Causal and Δ-Causal Broadcast in Opportunistic Networks

Frédéric Guidec, Pascale Launay, Yves Mahéo

Causal and Δ-Causal Broadcast in Opportunistic Networks

Frédéric Guidec[1,2], Pascale Launay[1,2], Yves Mahéo[1,2]

1 IRISA, Université Bretagne Sud, BP 573, 56017 Vannes Cedex, France
2 Laboratoire Cogitamus, France

***Abstract*—Causal broadcast is a fundamental communication abstraction for many distributed applications. Several implementations of this abstraction have been proposed over the last decades for traditional networks, that is, networks that assume the existence of a continuous bi-directional end-to-end path between any pair of nodes. Opportunistic networks constitute a kind of networks in which this assumption cannot be made, though, so the implementation of causal broadcast in such networks must be addressed differently. This paper presents two algorithms based on causal barriers that can ensure the causally-ordered delivery of broadcast messages in an opportunistic network, considering both cases where the messages propagate in the network without or with a bounded lifetime. The latter case is especially interesting in networks that must run for a long time, or with a population of nodes that changes continuously.**

I. Introduction

Causal broadcast is a high-level communication abstraction that ensures that messages broadcast in an asynchronous networking system are delivered in their broadcast causality order. More specifically, if a broadcast message *m* has been delivered to the application layer on node *u*, and if after that event node *u* broadcasts message *m'*, then *m'* is said to depend causally on *m*. Consequently, no network node can deliver *m'* to its own application layer unless it has delivered *m* first.

This abstraction is a major enabler for distributed applications, as it makes it possible for concurrent parts of an application to follow a consistent course of actions. In discussion forums or social networks, for example, the messages can be presented to a user in a causally consistent order, even though these messages may be received in any order.

The notion of causal order dates back to 1987, as it was first introduced in the ISIS system [1], [2]. Since then many algorithms have been proposed to ensure causal message delivery in either point-to-point or broadcast communication. Most of these algorithms rely on strong assumptions about the underlying network, though. For example it is often assumed that this network is connected (i.e., that there exists a bi-directional path between any pair of nodes), and that the number of nodes in this network — or at least those that need to exchange causally ordered messages — is known and is stable. Sometimes the topology of the network is assumed to be quite stable as well, so an overlay can be built and maintained in order to forward messages efficiently, especially in very large networks. Quite frequently it is assumed that reliable transmissions are ensured in the network, so causal message delivery is often built upon this property.

Overall, most of the causal broadcast algorithms that have been designed over the last decades are meant to run in the Internet, or in networks with similar properties as the Internet: the existence of a continuous bi-directional end-to-end path between any pair of nodes, low error rates, and reasonably short and roughly symmetric round-trip delays. Yet over the last few years a different kind of network has appeared, that does not share these properties.

An opportunistic network (or OppNet for short) is a network whose nodes are mostly mobile, and that operates solely by exploiting transient direct radio contacts between pairs of nodes [3]. Most of the times these contacts cannot be predicted in advance, so they must be exploited opportunistically whenever they occur, hence the opportunistic nature of communication in such a network.

Figure 1 shows a typical OppNet, in which the mobile nodes are handheld devices carried by human beings. These devices only come into contact with each other every now and then, depending on how their carriers move. In this figure the communicating devices are carried by people, but they may as well be carried by vehicles, by animals, by drones, or by any combination of mobile entities.

Although a radio contact between two nodes is only possible at close range in an OppNet, the transmission of messages over long distances is still possible in such a network, thanks to the "store, carry, and forward" principle: each mobile node can serve as a "data mule" for messages it has either produced itself or received recently, storing these messages in a local cache, and carrying them for a while before they can be forwarded to other nodes [4]. Thus, although an end-to-end path may not always exist between the sender of a message and the intended receiver(s), a *journey* may however exist between these sender and receiver(s). A journey is, basically, a succession of hops from one node to the next, each hop being only possible at the appropriate time, while two nodes are in radio contact.

Many message forwarding protocols have been proposed in the literature based on this general principle, each protocol addressing a particular kind of OppNet [5], [6], [7], [8], [9], [10]. The delivery of a message to a particular destination node is not always guaranteed, though, as it depends on how the nodes move and come into radio contact. Besides a message may take several minutes, hours, or even days to reach its destination. Again this depends on the particular dynamics of the whole network.

The population of nodes that constitute an OppNet is not

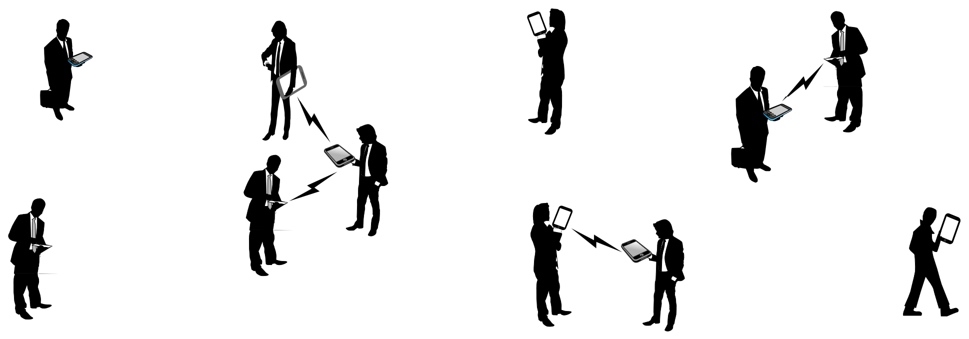


Figure 1: Example of an opportunistic network composed of handheld devices carried by human beings

always stable either. For example, in the OppNet depicted in Fig. 1, some people may occasionally shut their device off and thus leave the network (either temporarily or definitely), while others may enter the network at any time.

An OppNet is therefore a network where reliable communication cannot always be guaranteed, where the time required for a node to receive a message is usually not known either, and where even the number of nodes that constitute the network at any time may remain unknown.

Causal broadcast algorithms that have been designed for more “traditional” networks can hardly run in an OppNet. Yet an OppNet is a network in which applications have to be distributed, hence the need to support high-level communication abstractions such as causal broadcast in such a network.

In the remainder of this paper we propose two algorithms that can ensure causal broadcast in an OppNet. The first algorithm can be used when the broadcast messages have no bounded lifetime, while the second algorithm, called Δ-causal algorithm, takes care of messages with a bounded lifetime. Both algorithms are directly derived from [11] and [12] respectively, but they have been modified so as to take into account the peculiarities of opportunistic networks.

The remainder of this paper is organized as follows. Related work is presented and discussed in Section II. The system model we consider in this work is detailed in Section III. The causal and Δ-causal broadcast algorithms we propose are presented in sections IV and V respectively. Section VI presents experimental results we obtained while running both algorithms in realistic scenarios. Section VII concludes the paper.

II. Related work

Causal ordering has first been introduced in the ISIS system [1], [2]. The first approach implemented in this system consisted in piggy-backing in each message all its causal predecessors, that is, all the messages it causally depends on. This approach is effective, but not efficient: the communication overhead is large, so garbage collection is required every now and then to expunge old messages that are not significant anymore.

Vector clocks have then been introduced by Fidge [13] and Mattern [14] as a means to reduce the communication overhead [15]. Instead of piggy-backing all its causal predecessors (that is, all the messages that must be delivered before its own delivery), each message only carries a vector of logical timestamps. Each entry in this vector corresponds to a network or process, and each timestamp characterizes the last message received from that process by the sender of this message. Based on this meta-information, a process receiving a message can decide if it can be delivered immediately to the application layer, or if this delivery must be deferred until all its causal predecessors have been delivered locally.

The main advantage of using vector clocks is that their implementation can be quite compact: the vector size is O(N), where N is the number of processes that participate in group communication). The disadvantage is that they can only be used when N is known by every process, so each process can be assigned an integer number as an identifier. This is only possible, either when N is known in advance and remains static during the network’s entire lifetime, or when an additional group membership algorithm is used to manage the population of processes.

Although a vector clock is already a rather compact data structure, several vector compaction methods have been proposed in later years. For example, the idea proposed in [15] consists in piggy-backing in each broadcast message only information about what has changed in the sender’s vector clock since its last broadcast. The authors of [16] propose a general method using clock matrices instead of vector clocks, which can be used for both point-to-point and broadcast communication.

The causal broadcast algorithm defined in [11] defines the notion of *causal barrier*, which stems from the observation that when a message is sent in the network, there is no need for this message to carry information about all its causal predecessors. Instead a message only needs to carry information about its immediate predecessors, that is, the messages on which its delivery *directly* depends on. This minimal information constitutes the so-called causal barrier of a message, which can be piggy-backed in the broadcast messages at minimal cost.

The work described in [12] is also based on causal barriers. It proposes a special type of causal broadcast, called Δ-causal broadcast, that can be used when the broadcast messages have a bounded lifetime Δ. A message can then only be delivered before its lifetime expires. Otherwise this message is considered lost, and any causal dependency on this message is considered canceled.

More recently, several new algorithms have been proposed to support causal broadcast in different kinds of networking systems. For instance, causal broadcast in cellular networks is considered in [17], assuming that the wireless channels between mobile support stations and mobile hosts are reliable and FIFO.

A probabilistic causal message ordering mechanism to be used in large dynamic networks is defined in [18]. This approach reduces the cost of causal broadcast, while admitting a small (and controllable) rate of errors. It is assumed that a reliable broadcast primitive is available, though.

The new algorithm detailed in [19] also targets large dynamic systems. In this algorithm it is assumed that the processes communicate via an overlay network, whose links are FIFO. Besides each process is assumed to be aware of the links through which it can reach other processes.

The problem of the accumulation of control information in causal broadcasting is addressed in [20]. The authors propose a method whereby each process can forget obsolete control information about the messages it has already delivered. Since this algorithm does not assume that the underlying system ensures reliable broadcast, the focus is on preventing multiple deliveries of the same message. Like in [19], though, it is assumed that the processes communicate via the FIFO links of an overlay network.

Finally, the case where processes can crash is explored in [21]. An algorithm is presented that can tolerate any number of crashes in the system. It is assumed that the transmission channels are reliable, which notably means that messages are assumed never to get lost.

