Scalability Approaches for Causal Multicast: A Survey

Rubén de Juan-Marín, Hendrik Decker, J. Enrique Armendáriz-Íñigo,
José M. Bernabéu-Aubán, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València, 46022 Valencia (Spain)

{rjuan, hendrik, armendariz, josep, fmunyoz}@iti.upv.es

Technical Report IUMTI-SIDI-2015/001

Scalability Approaches for Causal Multicast: A Survey

Rubén de Juan-Marín, Hendrik Decker, J. Enrique Armendáriz-Íñigo,
José M. Bernabéu-Aubán, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València, 46022 Valencia (Spain)

Technical Report IUMTI-SIDI-2015/001

e-mail: {rjuan, hendrik, armendariz, josep, fmunyoz}@iti.upv.es

May 19, 2015

Abstract

Many distributed services need to be scalable: internet search, electronic commerce, e-government... In order to achieve scalability and fault tolerance, such applications rely on replicated components. Because of the dynamics of growth and volatility of customer markets, applications need to be hosted by elastic systems. In particular, the scalability of the reliable multicast mechanisms used for supporting the consistency of replicas is of crucial importance. Reliable multicast may propagate updates in a predefined order (e.g., FIFO, total or causal). Since total order needs more communication rounds than causal order, the latter appears to be the preferable candidate for achieving multicast scalability, although the consistency guarantees based on causal order are weaker than those of total order. This paper provides a historical survey of different scalability approaches for reliable causal multicast protocols.

1 Introduction

A growing number of dependable applications ensure their high availability and reliability by replicating their components. Typical examples are web search, e-business, grid computing and cloud computing. The replicas of such applications have to maintain some degree of consistency [42]; i.e., bounding the divergences allowed among the states of such replicas. So, when an update operation is initiated in one replica, the same operation (or its results) should be propagated to all other replicas and applied there. The degree of replica consistency depends on the particular application, on how the replicas are organised, and on the order of update propagation.

These dependable distributed applications should deal with a large number of users. Therefore, they need to be able to scale horizontally. Horizontal scalability implies that, whenever the number of nodes in a system increases, it is able to serve more users with the same QoS characteristics (e.g., average response time). In many applications, most operations are read-only and each client may be served by a different server process. Thus, an increase in the number of servers directly increases the service capacity. On the other hand, client operations that modify the server state should be propagated to all replicas. In many cases, the communication needed by updating operations is the bottleneck that prevents systems from scaling out.

Thus, a scalable multicast protocol for propagating updates to all servers is needed. FIFO total-order multicast provides the basis for supporting sequential consistency [23]. However, total-order multicast demands consensus at some point [16]. Thus, total order entails additional communication rounds and this compromises scalability. On the other hand, when updates are propagated among replicas using a reliable causal multicast, causal consistency is maintained. These multicast protocols only need a single communication step in the regular case. So, causal multicast provides an advantageous basis for scalable

communication. However, this single communication round uses messages that should carry some kind of causal history and the amount of such information depends on the system size. As a result, in order to multicast messages to as many servers as possible, some mechanisms should be devised to reduce the overall size of that causal history.

Scalable services are regularly deployed in multiple data centres and network partitions may arise. According to the CAP theorem [24], only two of the following three characteristics can be maintained simultaneously in a distributed system: Consistency, Availability and Partition-tolerance. So, in a distributed system where partitions may happen and where service availability is also a must, consistency needs to be sacrificed. Because of this, those systems usually adopt *eventual consistency* [20] and that condition can be easily met using causal consistency as its base [47].

Some examples of these scalable services, demanding causal multicasts, are:

1. *Services with commutable updates* (e.g., e-commerce services). Most client actions in e-commerce services consist in viewing the available products and updating the client's cart. Cart updates can be implemented as add or subtract operations on the set of elements in the cart and on the current warehouse stock levels. Concurrent operations started by different customers may be executed following different orders in different replicas. If those operations are commutable [48, 25] then there will be no danger in using causal communication: causally related operations follow the same order in all replicas and concurrent operations will generate the same server state once all they are applied.

   Commutable updates may be combined with lazy propagation [33]. Lazy update propagation and causal communication do not introduce any noticeable delays for managing updates. As a result, servers may reply earlier to client requests.

   All these mechanisms (causal communication, commutable operations, lazy propagation...) provide the basis for implementing *eventual consistency* [57] that is the common approach now in scalable distributed services.

2. *Mobile services*. Multiple applications that are deployed in mobile networks (as event notification services [37], virtual environments [65], information diffusion in VANETs [36, 62],...) do not require strong consistency. Thus, reliable causal multicast protocols for mobile systems have been proposed in many papers [7, 51, 50, 64]. Those protocols also involve some of the mechanisms described in this paper, since most mobile computers are battery-backed and they need to minimise message size in order to extend battery life.

   A distributed system with mobile nodes is prone to network partitions. If the services deployed in such system should be continuously available, different subsystems may have a divergent state when the partition is healed. The usage of a causal history (in any of its forms: precedent messages, vector clocks...) in the reliable causal multicasts combined with version vectors in the replicated components allows an easy identification of the state divergences [46] that should be fixed, simplifying the reconciliation algorithms demanded in those cases.

3. *Cloud management*. In recent cloud computing systems, some kind of tool is needed to administer their virtual infrastructure [59]; i.e., to manage the large amount of physical nodes that compose a data centre and the virtual machines that could be deployed in each of those nodes. Reliable causal multicasts provide a low-level mechanism for dealing with the communication needs of such management tools.

To the best of our knowledge, there are several textbooks (e.g., [14, 32]) that analyse and compare some reliable causal multicast algorithms, but there is yet no published survey or tutorial about the mechanisms used in the numerous reliable causal multicast protocols existing today. This paper attempts to fill this void, by identifying approaches that are better suited for attaining scalability. To this end, Section 2 specifies what is understood as reliable causal multicast. Section 3 summarises the main characteristics of the proposed approaches and gives a historical review of these mechanisms. Finally, Section 4 sums up comparing the surveyed causal multicast mechanisms and Section 5 concludes the paper.

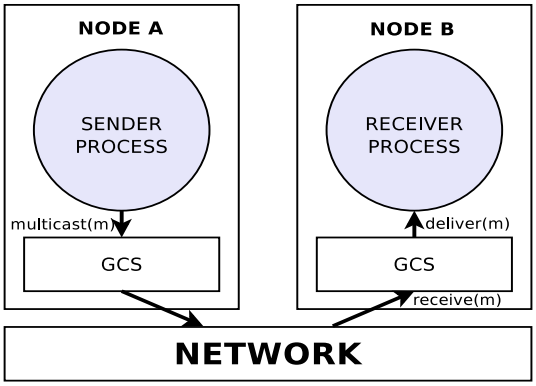

Figure 1: Communication events.

2 Reliable Causal Multicast

2.1 System Model

We assume an asynchronous distributed system that uses message passing as its interprocess communication mechanism. There exists a distributed application that needs causal propagation of messages to all the nodes where its processes are executed. The system being considered is that composed by all those nodes where the application components have been deployed.

Many surveyed papers state that they assume "*reliable communication*". However, often this only means that channels do not duplicate messages nor create spurious ones and that message delivery is guaranteed as long as sender and receiver do not fail; i.e., communication channels are *quasi-reliable* [2]. As a result, this implies that reliable message delivery is a responsibility of the multicast protocol.

A process is considered *correct* when it does not fail (in the scope of the communication steps being considered), and it is *faulty* otherwise.

Some of the surveyed multicast protocols manage uniform message delivery. *Uniform delivery* [26] means that if any correct or faulty process has delivered a message m, then all correct processes will be able to deliver m.

2.2 Communication Interface

The protocols presented in subsequent sections may be implemented inside a *group communication system* (GCS) [17]. Given a GCS, three different events can be distinguished with regard to message transmission, as shown in Fig. 1:

1. *multicast(m)*. The message m is multicast by its sender process (*sender(m)*).

2. *receive(m)*. The message m is received by a GCS module in each node where some process belonging to the group exists. It is temporarily buffered in the GCS module until causal order can be ensured.

3. *deliver(m)*. The message m is received by some target process from its underlying GCS module. The delivery of m complies with causal order. Note that the state reached after a previous *receive*(m) event does not necessarily guarantee that messages are already causally ordered. Further note that this third event may sometimes also be unambiguously called "message reception", if it is understood that the receiver is the target process, not its underlying GCS.

2.3 Specification

A process join or failure implies a *view change* event (i.e., a *view* encompasses the set of processes that constitute the group/system, and such a set is changed each time a process either joins or leaves the group)

in a GCS.

Since not all existing protocols are view-oriented, a general specification of reliable causal multicast is needed. To this end, that presented in [26] is taken here.

With this, reliable multicast is specified using these three properties:

P1 *(Validity)*. If a correct process p multicasts a message m, it eventually delivers m.

P2 *(Agreement)*. If a correct process p delivers a message m then all correct processes eventually deliver m.

P3 *(Integrity)*. For any message m, every correct process delivers m at most once, and only if m was previously multicast by $sender(m)$.

So, when no failures arise, a reliable multicast needs to be delivered by all processes that are alive. On the other hand, if its sender fails while a message m is being multicast, two outcomes may arise (considering P2): either all processes deliver m or none of them do so. Thus, if some of the target processes have delivered m, they should be able to maintain and forward m to the remaining target processes in case of failure of $sender(m)$, using a specialised algorithm to this end (e.g., the *flush* algorithm described in [13]).

We are interested in causal delivery, inspired by the *happens before* relation defined by Lamport [34]. Such relation can be summarised as follows.

Let us assume a distributed system consisting of a set S of processes whose execution could be represented by a sequence of events in each process. The *happens before* relation (quoting [34]) denoted by "$\rightarrow$" has this definition:

> The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions:
>
> 1. If a, b are events in the same process, and a comes before b, then $a \rightarrow b$.
> 2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
> 3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.
>
> Two distinct events a and b are said to be *concurrent* ($a \parallel b$) if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Note that $\not\rightarrow$ is the negation of $\rightarrow$. Clearly, this definition entails that $\rightarrow$ is anti-reflexive, anti-symmetric, anti-cyclic and transitive.

So, a reliable causal multicast protocol should comply with all properties as defined above for reliable multicasts, plus the next one:

P4 *(Causal order)*. If the multicast of a message m causally precedes (i.e., "*happens before*") the multicast of a message m', then if some correct process p delivers both messages then no correct process delivers m' unless it has previously delivered m.

2.4 Example

In order to illustrate what causal multicast is, imagine a simple execution that uses causal multicasts. Let us assume a system $S_1 = \{p_1, p_2, p_3, p_4\}$. These processes multicast four different messages m_1, m_2, m_3 and m_4 as shown in Fig. 2. We compactly denote Prop. P4 by $m \rightarrow m'$.

Note that $m_1 \rightarrow m_2$, since m_2 was sent by p_3 once it had delivered m_1. This introduces a problem in p_4, since it receives such messages in the opposite order, and this forces p_4 to buffer m_2 in its receiving queue until m_1 is delivered. This has been shown drawing m_2's reception in p_4 as a circle, and needing an additional arrow to represent its delivery (at the appropriate time). Also, $m_2 \rightarrow m_3$, since m_3 was sent by p_2 once it had delivered m_2. Again, this demands that p_1 stops m_3 delivery until m_2 is received and delivered. Finally, note that $m_3 \not\rightarrow m_4$, since p_1's GCS had received m_3 before sending m_4, but it did not deliver m_3 to p_1 because m_2 was still in transit. Since both m_1 and m_4 have been multicast by p_1 in that order, they also follow that $m_1 \rightarrow m_4$. Additionally, the following relations are also true: $m_2 \parallel m_4$ (since m_2 is sent by p_3

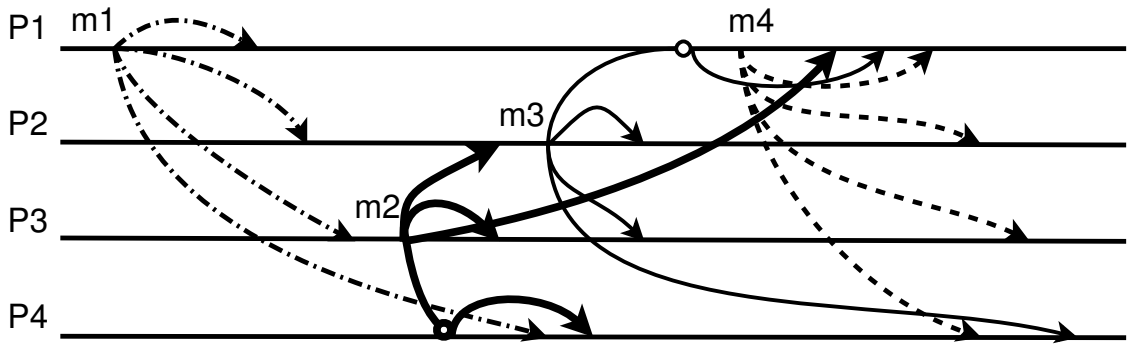

Figure 2: Execution Ex_1 with causal multicasts.

and m_4 by p_1 and m_2 is delivered in p_1 once m_4 was already multicast; i.e., there is no communication —or multicast-deliver— path relating both sending events), $m_3 \parallel m_4$ (because of the same reason), and $m_1 \rightarrow m_3$ (since p_2 delivered m_1 before multicasting m_3, and also due to transitivity: $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$).

3 Reliable Causal Multicast Scaling Mechanisms

A brief technical description of each generation of multicast scalability solutions is going to be given now, highlighting the advantages and drawbacks of each of them.

Scalability can be improved if the messages maintain a moderate size and the number of messages needed by the multicast protocol is minimised. The following sections describe how these two problems have been managed in different protocol classes. Section 3.1 is devoted to describe the first generation of causal multicast protocols, where scalability was not yet considered.

3.1 First Generation

Logical clocks [34] were devised for registering causal dependencies among the events of distributed executions. Thus, they record the causal dependencies between sending and reception events. The first generation of reliable causal multicast protocols followed a similar idea. However, those protocols preferred to (re-)transmit the causally precedent messages instead of adding any kind of logical clock value to the new messages being multicast. Thus, the senders simply added some previously received messages as a causal history in each new multicast; i.e., instead of tagging that new message with its intended clock value, the messages that have caused any increase in that value are also sent (as an ordered sequence) in that multicast. If any of the receivers has not yet delivered any of the messages in that sequence, it proceeds to their delivery in the appropriate order. This also guarantees causally ordered delivery.

CBCAST [12] was one of the first proposed protocols of this kind. However, Hadzilacos and Toueg [26] describe a similar one, more compact, and without implementation details. We present it into Fig. 3. It uses a modular design and requires an underlying uniform reliable FIFO multicast as its basic building block.

Note that in this algorithm any sender process p includes as causal history all messages that have been received by p since it sent its previous multicast. Since the underlying multicast service is uniform, reliable and FIFO, this guarantees that no message could be ever lost and that all causally precedent messages will be received by all target receivers. When a receiving process gets that message sequence, it only needs to check whether it had previously delivered any of its messages. If so, those messages are ignored; otherwise, they are delivered following the appropriate order.

In order to illustrate how this protocol works, Fig. 4 shows the same example depicted in Fig. 2 as it is managed by this protocol. Let us discuss its main steps in the sequel:

1. When p_1 multicasts m_1 no previous message was received yet. So, the sequence of this multicast only consists of $\langle m_1 \rangle$.

```
Initialisation:
 previousDelivered := ⊥

CausalMulticast(m):
 FIFOMulticast(previousDelivered · m)
 previousDelivered := ⊥

CausalDeliver(m):
 upon FIFODeliver(⟨ m1, m2, ..., mk ⟩) for some k do
  for i := 1 to k do
   if p has not yet executed CausalDeliver(mi)
   then
    CausalDeliver(mi)
    previousDelivered := previousDelivered · mi
   fi
  done
 done
```

Figure 3: Non-blocking causal multicast algorithm.

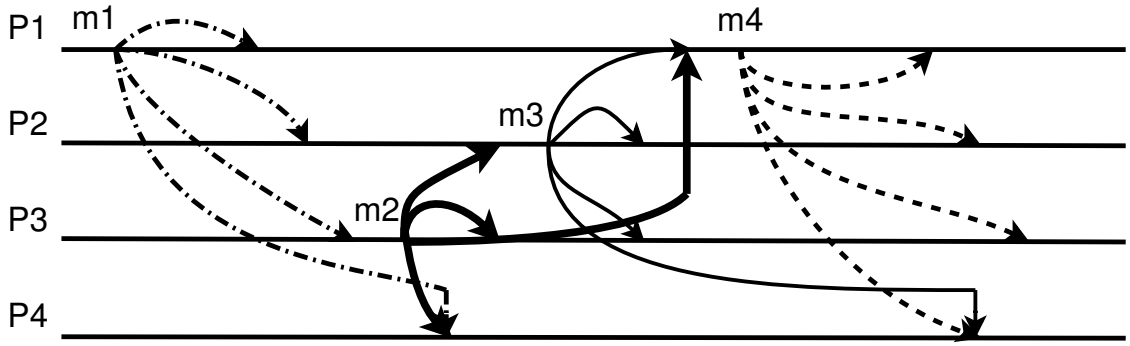

Figure 4: Execution Ex_2 of the non-blocking protocol.

2. When p_3 multicasts m_2 it should send the sequence $\langle m_1, m_2 \rangle$, since m_1 was already delivered in p_3.

   This explains why p_4 delivers both m_1 and m_2 at the same time (and in the correct order), since the original m_1 multicast by p_1 had not yet been received there when the sequence $\langle m_1, m_2 \rangle$ multicast by p_3 was received.

3. When p_2 multicasts m_3 it had already received m_1 and m_2, so the sequence being multicast is $\langle m_1, m_2, m_3 \rangle$ at that time.

   This explains that m_2 and m_3 are delivered at the same time in p_1.

4. Finally, when p_1 multicasts m_4 it multicasts the sequence $\langle m_2, m_3, m_4 \rangle$. This ensures that p_4 is able to deliver m_3 before m_4 when it receives such sequence.

Note that the resulting execution Ex_2 is quite different to the execution Ex_1 shown in Fig. 2. Ex_1 verifies that $m_2 \parallel m_4$ and $m_3 \parallel m_4$, but Ex_2 does not maintain the same relations. Because of the inclusion of potentially causally related messages, Ex_2 verifies that $m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4$; i.e., it has managed as "causally related" some messages that were actually concurrent.

Advantages. Since all messages included in the transmitted packets are appropriately ordered, causal order is trivially ensured in these protocols. Additionally, such causal delivery never demands that messages were blocked at their delivery step, since when a message is received and is ready to be delivered, all

its causally precedent messages have also been received and delivered. Note that none of the protocols explained in the following sections are able to maintain this non-blocking behaviour.

Drawbacks. Although the inclusion of precedent messages is able to ensure the protocol correctness and its non-blocking behaviour, there is a price to pay: the size of the transmitted packets could be very large when both the message sending rate and the group size are increased. This seriously compromises scalability and explains why this approach was abandoned when vector clocks were introduced in order to track message causality.

On the other hand, in systems with a few nodes and in intervals where only a single process is multicasting messages, no precedent message needs to be included in a multicast. As a result, the message propagation pattern being followed by every application should be adequately analysed or monitored before discarding this initial protocol. Despite this, most *scalable* applications require that every replica accepts user requests. As a result, a load-balanced application deployment would require that every replica initiates a similar amount of multicasts. In this scenario, this protocol regularly generates large sequences of messages in each sending operation, since multiple preceding messages should be included in every multicast.

Historical review. The first generation of causal multicast mechanisms was initiated by the CBCAST protocol designed in the Isis project [12]. Its advantages and drawbacks have already been described.

A logical consecutive step in this evolution was the design of a mechanism that still preserves some notion of the causal history in each multicast message, but without requiring the inclusion of entire past messages in such sendings. *Psync*'s *conversations* [48] provided a first solution in this line. It consisted in including only the identifiers of precedent messages, but not their contents. A similar solution was used in [33] for labelling client-ordered operations in their *lazy replication* proposal.

In both cases, the size of sent messages is reduced. However, this came at a cost: when causally-precedent messages are included in a multicast, its delivery does not demand any additional delay (i.e., it is *non-blocking*); on the other hand, if those precedent messages are not included and at least one of them has not been received yet, delivery is temporarily blocked. As a result, these identifier-based multicasts introduced *blocking delivery*. Those solutions were generalised with the introduction of *vector clocks*.

Alternatively, for point-to-point communication, Mattern and Fünfrocken [40] prove that historical information is not needed at all when the communication channels being used are synchronous. With this, precedent messages need not be included in the messages being sent, minimising the size of those messages. However, synchronous communication means that the sender cannot put any subsequent message in a communication channel until the receiver has acknowledged the reception of the current message. Using both sending and receiving buffers at both communication ends, synchronous communication is able to emulate a non-blocking behaviour for the senders. In spite of this, the overall communication performance is much worse than using vector clocks, to be described in the next section.

3.2 Vector Clocks

Vector clocks [22, 39] consist of as many entries as processes compose the system. They assume that those processes use natural numbers as their identifiers. So, in a process p each vector slot k maintains as its value the number of messages sent by process k known by process p.

Vector clocks can be considered as an adaptation of the version vectors proposed in [46] as a means to detect inconsistencies in distributed file systems. Indeed, version vectors maintain the number of updates applied by each node to a given file or set of files, while vector clocks hold the number of messages sent by each node. So, in both cases the vectors maintain the set of relevant events (either updates or message sendings) applied by each system node.

In the regular causal multicast protocols based on vector clocks, each process maintains a vector clock VC with as many components as processes exist in the system. Such clock entries are updated according to the rules from [13]:

- Each time a message m is multicast by p_i, it is tagged with the local vector clock of its sender (i.e., $VC_m := VC_{p_i}$), once such vector clock has increased its local entry (i.e., $VC_{p_i}[i] := VC_{p_i}[i] + 1$).
- When m is received by p_j, it is checked whether entry i (the sender's one) in the message vector clock is one unit greater than the value of this same entry in p_j's clock, and all other entries are greater or

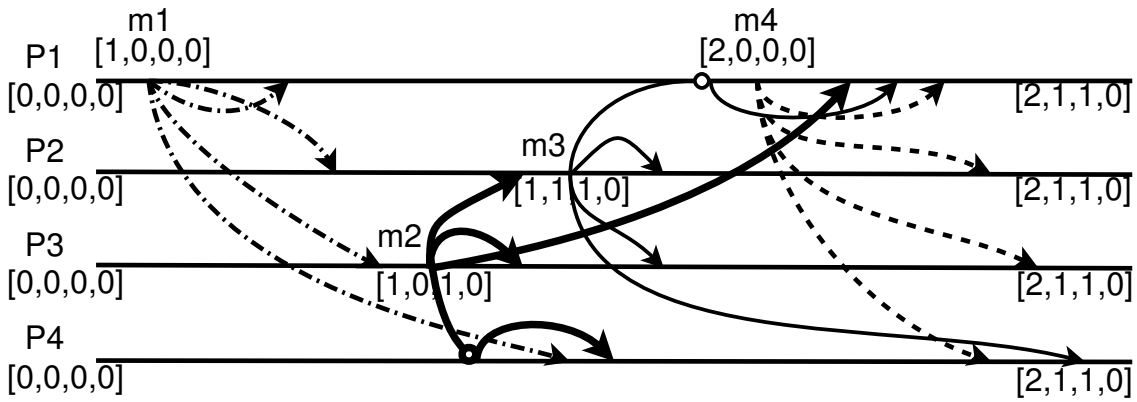

Figure 5: Execution Ex_1 managed with vector clocks.

equal in p_j's clock than in the message clock; i.e., the following should be respected:

$$VC_m[i] = VC_{p_j}[i] + 1$$

and, assuming that n is the system size, $\forall k \in \{1..n\} : k \neq i$

$$VC_m[k] \leq VC_{p_j}[k]$$

This implies that all causally preceding messages to m already sent by other group members have already been delivered in this receiving process.

This means that the message is buffered and not yet delivered as long as these conditions are not satisfied. When p_j delivers m, it also increases the local entry belonging to p_i (i.e., $VC_{p_j}[i] := VC_m[i]$).

Let us consider now the example Ex_1 previously shown in Fig. 2, discussing the vector clocks assigned by each process to each of the relevant events of that execution (see Fig. 5):

1. The system is composed of four processes, so all vector clocks are initialised to $[0,0,0,0]$.
2. The first event is the multicast of message m_1 by p_1. So, p_1 sets its vector clock to $[1,0,0,0]$ and m_1 also gets that timestamp.
3. Once such message is received, all processes verify that the delivery conditions are satisfied (i.e., the sender slot in the message is one unit greater than the same slot in the receiver's clock). Therefore, these processes deliver immediately m_1 and update their own clock to $[1,0,0,0]$.
4. Later, p_3 multicasts m_2 and to this end it sets its clock to $[1,0,1,0]$. Message m_2 is also timestamped with that value.
5. When m_2 is received, p_1, p_2 and p_3 already had a value greater than 0 in the first slot. So, they deliver m_2 without blocking. On the other hand, p_4 still had value $[0,0,0,0]$. So, it maintains m_2 in its receiving buffer until m_1 is delivered. As a result of m_2's delivery, p_2, p_3 and p_4 set their clock to $[1,0,1,0]$, whilst p_1 sets it to $[2,0,1,0]$ since it has previously multicast message m_4 as described in step 7.
6. Later, p_2 multicasts m_3 tagged with $[1,1,1,0]$. Such message is delivered in p_2 and p_3 without any problems, setting p_3's clock to $[1,1,1,0]$, too.

   When m_3 is received by p_1, p_1 still has a clock $[1,0,0,0]$. So, it is unable to deliver it, since its third slot should have value 1 in order to allow such delivery. This only happens once p_1 has multicast m_4 and received m_2, a bit later.

On the other hand, p_4 receives m_3 once it has already delivered m_4. There is no blocking in the delivery of that pair of messages since they are concurrent ($VC_{m_3} = [1,1,1,0]$ and $VC_{m_4} = [2,0,0,0]$) and p_4's clock ($[1,0,1,0]$) allows the delivery of any of them, since it maintains a sender's slot value one unit lower than that carried in the message and the other slot values are greater or equal than those included in such message.

7. Finally, p_1 sends m_4 with timestamp $[2,0,0,0]$. Later it receives and delivers m_2, setting its clock to $[2,0,1,0]$ and enabling the delivery of m_3 (updating p_1's clock to $[2,1,1,0]$) and m_4.

   A bit later, p_2, p_3 and p_4 are able to deliver m_4 without problems. All processes terminate with a vector clock with value $[2,1,1,0]$.

This technique has managed execution Ex_1 as initially described at the end of Section 2. Note that $m_2 \parallel m_4$ and $m_3 \parallel m_4$ since their pairs of vector clocks are incomparable[1] (i.e., in a per-slot comparison, not all the slots in a vector are lower than or equal to those in the other vector): $VC_{m_2} = [1,0,1,0]$, $VC_{m_3} = [1,1,1,0]$ and $VC_{m_4} = [2,0,0,0]$. On the other hand, $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$, since $VC_{m_1} = [1,0,0,0] < VC_{m_2} = [1,0,1,0] < VC_{m_3} = [1,1,1,0]$.

Advantages. Vector clocks are able to ensure that $VC(a) < VC(b) \Rightarrow a \rightarrow b$, whilst the original logical clocks cannot ensure such property. However, the rules to compare clocks are also more complex. Because of this, all causal dependencies among multicast messages can be easily tracked with vector clocks. This removes the need of including the causally precedent messages in each multicast. As a result, message sizes are compacted and scalability on size is improved.

Drawbacks. Unfortunately, the non-blocking behaviour of the previous family of protocols has been lost in this case. When a message is dropped or when the routing being used leads to unordered reception, messages are prevented from being delivered until the missed messages are appropriately re-sent, received and delivered. As a result, these protocols may block message delivery in some cases.

Moreover, although message size has been reduced, vector clock size still depends on the amount of processes that compose the system where such messages are multicast. Note also that this multicast service is commonly used by highly available applications that will exist for a long time. Thus, the vector clock slots need to hold long integer numbers. If there are a lot of processes, vector clocks will still be very large, and this might again compromise the scalability of this service. So, in order to be highly scalable, a system that plans to use a causal multicast service based on vector clocks should consider the size of those clocks to be included in each multicast message; e.g., with vector slots of b bits in a system with n processes, we are adding a "tail" of $(b * n/8)$ bytes to each multicast message. For instance, in a system with 200 processes, with vector slots of 32 bits, this adds 800 bytes of content to each message. This could be a non-negligible communication overhead for a scalable service. However, replicated services seldom use as many replicas. When they need to maintain a lot of context or data, they partition such data and each fragment is managed by a disjoint set of replicas, using *sharding* [61]. So, typical data replica cardinality seldom exceeds 10 replicas and this only introduces an overhead of 40 bytes per message in this example.

Historical review. The first protocols developed for the vector-clock-based approach [13] (anticipated in [55, 52], though for point-to-point communication instead of multicasts) turned out to be efficient and scaled better than those of the previous generation. Most existing GCSs use causal multicast protocols of this kind (Transis [6], Spread [5], JGroups [10],...).

Other subsequent proposals are based on this kind of causal history information and address other important issues. For instance, Schiper and Pedone [56] propose a protocol for open groups. In *open groups* [19], not only the members of the system group can multicast messages to its members, but any other process can. Most of the surveyed multicast protocols were intended for *closed groups* where only group members are able to multicast messages, compelling external clients to forward their messages to any internal process that later multicasts each message.

Almeida et al [3] propose a mechanism for bounding the size of the elements being used in version vectors. It is intended for point-to-point communication and demands the transmission of a short list of previous versions. So, it still combines the usage of vector versions (that are locally maintained in this

[1] Vector clocks are compared as follows:
$VC(a) \leq VC(b)$ iff $\forall i, i \in \{1..n\} : VC(a)[i] \leq VC(b)[i]$, and
$VC(a) < VC(b)$ iff $VC(a) \leq VC(b) \wedge \exists i : VC(a)[i] < VC(b)[i]$

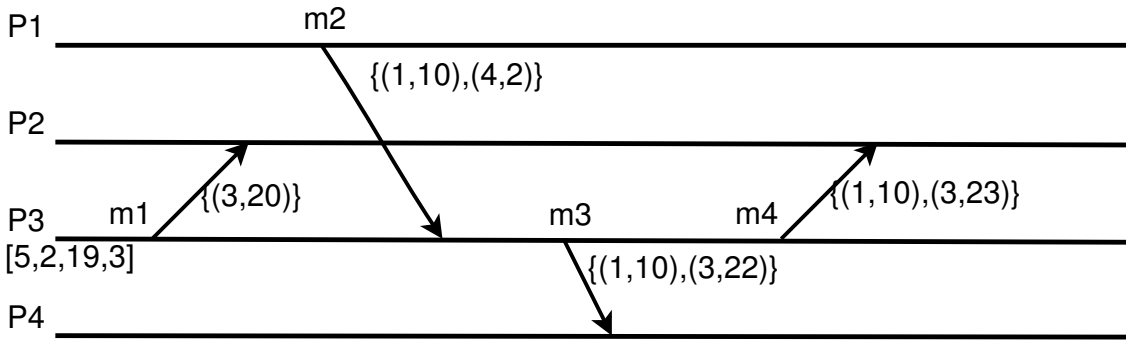

Figure 6: Execution Ex_3 with compacted clocks (unicast messages).

approach) with the transmission of some historical information in the messages being sent. With this, each node is able to prune the "history" being maintained in each one of the version vector components. Because of this, the values being used in those vector slots can be bounded and restarted soon, limiting them to a moderate size (e.g., in a system with N nodes, the upper bound can be set to N^2). This approach is able to remove the last drawback identified above although some adaptation is still needed for applying it to vector clocks, since it needs a thorough revision of the vector comparison operator when vector slots assume a numerical domain.

3.3 Compaction Approach

The vector compaction approach tries to minimise the size of the vector clocks being transferred in each multicast. To this end, only those slots that have changed and are still unknown for the receiving processes are included in the message timestamp. But there are two different mechanisms able to decide which slots need to be included: one for general causal communication [58] (i.e., point-to-point sendings), and another specific to multicast-based communication [13].

Let us start describing the general approach [58] intended for point-to-point communication. Each process p_i should maintain two additional vector clocks, named LU_{p_i} –last update– and LS_{p_i} –last sent–, besides VC_{p_i} as explained above. The $LU_{p_i}[j]$ holds the value of $VC_{p_i}[i]$ when p_i last updated its component j. In other words, to remember the state of p_i when it delivered the last message from p_j. On the other hand, $LS_{p_i}[j]$ maintains the value of $VC_{p_i}[i]$ when p_i sent its last message to p_j. Finally, when p_i sends a new message to p_j it only needs to transfer the entries $VC_{p_i}[k]$ (with $k \neq j$) such that $LS_{p_i}[j] < LU_{p_i}[k]$; i.e., it should propagate the sequence of pairs $\langle k, VC_{p_i}[k] \rangle$ that comply with such condition.

Let us illustrate how this compaction works with the example shown in Fig. 6 of a system with four processes. To this end, our description is focused on the state managed by process p_3. Its state when such execution fragment starts is:

Index	VC_{p_3}	LU_{p_3}	LS_{p_3}
1	5	17	16
2	2	10	18
3	19	19	–
4	3	8	15

To begin with, p_3 sends a causal message m_1 to p_2. So, it checks whether $LS_{p_3}[2]$ (whose value is 18) is lower than the $LU_{p_3}[k]$ values for $k \in \{1,3,4\}$. Only its own slot satisfies such condition, so m_1 only receives $\{(3,20)\}$ as its timestamp. As a result, the updated p_3's state is:

Index	VC_{p_3}	LU_{p_3}	LS_{p_3}
1	5	17	16
2	2	10	**20**
3	**20**	**20**	–
4	3	8	15

Then, process p_1 sends m_2 to p_3 with timestamp $\{(1,10),(4,2)\}$. So, p_3 delivers m_2 and updates its state as follows:

Index	VC_{p_3}	LU_{p_3}	LS_{p_3}
1	**10**	**21**	16
2	2	10	20
3	**21**	**21**	–
4	3	8	15

Note that $VC_{p_3}[4]$ has not been updated, since its value was already greater than that propagated in m_2's timestamp.

Later, p_3 sends m_3 to p_4. It checks whether $LS_{p_3}[4]$ (with value 15) is lower than the $LU_{p_3}[k]$ values for $k \in \{1,2,3\}$. The slots from p_1 and p_3 satisfy such condition, so m_3 uses $\{(1,10),(3,22)\}$ as its timestamp. As a result, the updated p_3's state is:

Index	VC_{p_3}	LU_{p_3}	LS_{p_3}
1	10	21	16
2	2	10	20
3	**22**	**22**	–
4	3	8	**22**

The same happens when p_3 sends m_4 to p_2. The timestamp for m_4 is $\{(1,10),(3,23)\}$ and the final state matrix is:

Index	VC_{p_3}	LU_{p_3}	LS_{p_3}
1	10	21	16
2	2	10	**23**
3	**23**	**23**	–
4	3	8	22

The compaction approach has not achieved impressive results in this execution because there are only a few processes in the assumed system. However, in a system like this with only four processes, only two bits are needed to maintain the process identifier. So, if the regular slot length is that of an integer (32 or 64 bits), the benefits of this compaction solution are clear.

In case of using only causal multicasts, the LS_{p_i} vector clock will not be needed since all its entries will hold the value $VC_{p_i}[i]-1$. On the other hand, the $LU_{p_i}[j]$ entries maintain in this case the value of $VC_{p_i}[i]$ when p_j multicast was delivered in p_i. This allows a trivial optimisation: instead of maintaining such LU_{p_i} vector clock, p_i only needs to recall the value of its VC_{p_i} clock when it multicast its last message (let us name it as LM_{p_i}), and p_i only needs to transfer those entries $k \neq i$ such that $LM_{p_i}[k] < VC_{p_i}[k]$. So, this compacting approach is equivalent to the one described by Birman et al [13] that is outlined at the end of this section.

Applying such compaction to the Ex_1 execution previously depicted in Fig. 2 and Fig. 5, and assuming that all processes initialise their vector clocks to $[0,0,0,0]$, the Ex_1's compacted version is shown in Fig. 7.

Advantages. The described compaction approach makes sense when the set of processes that multicast messages in a given group is localised; i.e., not all group members attain the same sending rates and many of them hardly send any message, at least at times. Thus, the processes that multicast more frequently will be able to compact a lot the vector clocks associated to their messages.

Assuming that n is the number of system processes, b is the number of bits needed to maintain a sequence number (i.e., the value of a vector clock entry), m is the number of bits needed to maintain a process identifier, and k is the proportion of clock entries that need to be propagated in each multicast, this technique is convenient when [58]: $k < \frac{n*b}{m+b}$. Note that $n*b$ is the size of a non-compacted vector clock and $m+b$ is the size of a compacted entry that needs to be propagated. This expression is easily held in most systems, as proven by Chandra et al [15].

Finally, although it was initially presented as a technique to improve scalability, it is able to provide important bandwidth saves even in systems that consist of a not so large number of processes (e.g., the performance analysis given in [15] does only consider systems with less than 100 processes).

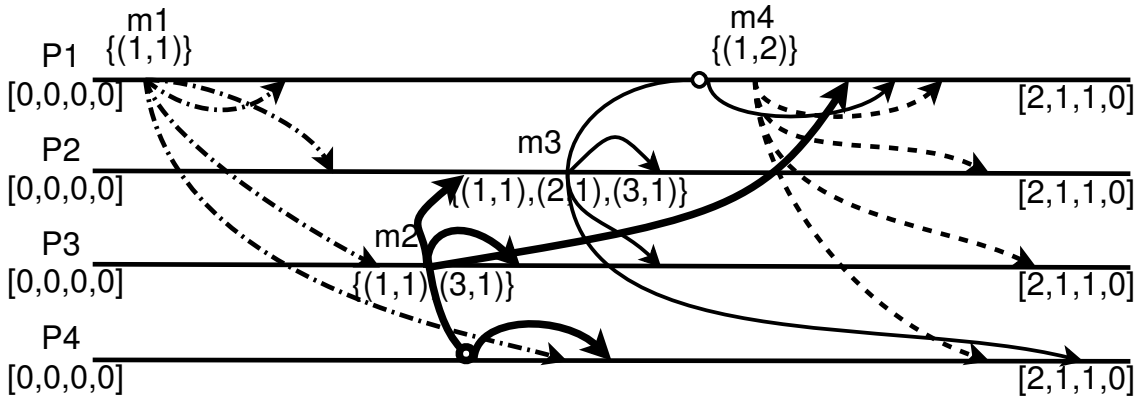

Figure 7: Execution Ex_1 with compacted clocks (bcast messages).

Drawbacks. In the general case, the usage of the described compaction approach reduces the needed bandwidth. However, this does not come for free since each process needs to maintain an additional copy of its vector clock. Nevertheless, that is a negligible space since the same amount of memory is added in every message when this compaction technique is not used.

Note that the compaction mechanism for point-to-point communication has only been described for completeness. Contrarily to what it suggests, the management of the multicast-oriented compaction approach is easy. Indeed, the slots to be transmitted may be progressively appended to a list as soon as any change arises. As a result, the computing overhead being introduced for managing these compacted vector clocks in the sender process directly depends on the rate of vector changes. So, that overhead is determined by the multicast traffic being originated by the other group members.

Historical review. As already indicated, the main problem of this new generation of protocols was the length of the vector clocks. A first solution was already presented by Birman et al [13]: it consisted in compacting clocks by recording only incremental changes. More precisely, for each new multicast message, each sender only sets those vector slots that have changed since its last multicast message. However, the reduction being achieved depends on the message transmission rate from each sender and on the number of system processes. Thus, other compacting approaches were due.

Inspired in the compacting approach from [13], a more general compacting solution was proposed by Singhal and Kshemkalyani [58]. It was later refined and formalised in [51, 31]. The degree of compaction of these solutions was evaluated in [15]. That evaluation showed that, in all analysed scenarios, the solution in [58] can avoid any penalty of additional space requirements, and that, in an optimal scenario, it is able to reduce the vector length to only 8% of its original size. In [49], the compacted causal clock information is propagated in (sometimes separate) *light control messages* (LCM). The performance evaluation given in that paper also proves that these mechanisms are able to enhance scalability.

Another compacting solution was described by Mostéfaoui and Raynal [43]. Although that solution was focused on the particular case of multiple groups that share some of their processes, it essentially used the same key ideas described above. It was able to manage multicasts to multiple groups using a single vector clock, with as many entries as groups. This is cheaper than the solutions given in [13, 60], but sometimes it also needs resynchronisation messages. In some way, this can also be seen as a precursor of the *subgroup interconnection* technique.

3.4 Interconnection Approach

In an interconnection scenario [21, 4], we assume that there are multiple subgroups that already have an internal causal multicast service. At least one process in each subgroup is chosen as its *interconnection server* (IS). The IS runs an interconnection protocol, ensuring that all messages initially multicast in its local subgroup are eventually delivered in all other interconnected subgroups. Analogously, all the messages multicast in remote subgroups are also eventually delivered in the local subgroup. Note that the IS provides

a regular application process interface to its local multicast service; i.e., it must not interfere with the local multicast protocol execution.

Causal separators [54] provide the basis to implement this kind of interconnection protocols, since they isolate each subgroup as a different causal zone. The fact that causal history does not need to be forwarded to other subgroups was proven by Rodrigues and Veríssimo [53]. The resulting interconnection protocol only needs either a FIFO transmission of the messages [27] when the interconnection servers are paired, or an independent (i.e., not related in any way to the internal multicast protocols used in the subgroups) causal multicast protocol executed by the set of all interconnection servers [9].

Although the complete proofs can be found in the cited papers, their justification is intuitive. Let us assume that our system S consists in the interconnection of k subgroups S_i ($1 \leq i \leq k$). Let $prec(m)$ be the set of messages that causally precede message m. Let $IS(S_i)$ be the interconnection server of subgroup S_i. Message m only needs to carry as its causal history some subset of $prec(m)$, and it is easy to argue that this subset is empty. Without loss of generality, let us assume that m was initially multicast in S_1. For each m' in $prec(m)$, m' was delivered to $IS(S_1)$ before m, since $m' \rightarrow m$. Thus, $IS(S_1)$ was able to propagate m' to all other S_i ($i \neq 1$) before m was propagated, since IS processes use FIFO channels [27] or a causal interconnection protocol [9] for message propagation. As a result, all messages in $prec(m)$ have been transferred and remulticast in each subgroup S_i before m is propagated. Since each message in $prec(m)$ has been remulticast in each subgroup S_i ($i \neq 1$) by the same $IS(S_i)$ process, all of them are causally related in such subgroups (since they have all been sent by the same sender, and causal order implies FIFO order). Thus, they should and will be delivered in their remulticast order. This eliminates the need of any causal history in the interconnection protocol.

Advantages. The first advantage of an interconnection solution is that it limits the scope of the causal history needed for implementing causal delivery. It is only used internally, in each subgroup. This guarantees that the size of such causal history is always kept small and does not depend on the overall size of the complete system. So, that facilitates the scalability of the causal multicast service.

Interconnection protocols allow [27] that a particular sender chooses either to multicast a message to its local subgroup or to the entire system.

Usually, each one of the interconnected subgroups has internal access to a very fast computer network, whilst the links used for implementing the interconnection have limited bandwidth [54]. Compared with deploying a single causal multicast protocol among all processes, the interconnection approach minimises the number of packets needed for implementing the multicast service through these links with limited bandwidth. Recall that the interconnection protocol only requires either a FIFO or a causal communication among the interconnection servers of each connected subgroup, and that only requires a single message to be communicated for each multicast message. Without an interconnection solution, the sender would have to emit multiple point-to-point messages, one for each receiver, and this could require that those inter-subgroup links were traversed multiple times for a given multicast. Thus, without interconnection the overall delivery delay would increase, since such low-bandwidth links will be easily saturated.

Finally, the interconnection approach seems to be the ideal candidate for dealing with *dynamic distributed systems*, i.e., those that may have a highly variable membership [44]. Note that the modularity of this approach allows that each interconnected subgroup may use internally any reliable causal multicast protocol. So, the implementation of such intra-subgroup protocols is not concerned with the global system membership. Additionally, some of the surveyed papers have shown that intra-subgroup protocols may also not depend on its own membership (e.g., the protocols [29] based on GPS modules for achieving a physical clock synchronisation, thus ensuring causal delivery by timestamping messages). As a result, that kind of solutions is able to easily deal with dynamic systems, ensuring thus an acceptable level of scalability for modern applications.

Drawbacks. Although an interconnection server introduces a short forwarding step, some papers (e.g., [49]) criticise that such application-level processing introduces a delay that might not be affordable by all kinds of applications. For instance, media stream multicasting should comply with soft real-time constraints and demand specific causal multicast protocols that tolerate message losses [63, 8, 11].

Kalantar and Birman [28] report a similar problem that is not specific of interconnected subgroups, since it also arises in any kind of overlapping groups: the *convoy phenomenon*. Processes that belong to more than one group (as the *interconnection servers*) introduce, at times, delays that generate a bursty propagation behaviour. Since they act as "bridges" between subgroups and multicasts follow a specific

order (causal in this case), when any message cannot be propagated (due to the temporary loss of a preceding message or because of a missed acknowledgement that might compromise the *agreement* in reliable multicast delivery) some subsequent messages get blocked until such propagation is resumed. If messages should cross multiple subgroups in order to get delivered (as it happens in a hierarchical structure with more than two layers), these delays in one of the propagation servers originate further delays in the remaining servers of that path, causing a *bursty* propagation that compromises scalability.

The solution to that problem has been already advanced in previous sections. Since the convoy phenomenon is caused by a pause in forwarding processes, we could use any of the available mechanisms for avoiding pauses. In the common case, a forwarding pause is caused by a missed precedent message. So, we could use any of the first generation protocols that include precedent messages in order to avoid such situations. System architecture should take this into account, using small subgroups that might only accumulate a few precedent messages in each multicast.

If failures arise, the reconfiguration and recovery of the system could need a non-negligible time if any of the interconnection servers has crashed. In such case, another server process should be chosen in that affected subgroup and its identity should be reported to all other subgroups.

Applications demanding a strong consistency among their component replicas usually require a global view management, even when those replicas are placed in different interconnected subgroups. The existing interconnecting protocols have not been designed considering a global view management. Fortunately, reliable causal multicast provides a basis for supporting causal consistency, and this consistency model cannot be qualified as strong; so such global view management will not be a problem in the general case.

Historical review. Meldal et al [41] proposed different ways for compacting the vector clocks used in static multi-centre topologies. They proved that when all subgroup interconnection channels are known in advance, vector clocks can be highly compacted. They also specified general rules to accomplish such compaction and showed that several topologies (e.g., a star) are able to generate minimal clock-related information, whilst others (e.g., a ring) cannot reduce it at all.

Such rules were used by Adly and Nagi [1], where a hierarchical topology was proposed, compacting the used vector clocks and enhancing thus its scalability.

This solution was further improved by Rodrigues and Veríssimo [54] by the introduction of the *causal separator* concept. Causal separators act as communication *gateways*, forwarding the messages transmitted among two or more subgroups, named *causal zones*. Any causal history information (e.g., vector clocks) used in each causal zone does not need to be known by the processes belonging to other causal zones. As a result, a system consisting of many processes could be divided into multiple causal zones with several causal separators. Hence, the size of the vector clocks managed by each process could be reduced, making thus practical the usage of the causal protocols described already in Section 3.2, since the causal history overhead may become negligible with an appropriate system division. Finally, [54] also show that FIFO point-to-point communication is enough for interconnecting causal zones.

The usage of causal separators was later refined by Baldoni et al [9]. The latter limited the number of causal separators in a given causal zone to be a single one, its *causal server*. In order to interconnect the causal servers of multiple groups, a regular causal multicast protocol should be used. This generates a hierarchical structure named *daisy architecture* [9].

This was the first example of a new scalability approach known as (non-intrusive) *subgroup interconnection* [21]. The best property of an interconnecting solution is that the interconnected subgroups may internally use the protocol of their choice that provides a suitable semantics (reliable causal delivery in the context of this paper, but subgroup interconnection can also be applied to other kinds of ordered delivery). There should exist an interconnecting protocol that is compatible with the same semantics, but it does not depend on the protocols being internally used in each subgroup. Although the protocols described in [54] were already able to ensure this property, Baldoni et al [9] were the first to explicitly state it.

Other papers presenting other interconnection solutions were published later. For instance, Johnson et al [27] describe solutions to interconnect causal and total-order multicast protocols.

Causal interconnection protocols only demanded reliable FIFO communication when two subgroups are interconnected, and also the avoidance of interconnection cycles when more than two subgroups are managed [27, Theorem 7]. In a similar way, Meldal et al [41] proved that the amount of clock-related information to be included in multicast messages can be reduced if the connecting paths are acyclic; also Stephenson [60] and Mostéfaoui and Raynal [43] proved that acyclicity is needed. On the other hand,

when total-order interconnection is intended, [60, 43] show that an intrusive interconnecting protocol is necessary; i.e., a multicast message cannot be locally delivered in the sending subgroup until the forwarder process has propagated the message and a global total order has been decided. Formal proofs of the impossibility of achieving total-order interconnection in a non-intrusive way were later given by Laumay et al [35] and Álvarez et al [4].

An alternative kind of interconnecting technique that also illustrates its modularity is described by Kawanami et al [29, 30, 45]. This approach does not use inside each interconnected subgroup any of the techniques previously presented in this paper, but a simple algorithm based on physical timestamping. To this end, it timestamps all multicast messages with the local clock of its sender process. This requires a physical clock synchronisation, which can be achieved using GPS modules in the host computers. Prakash and Baldoni [50] state that such level of physical synchronisation is sufficient. In order to implement the interconnection, a regular vector clock-based protocol is used by Kawanami et al [29], whilst the approaches in the other two papers either work with nothing but the original Lamport clocks [45], or those clocks combined with the physical timestamp used in the sender subgroup [30].

The interconnection modularity enables the protocols to be appropriately tailored and optimised with regard to the particularities of each system. Thus, if an intra-subgroup local network has high bandwidth and low delays, as in the cloud infrastructures mentioned by Matos et al [38] as a federation of data centres, then its performance can be very high, tolerating high internal message multicast rates. However, inter-subgroup channels do not provide high bandwidth. So, having a single interconnecting server per subgroup may become a bottleneck that limits the overall scalability. To overcome this limitation, multiple servers per subgroup could be deployed, thus setting multiple interconnecting channels between each pair of subgroups. That option has been analysed by de Juan-Marín et al [18], involving a simple subprotocol that ensures an overall FIFO order among all interconnection channels for each pair of interconnected subgroups, increasing thus the interconnection bandwidth.

4 Discussion

In order to sum up, Table 1 depicts the main characteristics of each presented solution. The multicasting mechanisms shown in that table are: *First Generation*, *Vector Clocks*, *Compaction Approach* and *Interconnection Approach*.

The characteristics being considered in such table rows are:

Causal History. It refers to how the solution ensures causal order. The possible values are: inclusion of causal precedent messages, vector clocks or compacted vector clocks.

Structure. This highlights whether the solution should have a logical structure of processes or not. This might introduce some overhead in case of failures, since such structure needs to be (partially) rebuilt when some processes fail.

Advantages. They can be: non-blocking delivery (Non-blck), or usage of small messages (Msg size) due to a minimisation of the causal history information being appended to each message.

Drawbacks. Different drawbacks are considered: usage of large messages (big msg), and blocking behaviour (blocking).

The interconnection approach need not be blocking, since it may use in each subgroup any causal history format, even causally precedent messages. Moreover, the interconnecting protocol need not transfer any causal history. Regarding internal message size, interconnected subgroups may use small messages when a non-blocking alternative is chosen. This is achieved selecting a small subgroups size, minimising thus the amount of precedent messages in each multicast.

Regarding adaptability to varying workloads, the interconnection approach is the best option since it does not demand any global membership management. View management is accomplished in each subgroup. This allows a precise implementation with negligible overhead, since all the internal communication channels will provide similar delays and bandwidth, using a fast network in most cases. Workload variations are easily managed, even when a lot of servers are involved in the scaling out actions. To this end,

	Multicasting Approaches			
	First generat.	*Vector clocks*	*Compact. approach*	*Intercon. approach*
Causal history	precedent messages	vector clocks	compact clocks	any
Structure	no	no	no	yes
Advantages	non-blck	–	msg size	msg size
Drawbacks	big msg	blocking	blocking	–

Table 1: Characteristics summary.

administrators do only require an appropriate subgroup deployment, removing or adding entire subgroups to the current system when needed.

For each causal multicast mechanism, the following list identifies the requirements of the applications for which the respective mechanism could be used:

1. *First generation*: Applications that need a non-blocking delivery of messages and are able to tolerate large messages (e.g., deployed in high-bandwidth networks).

2. *Vector clocks*: Applications that tolerate blocking delivery and are interested in small messages (e.g., low-end LANs and WANs), with low scaling-out needs.

3. *Compaction*: Applications that tolerate blocking delivery and are interested in the smallest messages (e.g., when deployed in WANs), with low scaling-out needs.

4. *Interconnection*: Applications with small message size on average and moderate-to-high scaling-out needs, that can be deployed on modular sets of processes.

Causal multicasts support causal consistency. Causal consistency implemented through an interconnection-based causal multicast protocol may support eventual consistency [47], dealing with the limitations imposed by the CAP theorem [24] since causal consistency is partition-tolerant and it does not compromise service continuity (i.e., availability). Each causal subgroup may be deployed in a different data centre and the interconnection protocol implicitly implements the update propagation tasks needed in order to reach a convergent state once a network partition has been repaired.

5 Conclusions

We have historically surveyed several mechanisms used for implementing reliable causal multicast. The first protocols included in each multicast message its own causal history; i.e., a set of precedent messages not yet delivered in all system processes. Thus, the causal order could be enforced in all receivers. That was simplified when vector clocks were introduced, reducing the size of the multicast messages. Some amount of causal information (the vector clocks themselves) is still kept, in proportion of the system size. In order to improve the scalability of these protocols, some kind of vector clock compaction mechanism may be used. We have found two types of compaction. The first only transmits the vector entries that have been modified since the last message multicast by the same sender. The second uses the interconnection principle in order to manage vector clocks only inside each subsystem. Thus, the vector size only depends on the amount of processes in each subsystem but not on the global system size. These approaches can be easily combined, for enabling reliable message multicasts in large systems and for facilitating the use of causal consistency in scalable applications. When causal consistency is combined with lazy propagation, eventual consistency is achieved. This further improves the scalability of replicated services.

Acknowledgements

This work was supported by *European Regional Development Fund* (FEDER) and *Ministerio de Economía y Competitividad* (MINECO) under research grant TIN2012-37719-C03-01.

References

[1] Adly N, Nagi M (1995) Maintaining causal order in large scale distributed systems using a logical hierarchy. In: IASTED Intnl Conf on Appl Inform, pp 214–219

[2] Aguilera MK, Chen W, Toueg S (1997) Heartbeat: A timeout-free failure detector for quiescent reliable communication. In: 11th Intnl Wshop on Distrib Alg (WDAG), Saarbrücken, Germany, pp 126–140

[3] Almeida JB, Almeida PS, Baquero C (2004) Bounded version vectors. In: 18th Intnl Conf on Distrib Comput (DISC), Amsterdam, The Netherlands, pp 102–116

[4] Álvarez A, Arévalo S, Cholvi V, Fernández A, Jiménez E (2008) On the interconnection of message passing systems. Inform Process Lett 105(6):249–254

[5] Amir Y, Stanton J (1998) The Spread wide area group communication system. Tech. rep., CDNS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins Univ.

[6] Amir Y, Dolev D, Kramer S, Malki D (1992) Transis: A communication subsystem for high availability. In: 22nd Intnl Symp on Fault-Tolerant Comp (FTCS), Boston, MA, USA, pp 76–84

[7] Anastasi G, Bartoli A, Spadoni F (2001) A reliable multicast protocol for distributed mobile systems: Design and evaluation. IEEE Trans Parallel Distrib Syst 12(10):1009–1022

[8] Baldoni R, Raynal M, Prakash R, Singhal M (1996) Broadcast with time and causality constraints for multimedia applications. In: 22nd Intnl. Euromicro Conf., Prague, Czech Republic, pp 617–624

[9] Baldoni R, Friedman R, van Renesse R (1997) The hierarchical daisy architecture for causal delivery. In: 17th Intnl. Conf. on Distrib. Comput. Syst. (ICDCS), Baltimore, Maryland, USA, pp 570–577

[10] Ban B (2002) JGroups - a toolkit for reliable multicast communication. Available at: http://www.jgroups.org

[11] Benslimane A, Abouaissa A (2002) Dynamical grouping model for distributed real time causal ordering. Comput Commun 25:288–302

[12] Birman KP, Joseph TA (1987) Reliable communication in the presence of failures. ACM T Comput Syst 5(1):47–76

[13] Birman KP, Schiper A, Stephenson P (1991) Lightweigt causal and atomic group multicast. ACM T Comput Syst 9(3):272–314

[14] Cachin C, Guerraoui R, Rodrigues LET (2011) Introduction to Reliable and Secure Distributed Programming (2. ed.). Springer

[15] Chandra P, Gambhire P, Kshemkalyani AD (2004) Performance of the optimal causal multicast algorithm: A statistical analysis. IEEE T Parall Distr 15(1):40–52

[16] Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267

[17] Chockler G, Keidar I, Vitenberg R (2001) Group communication specifications: a comprehensive study. ACM Comput Surv 33(4):427–469

[18] de Juan-Marín R, Cholvi V, Jiménez E, Muñoz-Escoí FD (2009) Parallel interconnection of broadcast systems with multiple FIFO channels. In: 11th Intnl Symp on Distrib Obj, