19] F. Rojas and M. Ahamad. 1996. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *WDAG 1996*. 71–88.
- [20] Válder Santos and Luis Rodrigues. 2019. Localized Reliable Causal Multicast. In *NCA*. 1–10.
- [21] A. Varga. 2001. The OMNeT++ discrete event simulation system. *Proc. ESM'2001* 9 (2001).
- [22] D. Wilhelm, L. Arantes, and P. Sens. 2020. *A scalable causal broadcast that tolerates dynamics of mobile networks*. Technical Report. Sorbonne University, UPMC. <https://hal-upec-upem.archives-ouvertes.fr/hal-02652082>