As mentioned in Section I, an OppNet is a network in which transmissions can only rely on pairwise radio contacts between mobile nodes. As a general rule a node cannot predict when a radio link is going to appear or disappear between itself and another node. Besides, the propagation of messages all over the network relies on the store, carry and forward principle. As a consequence, how a message will propagate cannot always be predicted. Finally, the number of nodes in the network is usually not determined in advance, since OppNets are typically networks whose population of nodes can vary over time.

For all these reasons, most of the above-mentioned causal broadcast algorithms could hardly be used in an OppNet. Algorithms based on vector clocks (or any variant thereof) require that the size of the population of nodes (or processes) be known in advance. Incidentally they also require that each node (process) be uniquely identified by an integer number, which is not always feasible in a network whose nodes are not necessarily administered by any central authority. Algorithms that assume reliable (or FIFO) communication links cannot be used either in OppNets, because message delivery in such networks cannot always be guaranteed, and because a node may receive messages in any order.

Among these algorithms, though, those based on causal barriers present interesting features. Their implementation does not require any additional control messages, and the meta-information (i.e., the causal barrier) piggy-backed in each broadcast message is minimal. These algorithms are usually presented as requiring no prior knowledge of the network or communication topology. Yet, the algorithms presented in [11] and [12] both require that each process maintains data structures that are defined as arrays (or vectors) of N entries, N being the number of processes involved in causal broadcast. In other words, both papers assume that N is known, and that each process is identified by a unique integer value taken in the range [0..N-1]. This approach is actually quite similar to that used in algorithms based on vector clocks, although the meta-information piggy-backed in the broadcast messages is different.

The notion of Δ-causal broadcast addressed in [12] can be especially interesting in OppNets, since assigning messages a bounded lifetime is common practice in such networks. This is a convenient means of preventing the saturation of the message cache maintained by each node.

Overall, causal barriers and Δ-causality are interesting features that fit the needs of OppNets perfectly. However, the algorithms presented in [11] and [12] cannot be implemented directly in OppNets, for they assume too much knowledge of the network, and they could not be used in an OppNet with a high churn ratio. The two algorithms presented in the remainder of this paper are definitely inspired from [11] and [12], but special attention has been paid to removing these limitations.

To the best of our knowledge, causal broadcast in OppNets has so far only been addressed in [22] and [23]. In [22] causal broadcast is obtained with an algorithm that is derived from the CBCAST algorithm [15], and in [23] an algorithm derived from [11] is used instead. In both papers, though, the focus is on supporting conflict-free replicated data types (CRDTs) in OppNets, so very little detail is provided about how causal broadcast is actually implemented. Besides, Δ-causal broadcast is not considered in any of these papers.

III. System model

The two algorithms defined in the next sections are meant to run on top of an opportunistic networking layer, and under an application layer that requires broadcast messages to be causally ordered. The opportunistic networking layer is assumed to run according to the *store, carry, and forward* principle [4], ensuring the dissemination of messages over the whole network. The following lines explain what is expected from this networking layer, and how the causal broadcast layer can itself interact with the application layer.

- The opportunistic networking layer provides a function *oppnet_id()* that returns the unique identifier of the local host in the opportunistic network. This identifier is of type ID_TYPE. On a smartphone it may for example be the IMEI of the device, or a self-assigned IPv6 address. In any case we do not assume that this identifier is an integer value. Likewise, we do not assume that the number of hosts in the network is known, or that this number remains stable during the network's lifetime. As a consequence, the data structures each host needs to maintain to ensure causal ordering cannot simply be based on arrays or vectors. Instead the algorithms described

below rely on maps of <*key, value*> pairs, where the key appears at most once and may be either the identifier of a host, or that of a message. This is a major difference with algorithms that rely on vector clocks or similar structures.
- Each message carries a unique identifier, which is produced by the opportunistic networking layer on the sender side. Since our causal broadcast algorithms are meant to be implemented over an existing opportunistic networking layer, we cannot assume that we can freely assign an identifier to a message. Instead we must do with the identifiers produced by the networking layer. However we assume that we can freely add meta-information to each message before it is sent in the network.
- The opportunistic networking layer maintains a cache in which any message produced locally or received from the network is systematically deposited. The capacity of this cache is of course limited, so the networking layer must implement a strategy to reclaim some space when the cache is full. However, since storing messages while moving around is a prime requirement for mobile hosts in an opportunistic network (as part of the *store, carry and forward* principle), we assume that the cache's capacity is such that messages can stay there long enough to disseminate properly in the network, and to be of some use to high-level services that rely on the networking layer (such as our causal broadcast algorithm).
- The opportunistic networking layer provides a function *oppnet_broadcast()* that makes it possible to broadcast a message in the network, and a notification function *oppnet_receive()* that is called whenever a new broadcast message has been received from the network. We make no particular assumption about how a message is actually broadcast. Epidemic forwarding is typically an effective method to broadcast information in an OppNet [24], [25], but other methods may be used, for example in networks where mobility or contacts follow regular patterns.
- The opportunistic networking layer takes care of duplicates: the same message may be received several times from the network, but it will be put in the local cache only once, and function *oppnet_receive()* will therefore be called only once (just after the message has been deposited in the cache).
- Interaction between the causal broadcast algorithm and the application layer is based on functions *co_broadcast()* and *co_deliver()*. Function *co_broadcast(m)* can be called by the application layer to request the broadcast of message *m*, and function *co_deliver(src, m)* is called when a message *m,* which has been sent by node *src,* is fit to be delivered to the application layer.

It is worth highlighting that we do *not* assume reliable broadcast in the OppNet, because in such a network there is usually no guarantee that a broadcast message will eventually reach every possible node. In fact, since OppNets are dynamic by nature, reliable broadcast is only possible in some of them, and sometimes only during specific time intervals. Achieving reliable broadcast therefore depends on the dynamics of the network considered. The algorithms described below are meant to ensure that the messages delivered to the application layer of a node are causally ordered, but they cannot ensure that all messages are delivered.

Terminology: In papers that deal with opportunistic networking, the term *message delivery* is often used to denote the event where a message is delivered to a network node, that is, the event where the message is actually *received* by a node. In this paper we must distinguish between the event where a message is received by a node, and the event where the message can be delivered to the application layer on this node, with causal order preserved. In order to prevent any confusion between these two kinds of events we use the term *co-deliver* (which stands for causally ordered delivery) in the remainder of this paper to denote the event when a message is delivered to the application layer.

IV. A CAUSAL BROADCAST ALGORITHM FOR OPPORTUNISTIC NETWORKS

In this section we do not consider the case where broadcast messages are given a bounded lifetime when they are sent in the network. This case will be addressed in the next section.

The algorithm described below is strongly inspired from [11], but it has been designed so as to fit the system model described earlier.

A. Data structures maintained by each mobile host

- *st:* this is an integer variable that serves as a sequence tag[1] for messages *co_broadcast* by this host;
- *c_barrier:* this is a registry of messages that are immediate predecessors of the next message that will be *co_broadcast* by this host. This registry is actually a map of <*src,st*> pairs, where each pair identifies one immediate predecessor.
- *co_delivered:* this is a registry of messages that have been co-delivered locally. This registry is also structured as a map of <*src, st*> pairs, each pair indicating the sequence tag *st* of the last message from *src* that has been co-delivered on the local host.
- *pending:* this is a registry of messages that have been received by the local host, but not co-delivered to the application layer yet (because they still depend on as-yet-undelivered messages). It is structured as a map of <*mid, cb*> pairs, where *mid* is the unique identifier of a pending message, and *cb* is the current state of the causal barrier of this message. This causal barrier is updated whenever an immediate predecessor of this message is co-delivered locally, and once the barrier is empty the message can itself be co-delivered.

Code 1 shows the definition of these data structures.

B. Algorithm

Let us first consider what happens when the application layer needs to broadcast a new message.

[1]We deliberately avoid using the term "sequence number" here, because this tag will be assigned date values in Section V.

Code 1 Definitions and initialization of data types

```
  def ID_TYPE: String
  def CB_TYPE: Map<ID_TYPE, long int>

  Initialization:
     static ID_TYPE host_id ← oppnet_id()
     long int st ← 0
     CB_TYPE c_barrier ← ∅
     CB_TYPE co_delivered ← ∅
     Map<mid, CB_TYPE> pending ← ∅
```

Code 2 Function *co_broadcast()*

```
Function co_broadcast(m):
  st ← st + 1
  oppnet_broadcast(<host_id, st, c_barrier, m>)
  c_barrier ← ∅
  performDelivery(host_id, st, m)
```

Code 4 Function *forgetDependency()*

```
Function forgetDependency(src, $st_{src}$):
  for all <mid, cb> in pending do
    if ∃ <src,st'> in cb s.t. st' $\leq$ $st_{src}$ then
      cb.remove(src)
    fi
  done
```

Code 5 Function *oppnet_receive()*

```
  Upon oppnet_receive <mid, src, $st_{src}$, cb, m> do
    expungeDeliveredPredecessors(cb)
    if cb $\neq \varnothing$ then
      pending.put(mid, cb)
    else
      performDelivery(src, $st_{src}$, m)
      checkPendingMessages()
    fi
```

When function *co_broadcast(m)* (see Code 2) is called, the sequence tag is first incremented, and the message is then broadcast by calling *oppnet_broadcast()*. The message that is actually broadcast is a combination of *m* with the local host's id, the new value of the sequence tag, and the current state of the local causal barrier. Once the message has been broadcast, the *c_barrier* registry is reset, and function *performDelivery()* is called in order to co-deliver the message locally.

Function *performDelivery()* (see Code 3) is used to co-deliver any message to the application layer, whether this message has been produced locally or received from another host. When this function is called, the *co_delivered* registry and the *c_barrier* registry are both updated in order to reflect the co-delivery of this message (i.e., the sequence tag of the last message co-delivered from source *src* is now st_{src}). Function *co_deliver()* is called to co-deliver the message (with indication of its source) to the application layer, and function *forgetDependency()* is called to update the pending registry accordingly.

Whenever a message *m* (identified as pair <*src*, st_{src}>) is co-delivered, function *forgetDependency()* is called to expunge any dependency on that message from the causal barriers of all the messages that are waiting in the pending registry. Basically, if the causal barrier of a pending message *m'* contains a pair <*src, st'*> such that $st' \leq st$, then it means that *m'* was waiting for *m* to be co-delivered. Now that *m* has been co-delivered, the dependency of *m'* on *m* can be removed from the causal barrier.

Any causal dependency is thus removed from the causal barriers of pending messages whenever possible. When the causal barrier of a pending message is empty, it means that this message is fit to be co-delivered.

Code 3 Function *performDelivery()*

```
Function performDelivery(src, $st_{src}$, m):
  co_delivered.put(src, $st_{src}$)
  c_barrier.put(src, $st_{src}$)
  co_deliver(src, m)
  forgetDependency(src, $st_{src}$)
```

Let us now examine what happens when a broadcast message is received. Function *oppnet_receive()* is called by the opportunistic networking layer as soon as a new broadcast message has been received and put in the local cache (see Code 5). The causal barrier contained in this message is first updated based on the current state of the *co_delivered* registry. This update is performed by calling function *expungeDeliveredPredecessors(cb)* (see Code 6), which basically removes from the causal barrier *cb* (passed as a parameter) any pair <*src, st*> such that <*src, st'*> exists in registry *co_delivered,* and $st' \geq st$. In other words, this update consists in expunging from the causal barrier any dependency on a predecessor that has already been co-delivered locally. Once the causal barrier of a newly received message has been updated, this barrier is examined. If it is not empty yet, then the co-delivery of the message must be deferred. In that case an entry <*mid, cb*> is added to the *pending* registry, where *mid* is the unique identifier of the message (this is the id that will later make it possible to retrieve this message from the local cache), and *cb* is the current state of its causal barrier. If the causal barrier of the message is empty, then function *performDelivery()* is called to co-deliver the message to the application layer. The co-delivery of this message may have an impact on the deliverability of other messages that were specifically waiting for this message to be co-delivered. Function *checkPendingMessages()* (see Code 7) is therefore called to evaluate the consequences of this co-delivery. Basically, this function looks in the pending registry for messages whose causal barrier is now empty (and which can thus be co-delivered). Each entry with an empty causal barrier is removed from this registry, the actual message is retrieved from the cache, and this message is co-delivered to the application layer. Note that this co-delivery may in turn render other waiting messages deliverable. This is the reason why function *checkPendingMessages()* defines a loop that runs until no further message can be co-delivered for the time being.

Code 6 Function *expungeDeliveredPredecessors()*

```
Function expungeDeliveredPredecessors(cb):
  for all <src, $st_{src}$> in cb do
    if $\exists$ <src,st'> in co_delivered s.t. st' $\geq$ $st_{src}$ then
      cb.remove(src)
    fi
  done
```

Code 7 Function *checkPendingMessages()*

```
Function checkPendingMessages():
  repeat
    msg_delivered $\leftarrow$ False
    for all <mid, cb> in pending do
      if cb $= \varnothing$ then
        pending.remove(mid)
        <src, $st_{src}$, m> $\leftarrow$ cache.extract(mid)
        performDelivery(src, $st_{src}$, m)
        msg_delivered $\leftarrow$ True
      fi
    done
  until not msg_delivered
```

C. Considerations about communication and storage overheads

An interesting consequence of the presence of a cache in the opportunistic networking layer is that, in our causal broadcast algorithm, we do not have to care about preserving messages until they can be co-delivered to the application layer. We only have to maintain enough information to be able to retrieve a message from the cache whenever necessary. The registries mentioned in the former section therefore do not actually store any message. They only maintain meta-information that makes it possible to ensure the causal ordering of message deliveries, and to retrieve a message from the cache when its co-delivery is actually required. The data space required to maintain these registries therefore remains quite negligible compared to the data space occupied by the cache itself.

More specifically, the size of the causal barrier is proportional to the number of immediate predecessors of the next message that will be broadcast locally. In the worst case a host may receive at least one message from each possible source between two of its own successive broadcasts, and in that case the size of the causal barrier would be proportional to the number of (other) hosts in the network. In practice, though, the causal barrier will most of the time be far smaller than that maximal size, as it is reset every time a new message is broadcast locally.

The *co_delivered* registry typically contains one entry for each source from which a message has been received in the past. Its size is therefore proportional to the number of (other) hosts in the network.

The *pending* registry only maintains information about messages that have not been co-delivered yet, so its size depends on the actual ordering of message receipts and co-deliveries on each host. As soon as a message is co-delivered, though, the corresponding entry is removed from this registry, so there is no risk to see its size growing continuously.

V. A Δ-causal broadcast algorithm for opportunistic networks

A. Motivation

An opportunistic network may run for a very long time, and possibly continuously. Since the dissemination of messages in such a network relies on the *store, carry and forward* principle, the cache each node maintains to store messages is likely to saturate after a while. Several methods have been proposed in the literature to address this problem, for example by dropping messages randomly, or based on their assumed utility. The most common method consists in dropping messages after they have disseminated in the network for a certain amount of time. When it is sent in the network, each message is allowed a certain lifetime, after which all copies of this message are removed from the caches where they have been stored. The problem is thus for the designer of an application, or for the administrator of the network (assuming there is indeed an administrator), to determine how long a message should be allowed to propagate in the network. Sometimes the lifetime of a message may be determined by the very nature of the information it contains: a message transporting for example the weather forecast for tomorrow does not need to keep propagating in the network the day after tomorrow. Sometimes this lifetime may be determined based on insights about how long it usually takes for a message to reach its destination(s): if statistics show that the messages propagating in a certain opportunistic network always reach any possible destination in less than two hours, there is no need to give each message a lifetime of several days.

Δ-causal ordering is a communication abstraction designed for distributed applications whose messages have a bounded lifetime but must still be co-delivered according to causal ordering. Once a message m_1 has reached the end of its lifetime, the information it contains is considered as irrelevant for the application layer, so it can safely be discarded. The consequence for any other message m_2 whose co-delivery depends on that of m_1 is that once m_1 has been discarded, m_2 can be considered as not being dependent on m_1 anymore.

With Δ-causal ordering, the idea is to co-deliver as many messages as possible before their lifetime is over, while respecting the causal order of the messages that are actually co-delivered.

A Δ-causal broadcasting algorithm based on the notion of causal barrier has been presented in [12]. This algorithm resembles the one presented in [11] (which does not account for bounded lifetimes), but instead of tagging each message with a sequence number, it tags it with its sending date, with the assumption that two messages sent successively by the same process cannot have the same sending date. When a message is received by a node, its deadline is calculated based on its sending date and its lifetime, and if this deadline is over the message is discarded. In [12] it is assumed that all nodes can access a common clock, though. This is a strong assumption that does not always hold true in an OppNet. Sometimes the mobile nodes include a GPS receiver, which can serve as a common time reference in the network. This is typically the case in a network whose nodes are carried by vehicles. But in other kinds of networks, such as those whose nodes must run indoor or with a very limited power-budget, relying on GPS receivers is not an option. As a general rule, it can however be assumed that each node has a local clock, and that all clocks are "reasonably" in sync with each other. The time returned by two different clocks may thus differ by a few seconds, but this difference is negligible considering the transmission delays observed in an OppNet (sometimes up to a few minutes or hours). When Δ-causal ordering is expected in broadcast messages, the question is mostly to determine if two nodes with slightly divergent clocks may take decisions that would violate the causal ordering of the messages co-delivered to their application layer. Let us consider the case where two nodes u and v receive the same message m, whose deadline is almost over. If node u has a clock that is a bit late compared to that of node v, then node u may consider that m is still valid and co-deliver it to its application layer. Node v however may observe that m is actually expired, and simply discard it. Thus, nodes with different clocks may take different decisions regarding how to process the same message. But this observation does not mean that a node may co-deliver messages in an order that is not causally consistent. Since each node takes decisions based on its own clock, the order in which it co-delivers messages to its application layer remains causally consistent, even though the messages it actually co-delivers may not be the same as those delivered on another node.

B. Additions to the system model

The system model for our Δ-causal broadcasting algorithm is similar to the one described in Section III, with the following additional assumptions:

- Function *oppnet_broadcast(m, [dln])* admits an optional parameter *dln* that defines the deadline of message *m*. This information is transported with the message (as meta-data), and the networking layer automatically takes care of messages when they expire. More specifically, when a message is received, its deadline is compared to the current time (as returned by the local clock), and if this deadline is over the message is discarded rather than being deposited in the local cache.
- The networking layer spontaneously expunges any message whose deadline is over from the local cache, and before doing so it invokes the notification function *oppnet_discard(mid)* in order to inform any higher-level algorithm that message *mid* is expiring.

C. Algorithm

The Δ-causal broadcast algorithm described below is strongly inspired from [12], but it has been designed so as to fit our system model.

Since this algorithm has the same general architecture as the one presented in Section IV, we only highlight in this section the lines that are either different or new in this algorithm.

Code 8 Definition of the lifetime of messages

```
Initialization:
  static long int lifetime ← ...
```

Code 9 Function *co_broadcast()* for Δ-Causal broadcasting

```
Function co_broadcast(m):
  st ← NOW + lifetime
  expungeExpiredPredecessors(c_barrier)
  oppnet_broadcast(<host_id, st, c_barrier, m>, st)
  c_barrier ← ∅
  performDelivery(host_id, st, m)
```

The lifetime assigned to messages is of course a parameter of this algorithm (see Code 8). In this example we assume that the lifetime variable takes a static value, but this value may actually change over time, for example to give messages a longer lifetime when they are broadcast at the end of the day, or just before the week-end.

When a message is *co_broadcast* the sequence tag it carries is not a sequence number anymore, but a date. In [12] this date is the sending date of the message, so it is up to each receiver to calculate the message's deadline based on this sending date and on the lifetime of the message. In our algorithm the deadline is calculated on the sender side. This is a very minor difference with [12], and it has no side effect on the behavior of the algorithm.

Before the message is sent in the network, function *expungeExpiredPredecessors()* is called in order to clean up the causal barrier of the sending node. Indeed, this causal barrier is meant to specify what are the immediate predecessors of the message that is being sent. But since the identities of these predecessors may have been put in the *c_barrier* registry some time ago, some of them may have already reached their deadline. There would be no advantage in sending a message with a causal barrier that indicates that it depends on predecessors that are already expired. Function *expungeExpiredPredecessors()* therefore cleans up this causal barrier by removing any dependency on an expired predecessor. The consequence of calling this function before broadcasting a message is that the causal barrier embedded in this message may be slightly smaller. The gain, if any, is thus observed in the communication overhead.

Function *expungeExpiredPredecessors()* is also called whenever a message is received from the networking layer, in order to clean up the causal barrier of this message from any reference to an expired predecessor (see Code 11). Remember that in an OppNet the journey of a message from a sender

Code 10 Function *expungeExpiredPredecessors()* for Δ-Causal broadcasting

```
Function expungeExpiredPredecessors(cb):
  for all <src, $st_{src}$> in cb do
    if $st_{src}$ < NOW then
      cb.remove(src)
    fi
  done
```

Code 11 Function *oppnet_receive()* for Δ-Causal broadcasting

```
  Upon oppnet_receive <mid, src, $st_{src}$, cb, m> do
    expungeExpiredPredecessors(cb)
    // Remainder of the code as shown in Code 5
```

Code 12 Function *forgetExpiredMessages()*

```
Function forgetExpiredMessages():
  for all <mid, cb> in pending do
    expungeExpiredPredecessors(cb)
  done
  for all <src, $st_{src}$> in co_delivered do
    if $st_{src} <$ NOW then
      co_delivered.remove(src)
    fi
  done
```

to a receiver may take minutes, or even hours. So it makes sense, when a message is received, to check if some of its predecessors have not expired during its journey.

Note that when function *oppnet_receive()* is called there is no need to check if this message has reached its deadline, for this has already been verified by the opportunistic networking layer. Any message that comes from the networking layer is thus a message whose deadline has not been reached yet.

Information pertaining to expired messages can also be expunged on occasions from the *pending* registry and the *co_delivered* registry. This is achieved in function *forgetExpiredMessages()* (see Code 12), which is called whenever the algorithm must check if pending messages can now be co-delivered (see Code 13).

When a message that is present in the cache expires, the networking layer calls function *oppnet_discard(mid)* before actually removing this message from the cache. The algorithm therefore checks if this message that is expiring was registered in *pending* (which means it has not been co-delivered yet). If that is the case, then the corresponding entry is removed from the *pending* registry, meta-information (namely the <*src*, st_{src}> pair) about this message is extracted from the cache, so function *forgetDependency()* can be called to expunge any dependency on that message from the pending registry. Finally, function *checkPendingMessages()* is called because some of the pending messages may be deliverable now that they do not depend on <*src*, st_{src}> anymore.

D. Considerations on communication and storage overheads

Since the registries maintained by a mobile host have the same structure and the same role as in Section IV, the data space they occupy is bounded by the same limits. The causal barrier transmitted with every message may be slightly smaller on average, though, for references to expired predecessors are systematically expunged from this barrier just before the message is sent in the network. The *pending* registry should also occupy less data space on average, because references to expired messages are expunged from the causal barriers it contains (one for each pending message) whenever function *checkPendingMessages()* is called. Finally, the *co_delivered* registry may grow and shrink over time, while it can only grow or keep the same size in the algorithm presented in Section IV.

Adjusting the size of the *co_delivered* registry dynamically is actually of prime importance when Δ-causal broadcast is required in an opportunistic network that exhibits a high churn ratio. Indeed, in a network whose population of nodes is renewed continuously, a long-standing node could endlessly receive messages from sources it has so far never heard of. Function *forgetExpiredMessages()* (see Code 12) helps to prevent the *co_delivered* registry from growing endlessly, by removing from this registry any entry that is not useful anymore.

Code 13 Function *checkPendingMessages()* for Δ-Causal broadcasting

```
Function checkPendingMessages()
  forgetExpiredMessages()
  // Remainder of the code as shown in Code 7
```

VI. Experimentation

Evaluating the performance of a distributed algorithm designed for OppNets is always a challenge, because the behavior of this algorithm is directly dependent on the dynamics of the network considered. In the literature, protocols and higher-level algorithms are often evaluated using plain simulators, which simulate the movement or the radio contacts between mobile devices, but with which the communication protocols and distributed algorithms are often over-simplified. The validity of results thus obtained is therefore debatable, as the simulation conditions can always be deemed as not realistic enough.

In order to mitigate this problem, we implemented the algorithms described in sections IV and V over a fully-functional opportunistic networking system that can run equally well in a real setting or in simulated conditions. This system, called DoDWAN, is implemented in Java, and it can notably run on Android smartphones, laptops, and tablets [26], [27]. DoDWAN essentially supports the dissemination of messages in an OppNet, using the epidemic forwarding model. By implementing our causal and Δ-causal broadcast algorithms over this platform, we ensure that they can indeed rely on the opportunistic communication services offered by a real opportunistic system.

Besides running on real mobile devices, DoDWAN can also be used with the LEPTON emulation platform [28]. With LEPTON, many instances of an opportunistic networking system (DoDWAN in that case) can run concurrently on a single workstation (or on a cluster of workstations if needed), while the radio contacts between these virtual nodes are simulated by the emulation platform. With this approach the same mobility or contact scenario can be replayed as many times as needed, and this scenario can involve hundreds or thousands of nodes, which would be intractable in real conditions.

A. Causal and Δ-causal broadcast in a city bus network

In order to evaluate the algorithms presented in sections IV and V, let us first consider a scenario in which the mobile

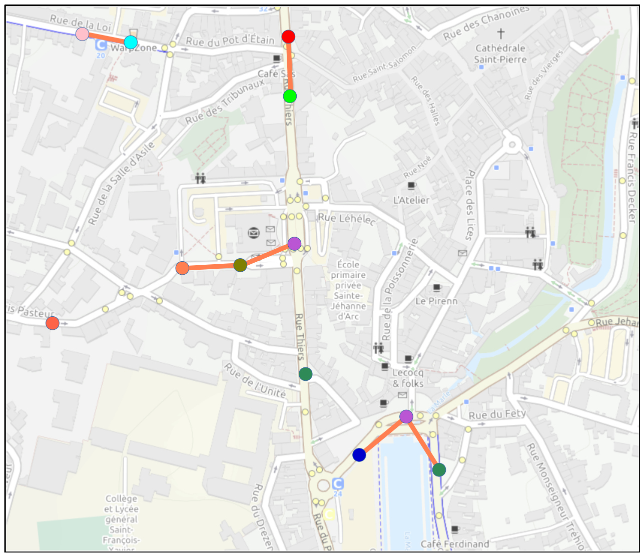


Figure 2: Snapshot of the city bus network scenario (close-up)

Metrics	Values (* = min / max / avg / stdev values)
Duration of the scenario	14h15’
Nb of nodes	60 (total) / 56 (max. active simultaneously)
Activity duration per node	26’ / 14h08’ / 10h58’ / 3h26’ (*)
Number of contacts	8.892
Durations of contacts	1” / 04’09” / 1’24” / 1’05” (*)

Table I: Statistics about the city bus scenario

nodes are city buses.

a) Mobility and contacts scenario: In order to simulate the mobility of these buses, and radio contacts between them, mobility traces have been produced by extracting the characteristics of the bus routes of a real bus network from OpenStreetMap, and combining these routes with the bus timetables. In that case we considered the bus network of the city of Vannes (France). Based on this mobility scenario, radio contacts have been calculated, assuming a transmission range of 200 meters around each bus. The result is a mobility and contacts scenario that can be played by LEPTON. A snapshot collected while running this scenario is presented in Figure 2, and a video is available on our website[2].

This scenario involves 60 buses covering the city, between 05:30 in the morning and 19:45 in the evening. These buses neither start nor stop running at the same time, as can be observed in Figure 3. Besides, a few of them only run at specific periods during the day. These are typically school buses that transport students to school or back home. The OppNet considered in this scenario therefore exhibits moderate churn, as the number of mobile nodes (in that case, buses) involved in that scenario is not stable over time. Statistics about this scenario are available in Table I. Note that 8.892 radio contacts occur between the nodes during one day, with an average contact duration of 1’24”.

b) Application scenario: The application scenario we consider is defined as follows: each bus broadcasts messages periodically, and each message is meant to be received — ideally — by all other buses. The broadcast period is set to 20 minutes, and the first message is sent by a bus 20 seconds after it starts running. Since all buses do not start running at the same time, and since they do not run for the same amount of time, they do not broadcast the same number of messages. Actually, 2004 messages are broadcast in this application scenario, and since a bus may broadcast a message after other buses have already stopped running, the number of potential receivers for each message depends on when it is broadcast, and by which bus.

[2]https://casa-irisa.univ-ubs.fr/lepton/videos.html

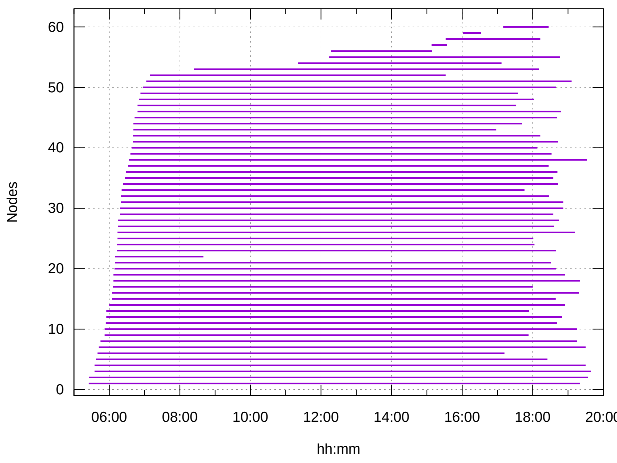


Figure 3: Running intervals in the city bus network scenario (each horizontal line represents the period when a bus runs)

c) Experimental procedure: The scenario described above (that actually combines the mobility and traces scenario with the application scenario) has been run with LEPTON driving DoDWAN nodes. This experiment has first been conducted assuming that the messages broadcast by each bus have no bounded lifetime. In that case an implementation of the causal broadcast algorithm described in Section IV was run by each DoDWAN node. We then assumed that the messages have a bounded lifetime, and the Δ-causal broadcast algorithm described in Section V was run by each DoDWAN node.

d) Metrics considered: During each experiment we recorded data in order to observe the following metrics:

- $bcast_i$: the number of messages broadcast by node i;
- $rcvd_i$: the number of messages received by node i;
- $codlvd_i$: the number of messages co-delivered by node i to its application layer;
- $expd_i$: the number of messages that expired on node i before being co-delivered to the application layer (this is only applicable to messages with a bounded lifetime, and there is necessarily $codlvd_i \leq rcvd_i$);
- scb_m: the size of the causal barrier transported in message m (i.e., number of entries in this data structure);
- $sdlv_i$: the size of the *co-delivered* registry maintained on node i (i.e., number of entries in that registry);
- $tdelay_{m,i}$: the transmission delay of message m to node i (i.e., time elapsed between the time when message m is broadcast by its source node, and the time when this message is received by node i);

- $ltcy_{m,i}$: the co-delivery latency for message m on node i (i.e., the time elapsed between the time when m is received on node i, and the time when it can be co-delivered to that node's application layer).

Note that $tdelay_{m,i}$ is a metric of how long it takes for message m to propagate in the network and reach node i. This is therefore an indication of how long it would take for m to be delivered on i if there was no obligation to deliver messages in causal order. In contrast, $ltcy_{m,i}$ is an indication of how much additional time it takes to ensure causal order in message delivery.

Based on these data we collected, we also calculated several other metrics:

- $codlvdelay_{m,i} = tdelay_{m,i} + ltcy_{m,i}$: the age of message m when its is co-delivered on node i.
- $codlvratio_i = \frac{codlvd_i}{bcast_i + rcvd_i}$: the co-delivery ratio observed on node i (note that the messages co-delivered on node i have either been broadcast by node i, or received by node i);
- $expratio_i = \frac{expd_i}{rcvd_i}$: the expiration ratio observed on node i (note that the messages that expire on node i before being co-delivered to the application layer have necessarily been received by that node, since the messages broadcast by a node are co-delivered immediately to its application layer);

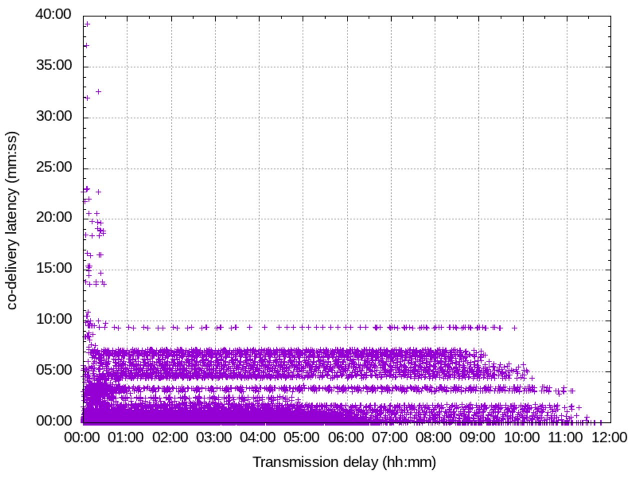
e) Results:

Pure causal broadcast (i.e., no bounded lifetime).

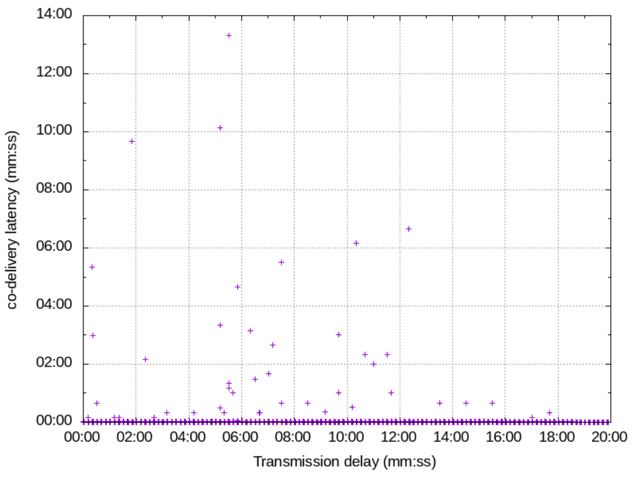
Let us first focus on the results obtained when the messages have no bounded lifetime, that is, the results obtained using the causal broadcast algorithm described in Section IV. Each column in Table II shows details about the number of messages that were broadcast, received, co-delivered, or that expired during a particular experiment. Additionally, each line in Table III provides statistical measures about the transmission delays and co-delivery latencies observed for all messages. The first column in Table II shows that in the experiment conducted with no message lifetime, all the messages got co-delivered on all nodes, hence a 100% co-delivery ratio. This confirms that causal broadcast can indeed be ensured in an OppNet such as the one considered in this scenario.

Figure 4a shows a plot of the co-delivery latency observed for each message against its transmission delay. Notice that in this figure the units in the x and y scales are different. A message may reach a node several hours after it has been sent in the network. However, once this message has been received by a node, the additional time required to co-deliver this message to the local application layer (respecting causal order) is usually within minutes. Indeed, as shown in Table III the average transmission delay observed for messages with no lifetime is 43'50", but the average co-delivery latency is only 13".

Details about the statistical distribution of transmission delays and co-delivery latencies are available in Figure 5, which shows the cumulated distribution functions (CDF) for both metrics. For now, let us again consider only the experiment involving messages with no lifetime. It can be observed that about 50% of these messages are received in less than 20



(a) No lifetime



(b) 20' lifetime

Figure 4: Transmission delay vs co-delivery latency in the city bus scenario

minutes, about 88% in less than an hour, and about 92% in less than two hours. The reason why a few messages still need several hours to be received by some nodes is that these nodes only run episodically in the network. A message broadcast by a node around 08:00 can hardly be received in the morning by a node that only starts running after noon. This node will only have a chance to receive the message after it has started running, that is, several hours after the message has been sent in the network.

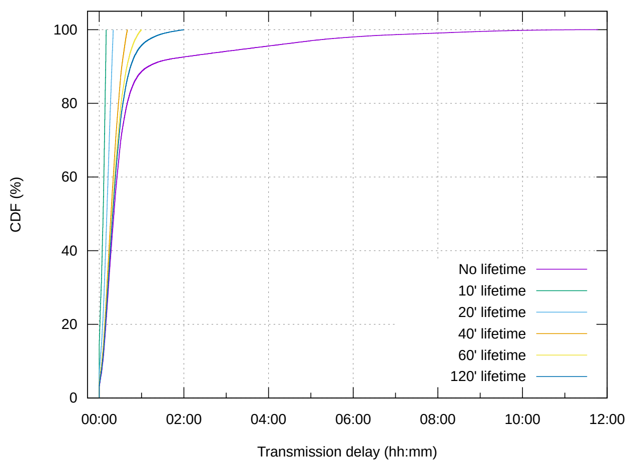
In Figure 5b it can be observed that for most of the messages the co-delivery latency is quite low. Actually, for 90% of the co-delivery events the latency is less than 7.6 seconds, and for 95% of them it is less than 50 seconds. Considering that the actual transmission of a message often takes several minutes in a scenario such as the one considered here, we can conclude that the additional latency incurred in order to ensure causal order remains quite negligible for most messages. Only a few co-deliveries require a latency over a minute. You may notice that in Fig. 5b the range along the x axis has deliberately been cut at 60 seconds, in order to keep the figure readable.

Lifetime	None	120’	60’	40’	20’	10’
nb broadcast events	2004	2004	2004	2004	2004	2004
nb receive events	106580	98610	94329	85301	50365	21571
nb co-delivery events	108580	100614	96331	87304	52364	23572
co-delivery ratio	100%	100%	100%	100%	99.99%	99.99%
nb expiry events	N/A	0	0	0	5	3
expiry ratio	N/A	0%	0%	0%	0.01%	0.01%

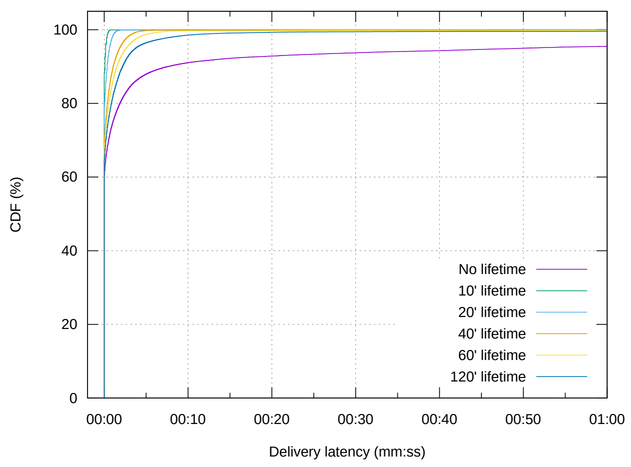
Table II: Statistics about message processing in the city bus scenario

Lifetime	*tdelay* (transmission delay)	*ltcy* (co-delivery latency)	$codlvdelay = tdelay + ltcy$
None	4 ms / 11h47’ / 43’50” / 1h22”	1 ms / 39’13” / 13” / 1’03”	5 ms / 11h47” / 44’04” / 1h22’
120’	4 ms / 1h59’56” / 23’18” / 17’55”	1 ms / 34’53” / 2” / 27”	5 ms / 2h / 23’20” / 17’57”
60’	4 ms / 59’57” / 20’43” / 13’05”	1 ms / 39’20” / 1” / 18”	4 ms / 59’57” / 20’44” / 13’06”
40’	2 ms / 39’54” / 17’50” / 10’04”	1ms / 15’41” / 0.5” / 9”	2 ms / 39’55” / 17’51” / 10’04”
20’	2 ms / 19’53” / 10’47” / 5’27”	1 ms / 13’19” / 0.2” / 6”	2 ms / 19’53” / 10’47” / 5’27”
10’	2 ms / 9’51” / 5’27” / 3’02”	1 ms / 3’39” / 62 ms / 2”	2 ms / 9’51” / 5’27” / 3’02”

Table III: Statistics about transmission and co-delivery delays in the city bus scenario (all quadruplets show min / max / avg / stdev values)



(a) Transmission delays



(b) co-delivery latencies

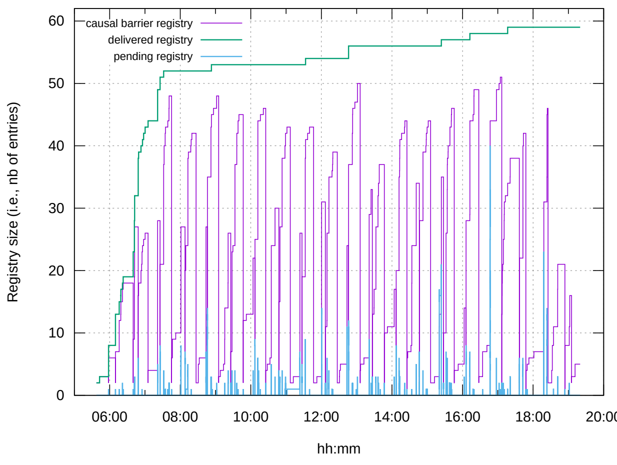
Figure 5: Cumulated distribution functions of transmission delays and co-delivery latencies in the city bus scenario

Actually, the longest co-delivery latency observed during this experiment is 39’13” (as shown in Table III). Such a late co-delivery can typically be observed when a node u receives message m_1 during a contact with another node v, while m_1 depends causally on m_2, which has not been received yet, and will not be received for a while. This may happen for example when a contact between two nodes is disrupted before they have finished exchanging messages. In such a situation node u, which misses m$_2$, must wait for the next opportunity to obtain that message. Meanwhile m_1 must remain in its cache, pending co-delivery.

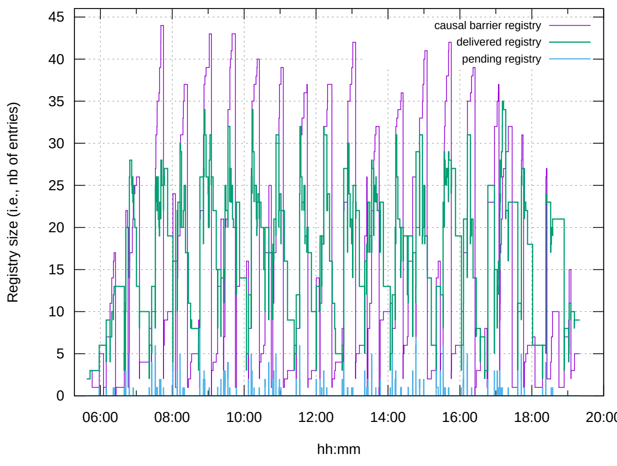
Figure 6a shows how the registries maintained by a node evolve during the experiment. In that case we focus on the registries of a single node we selected. This node’s id is L01-1, which simply means that this is bus #1 in the bus line #01. We verified that the registries on other nodes evolved approximately the same way as what is presented here for node L01-1. It can be observed that the size of the *causal_barrier* registry keeps growing as L01-1 receives messages (from as yet unheard-of nodes), and shrinks to 1 whenever it broadcasts a message. Notice that although the network considered in that scenario includes 60 nodes, the *causal_barrier* registry never contains as many entries. The size of the *pending* registry also keeps growing during the experiment, for node L01-1 has to keep messages in this registry until they can be co-delivered. Yet, although 2004 messages are broadcast during the experiment, L01-1 never has to maintain more than a few dozen messages simultaneously in its pending *registry*. Finally, it can be observed that the size of the *co-delivered* registry keeps growing all day long, as the node receives messages from source nodes it had so far never heard of (so any discovery of a new source yields the creation of a new entry in the *co-delivered* registry). Of course, if the same population of nodes continued running in the evening, the size of the *co-delivered* registry would stabilize once each node has been heard of at least once. On the contrary, the *co-delivered* registry would keep growing if new nodes kept joining the network.

Δ-causal broadcast (i.e., with bounded lifetimes).

Let us now consider the results obtained when running



(a) No lifetime



(b) 20’ lifetime

Figure 6: Evolution of the size of the registries maintained on a node (while running the city bus scenario)

the same scenario with the Δ-causal broadcast algorithm described in Section V. Several experiments have been conducted with that algorithm, with message lifetimes ranging from 10 minutes up to 120 minutes. Considering Table II again, we can observe that the number of broadcast messages is of course the same in all experiments, since we replayed every time the same application scenario. The number of messages received by all nodes however decreases as their lifetime gets shorter. This is perfectly normal, as a message with a short lifetime cannot necessarily reach all possible nodes before it expires. Besides, we have already seen that in this scenario some nodes only run for a short time in the afternoon. These nodes can hardly be reached by messages that are broadcast early in the morning, with lifetimes that do not exceed a few hours.

Figure 4b shows a plot of the co-delivery latency against the transmission delay, based on data recorded with a 20’ lifetime. Notice that in this figure the units in the x and y scales are different. This time the transmission delay of a message and its co-delivery latency are both bounded by the lifetime, and the sum of both metrics (which corresponds to the age of a message when it is co-delivered) is also bounded by the lifetime. In other words, a message can be neither received nor co-delivered after it has expired. This is confirmed by the maximal values shown in Table III.

In Fig. 4b it can be observed that although the transmission delay can take several minutes (up to 20’ in that case), the co-delivery latency is only a few seconds for most message co-deliveries. This is confirmed by the cumulated distribution functions shown in Figure 5. With a 20’ lifetime, for example, the co-delivery latency is less than 1.2 seconds for 99% of the messages. With a 40’ lifetime, it is less than 3.4 seconds for 99% of the messages.

Figure 6b shows how the registries maintained by node L01-1 (the same node as in Fig. 6a) evolved during the experiment involving messages with a 20 minute lifetime. The main difference with Fig. 6a is that the size of the *co-delivered* registry does not grow all day long. Instead it grows and shrinks continuously, as L01-1 receives messages from nodes it has not heard of recently, and forgets these nodes again after a while. This possibility for L01-1 to expunge any reference to not-recently heard-of nodes from its *co-delivered* registry is an important feature of our Δ-causal broadcast algorithm, as it makes it possible for this algorithm to run in OppNets with very high churn ratios. This is illustrated further in Section VI-B.

Overall, the experiments conducted with the city bus scenario confirm that causal and Δ-causal broadcast can be obtained in an OppNet, with a high co-delivery ratio if the dynamics of the network allows messages to disseminate properly. They also show that although transmission delays can be quite long in such a network, the additional latency incurred by the nodes in order to ensure the causally ordered co-delivery of messages is often negligible.

The city bus network considered in this section is just an example of an OppNet. This particular network runs for a limited duration (about 14 hours), with a limited population of nodes, and exhibiting only moderate churn. This is the reason why disseminating messages with no bounded lifetime is possible, and therefore why the causal broadcast algorithm (which assumes no message lifetime) can be used in such a scenario. In other kinds of networks, assigning each message a bounded lifetime is required. In such networks Δ-causal broadcast would be the only option to co-deliver broadcast messages in causal order. This is exemplified in the next section.

B. Δ-causal broadcast in an opportunistic network with high churn

Let us now consider an OppNet exhibiting a high churn ratio. In this particular network the mobile nodes are devices carried by pedestrians, which wander in the streets of the Östermalm district in downtown Stockholm. This is referred to as the Östermalm scenario in the following lines.

a) Mobility and contacts scenario: The dataset describing the mobility of these pedestrians is available in the CRAWDAD database [29]. It has been designed using the

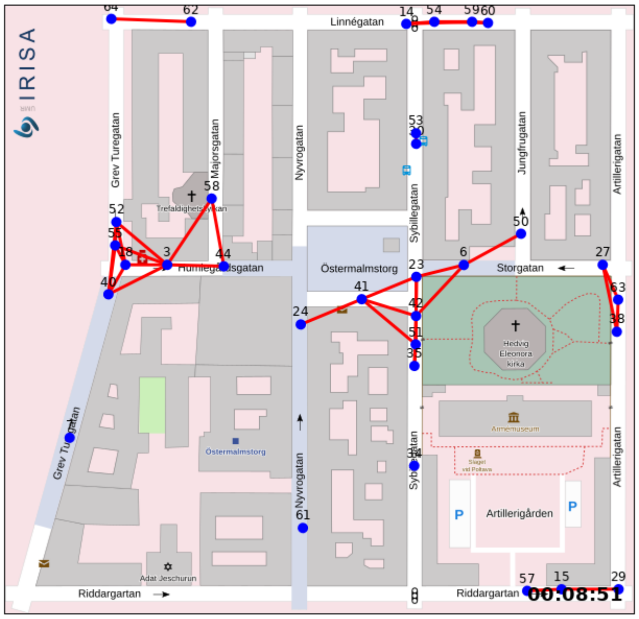


Figure 7: Snapshot of the Östermalm scenario

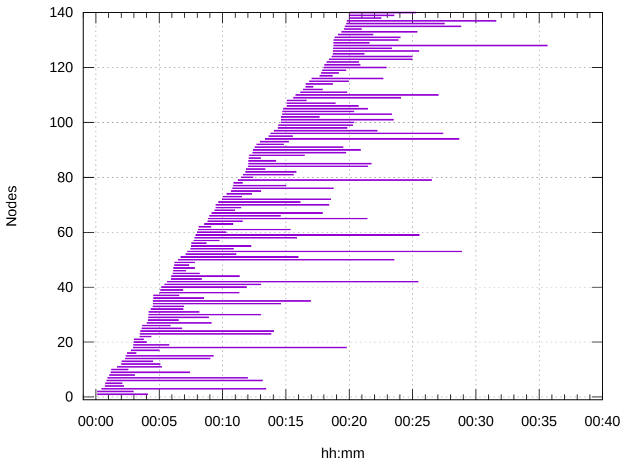


Figure 8: Running intervals of the nodes entering the area during the first 20 minutes of the Östermalm scenario (each horizontal line represents the period during which a node traverses the area)

Metrics	Values (* = min / max / avg / stdev values)
Duration of the scenario	5h
Nb of nodes	2092 (total) / 57 (max. active simultaneously)
Activity duration per node	7.8” / 32’16” / 5’25” / 4’09” (*)
Number of contacts	14807
Durations of contacts	0.6” / 8’12” / 45” / 39” (*)

Table IV: Statistics about the Östermalm scenario

Legion Studio mobility simulator[3], in order to illustrate a typical OppNet with churn [30]. Nodes (i.e., pedestrians) enter the Östermalm area according to a Poisson process, and each node traverses this area at a speed that is chosen from a truncated normal distribution (0.6–2.0) with a mean speed of 1.3 m/s. Several tracesets are actually available in this dataset. For our experiment we used traceset *ostermalm_001_1*, which defines a scenario with an entry rate of 0.01 node/s.

This scenario describes the mobility of nodes over an interval of five hours. During that interval, 2092 nodes traverse the area. Figure 8 shows the activity periods of the 140 first nodes that enter the Östermalm area at the beginning of this scenario (we did not cover the 5 hour scenario here, to keep the figure readable). It is obvious in this figure that the time spent by a node in the Östermalm area is variable. The average traversal time over the five hours of the whole scenario is 5’25”, but this time actually ranges from 7.8” to 32’16”, as shown in Table IV. Besides, the number of nodes that are present simultaneously in the area changes continuously, but this number never exceeds 57 nodes. These figures confirm that the mobile nodes considered in this scenario together compose an OppNet with high churn.

Since the dataset available on the CRAWDAD database only provides information about the mobility of the nodes, we calculated radio contacts between them, assuming a transmission range of 50 meters. With this range, 14.807 radio contacts occur between pairs of nodes in the scenario, with an average contact duration of 45”. The resulting mobility and contacts scenario can be played by LEPTON. A snapshot collected while running this scenario is presented in Figure 7. A video is also available on our website[2].

b) Application scenario: We consider an application scenario, whereby each node broadcasts a message every minute while traversing the area, starting 20 seconds after entering the area. Overall, 10557 messages are broadcast by the whole population of nodes during the 5 hour interval.

c) Experimental procedure and metrics considered: We followed the same experimental procedure, and considered the same metrics, as for the city bus scenario. The only difference is that we did not attempt to run the causal broadcast algorithm (with unbounded message lifetime) with the Östermalm scenario, as this would not make much sense. We therefore assumed that the messages must systematically have a bounded lifetime in such a scenario, and the Δ-causal broadcast algorithm was thus used in each experiment.

d) Results: Several experiments have been conducted, with message lifetimes ranging from 5 minutes up to 20 minutes. Table V shows how many messages were processed by the nodes in each experiment, and Table VI provides statistical measures about transmission delays and co-delivery latencies.

The number of messages received and co-delivered by the nodes get lower as the lifetime of these messages is shorter. Indeed, the number of messages that disseminate in the network at any time is larger with a longer lifetime. Therefore, the shorter the lifetime, the lesser a node is likely to receive many messages while traversing the Östermalm area. The co-delivery ratio observed with this scenario is not as good as with the city bus scenario, but it remains quite good anyway. Even with a 20’ lifetime, more than 95% of the messages received by the nodes eventually get co-delivered to the application layer. Interestingly, the number of messages that expire before

[3]Legion Studio software http://www.legion.com/legion-studio.

Lifetime	20’	15’	10’	5’
nb broadcast events	10557	10557	10557	10557
nb receive events	1,475,328	1,192,205	884,810	539,652
nb co-delivery events	1,414,438	1,169,382	883,120	545,018
co-delivery ratio	95.19%	97.22%	98.63%	99.06%
nb expiry events	1573	1074	724	716
expiry ratio	0.11%	0.09%	0.08%	0.13%

Table V: Statistics about message processing in the Östermalm scenario

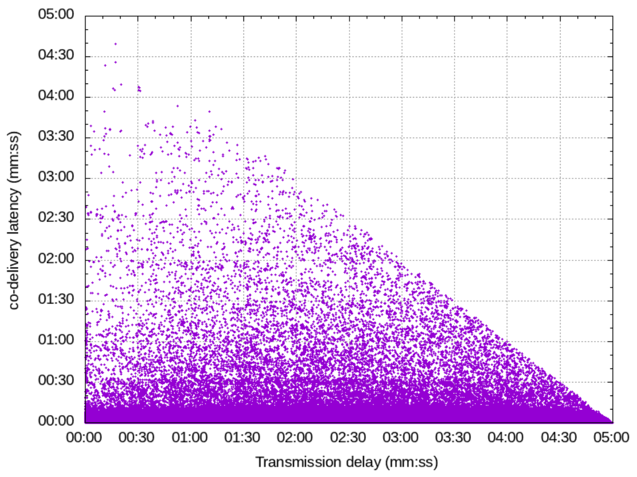
being co-delivered is higher in that scenario. Remember that these are messages that, after being received by a node, are deposited in its pending registry, and eventually get expunged from this registry (because their lifetime is over) instead of being co-delivered. The last line in Table V shows that the expiry ratio remains very low, though. An overwhelming majority of messages get co-delivered before expiring.

You may have noticed that the sum of the co-delivery ratio and of the expiry ratio is not 100%. This is because when a node leaves the network (which happens very frequently in that scenario) its *pending* registry may still contain messages that have not expired yet, and that have not yet been co-delivered either. These messages are neither accounted for in the co-delivery ratio, nor in in the expiry ratio.

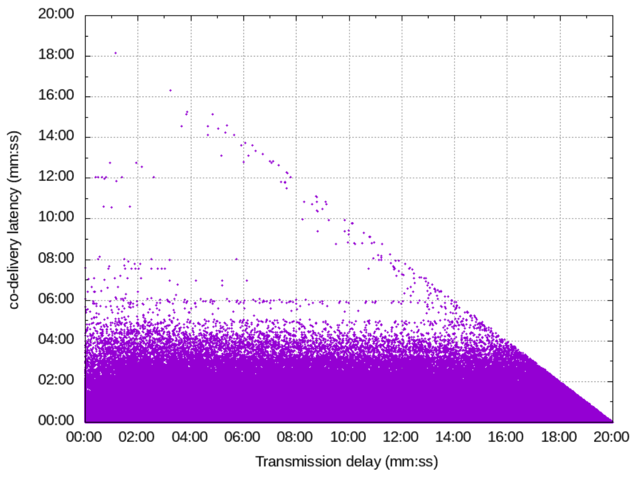
Figure 9 shows plots of the co-delivery latency observed for each message against its transmission delay, in the experiments conducted with 5’ and 20’ lifetimes. Again it can be observed that a message can often be received by a node several minutes after having been sent in the network, but that the additional time it takes to co-deliver this message is usually far shorter. This is confirmed by the average values shown in Table VI, and by the cumulated distribution functions presented in Figure 10. Notice that in Figure 10b the range along the x axis has been cut at 90 seconds, which makes the figure easier to read. The longest co-delivery latency observed during this experiment was actually 18’09” with a 20’ lifetime, as shown in Table VI.

By comparing Fig. 10a and Fig. 10b it can be observed that the co-delivery latency remains globally negligible compared to the transmission delays. Whatever the lifetime considered, the transmission of a message (from its source to a receiver) usually takes several minutes, while the co-delivery latency is usually well under one minute. For example, with a 10’ lifetime the transmission delay is over 5 minutes for 58% of the messages, while the co-delivery latency is under 25” for 95% of the messages, and under 10.7” for 80% of them.

Figure 11 shows how the registries maintained by a particular node (node N777 in that case) evolved during the experiments conducted with 5’ and 20’ lifetimes. Node N777 enters the network around 01h49’, and leaves the network 5 minutes later. During these five minutes, it publishes 5 messages in both experiments. In the 5’ lifetime experiment it receives 269 messages, and co-delivers 274 messages to the application layer. In the 20’ lifetime experiment it receives 760 messages, and co-delivers 764 messages. Thanks to the Δ-causal broadcast algorithm, the size of the *co-delivered* registry on node N777 does not grow as long as it stays in the network. In fact this registry grows rapidly when N777 enters the network, and then its size remains quite stable afterwards.



(a) 5’ lifetime



(b) 20’ lifetime

Figure 9: Transmission delay vs co-delivery latency in the Östermalm scenario

The *pending* registry also grows rapidly at first, but it rapidly shrinks down. The rapid growth of both registries when N777 enters the network can easily be explained. Soon after entering the network, node N777 gets in radio contacts with several other nodes. Most of these nodes have already “lived” in the network for a while, so they are carrying many messages in their cache. N777 can therefore rapidly obtain a fair number of messages, most of which have been broadcast even before it entered the network. These messages are received in any order by the opportunistic networking layer, though, so they cannot necessarily be co-delivered immediately to the application layer. They are therefore first put in the *pending* registry (whose size grows rapidly), until the Δ-causal broadcast algorithm manages to co-deliver them to the application layer. In Fig. 6b we can see that when N777 enters the network, almost 400 messages are received and deposited in its *pending* registry in only a few seconds. Over the same laps of time the size of the *co-delivered* registry reaches up to 150 entries, which means that at least some of the newly received messages are co-delivered almost immediately to the application layer,

Lifetime	*tdelay* (transmission delay)	*ltcy* (co-delivery latency)	$codlvdelay = tdelay + ltcy$
20’	2 ms / 19’59” / 9’03” / 5’59”	1 ms / 18’09” / 19 ” / 30”	2 ms / 20’ / 9’22” / 6’02”
15’	3 ms / 14’59” / 6’40” / 4’27”	1 ms / 13’08” / 13” / 23”	3 ms / 15’ / 6’53” / 4’30”
10’	4 ms / 10’ / 4’23” / 2’55”	1 ms / 8’48” / 7” / 16”	4 ms / 10’ / 4’31” / 2’56”
5’	3 ms / 5’ / 2’16” / 1’28”	1 ms / 4’39” / 2” / 10”	3 ms / 5’ / 2’19” / 1’28”

Table VI: Statistics about transmission and co-delivery delays in the Östermalm scenario (all quadruplets show min / max / avg / stdev values)

and these messages come from 150 different sources. Most of the other messages are rapidly expunged from the pending registry, though, as they get co-delivered to the application layer. After this initial surge of receive and co-delivery events, the size of the *co-delivered* registry does not keep growing, even though new nodes keep entering the network. This is because at the same time other nodes leave the network, and are thus “forgotten” by the Δ-causal broadcast algorithm which does not hear of them anymore. Finally, as expected the size of the *causal_barrier* registry varies up and down as messages are co-delivered and broadcast by node N777 as long as it remains in the network.

Similar observations can be made when examining Fig. 11a. The registries do no grow as much as in Fig. 6b, though, because with a shorter lifetime the number of messages that disseminate in the network is of course far smaller.

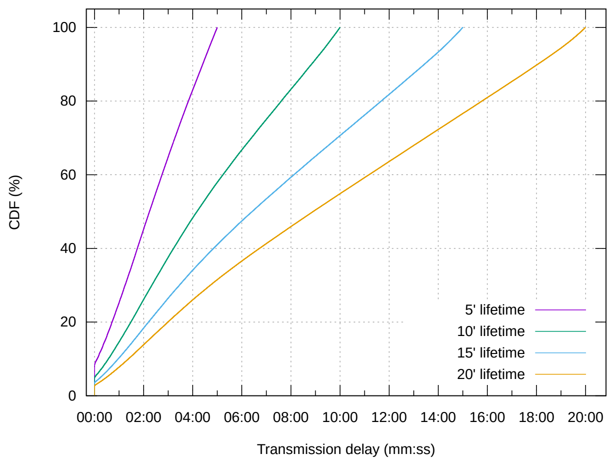
Overall, these results confirm that our Δ-causal broadcast algorithm can indeed run in a network exhibiting a very high churn ratio, while avoiding that the data space used to maintain dependency information on each node grows monotonically.

VII. Conclusion

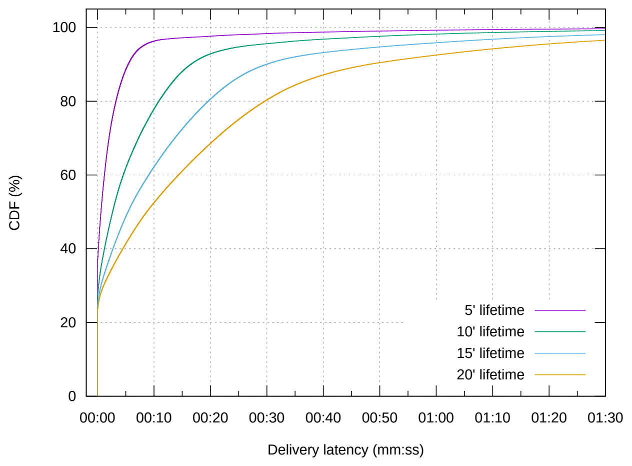
In this paper we have proposed two algorithms that make it possible to ensure causal broadcast and Δ-causal broadcast in opportunistic networks (OppNets). Both algorithms are derived from [11] and [12] respectively, but they have been adjusted so as to take into account the peculiarities of OppNets. Most notably, they can run in a network in which the population of nodes is unknown, and can change continuously. Experimental results produced by running these algorithms in an emulated environment, using realistic mobility and contact scenarios, confirm that causal broadcast can indeed be ensured in an OppNet, with a ratio of delivered messages that remains close to 100%. The delivery latency of a message (i.e., the time required to deliver this message once it has been received) is usually negligible compared to its transmission delay. Finally, when using Δ-causal broadcast algorithm in an OppNet with a high churn ratio, the data space occupied on each node to maintain dependency information does not monotonically increase as nodes join and leave the network over time.

Funding

This work was supported by the French ANR (Agence Nationale de la Recherche) under grant number ANR-16-CE25-0005-02.



(a) Transmission delays



(b) co-delivery latencies

Figure 10: Cumulated distribution functions of transmission delays and co-delivery latencies in the city bus scenario

References

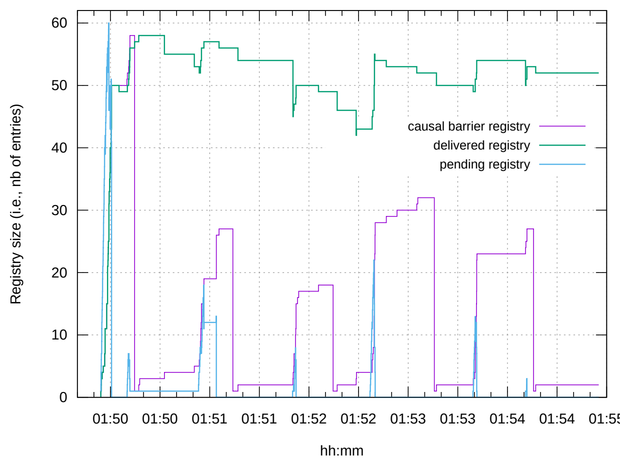
[1] Ken Birman. Replication and Fault-Tolerance in the ISIS System. In *ACM SIGOPS Operating Systems Review*, volume 19, pages 79–86, 12 1985.

[2] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

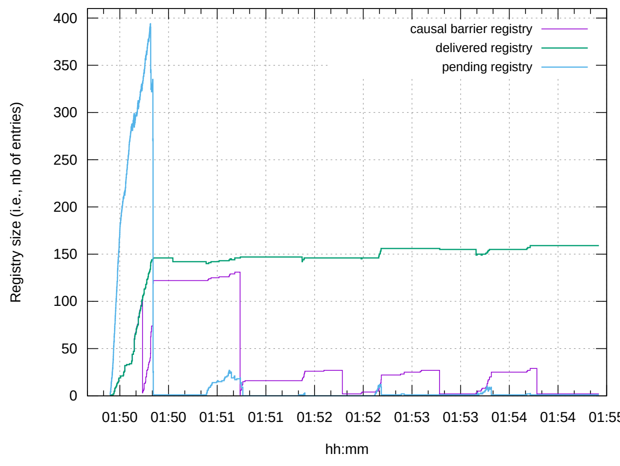
[3] Luciana Pelusi, Andrea Passarella, and Marco Conti. Opportunistic Networking: Data Forwarding in Disconnected Mobile Ad Hoc Networks. *IEEE Communications Magazine*, 44(11):134–141, November 2006.

[4] Kevin Fall. Messaging in Difficult Environments. Technical Report IRB-TR-04-019, Intel Research Berkeley, 2004.

[5] Chiara Boldrini, Marco Conti, and Andrea Passarella. Autonomic Behaviour of Opportunistic Network Routing. *Inderscience Interna-*



(a) 5' lifetime



(b) 20' lifetime

Figure 11: Evolution of the size of the registries maintained on node N777 (while running the Östermalm scenario)

tional Journal of Autonomous and Adaptive Communications Systems, 1(1):122–147, 2008.

[6] Maurice J. Khabbaz, Chadi M. Assi, and Wissam F. Fawaz. Disruption-Tolerant Networking: A Comprehensive Survey on Recent Developments and Persisting Challenges. *IEEE Communications Surveys and Tutorials*, 14(2):607–640, 2012.

[7] Hoang Anh Nguyen and Silvia Giordano. Routing in Opportunistic Networks. *International Journal of Ambient Computing and Intelligence (IJACI)*, 1(3):19–38, 2009.

[8] Alicia Triviño Cabrera and S. Cañadas Hurtado. Survey on Opportunistic Routing in Multihop Wireless Networks. *International Journal of Communication Networks and Information Security (IJCNIS)*, 3(2):170–177, August 2011.

[9] Zhensheng Zhang. Routing in Intermittently Connected Mobile Ad Hoc Networks and Delay Tolerant Networks: Overview and Challenges. *IEEE Communications Surveys and Tutorials*, 8(1):24–37, January 2006.

[10] Zhensheng Zhang and Qian Zhang. Delay/disruption tolerant mobile ad hoc networks: latest developments. *Wireless Communications and Mobile Computing*, 7(10):1219–1232, May 2009.

[11] Ravi Prakash, Michel Raynal, and Mukesh Singhal. An Efficient Causal Ordering Algorithm Suited to Mobile Computing Environments. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 744–751, Honk Kong, May 1996. IEEE.

[12] Roberto Baldoni, Ravi Prakash, Michel Raynal, and Mukesh Singhal. Efficient Delta-Causal Broadcasting. *International Journal of Computer Systems Science and Engineering*, 13:263–271, 1998.

[13] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, volume 10 of *Australian Computer Science Communications*, pages 56–66, 1988.

[14] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, October 1989. North-Holland.

[15] Ken Birman, André Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 08 1991.

[16] Mukesh Singhal and Ajay Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47–52, 08 1992.

[17] Chafika Benzaid and Nadjib Badache. An Optimal Causal Broadcast Protocol in Mobile Dynamic Groups. In *International Symposium on Parallel and Distributed Processing with Applications, ISPA 2008*, pages 477–484, 12 2008.

[18] Achour Mostefaoui and Stéphane Weiss. Probabilistic Causal Message Ordering. In *14th International Conference on Parallel Computing Technologies (PaCT 2017)*, volume 10421 of *LNCS*, pages 315–326, Nizhni Novgorod, Russia, September 2017. Springer.

[19] Brice Nédélec, Pascal Molli, and Achour Mostefaoui. Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems. In *37th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60, Salvador de Bahia, Brazil, October 2018. IEEE.

[20] Brice Nédélec, Pascal Molli, and Achour Mostefaoui. Causal Broadcast: How to Forget? In *22nd International Conference on Principles of Distributed Systems (OPODIS)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:16, Hong-Kong, December 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[21] Achour Mostefaoui, Matthieu Perrin, Michel Raynal, and Cao Jiannong. Crash-Tolerant Causal Broadcast in O(n) Messages. *Information Processing Letters*, 151:105837, 2019.

[22] Mihail Costea, Radu-Ioan Ciobanu, Radu-Corneliu Marin, Ciprian Dobre, Constandinos X. Mavromoustakis, and George Mastorakis. *Causal and Total Order in Opportunistic Networks*, chapter Emerging Innovations in Wireless Networks and Broadband Technologies, pages 221–262. IGI Global, 2016.

[23] Mihail Costea, Radu-Ioan Ciobanu, Radu-Corneliu Marin, Ciprian Dobre, Constandinos X. Mavromoustakis, George Mastorakis, and Fatos Xhafa. Total Order in Opportunistic Networks. *Concurrency and Computation: Practice and Experience*, 29(10), 2017.

[24] Amin Vahdat and David Becker. Epidemic Routing for Partially Connected Ad Hoc Networks. Technical Report CS-200006, Duke University, Durham, USA, April 2000.

[25] Anwitaman Datta, Silvia Quarteroni, and Karl Aberer. Autonomous Gossiping: a Self-Organizing Epidemic Algorithm for Selective Information Dissemination in Mobile Ad-Hoc Networks. In *1st International Conference on Semantics of a Networked World (ICSNW'04)*, number 3226 in LNCS, pages 126–143, Paris, France, June 2004.

[26] Julien Haillot, Frédéric Guidec, Serge Corlay, and Jacques Turbert. Disruption-Tolerant Content-Driven Information Dissemination in Partially Connected Military Tactical Radio Networks. In *28th IEEE Military Communication Conference (MILCOM'2009)*, pages 2326–2332, Boston, USA, October 2009. IEEE CS.

[27] Julien Haillot and Frédéric Guidec. A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks. *Journal of Mobile Information Systems*, 6(2):123–154, 2010.

[28] Adrián Sánchez-Carmona, Frédéric Guidec, Pascale Launay, Yves Mahéo, and Sergi Robles. Filling in the missing link between simulation and application in opportunistic networking. *Journal of Systems and Software*, 142:57–72, August 2018.

[29] Sylvia Todorova Kouyoumdjieva, Ólafur Ragnar Helgason, and Gunnar Karlsson. CRAWDAD dataset kth/walkers (v. 2014-05-05). CRAWDAD wireless network data archive (https://crawdad.org), May 2014.

[30] Ljubica Pajevic and Gunnar Karlsson. Modeling Opportunistic Communication with Churn. *Computer Communications*, 96:123–135, 2016.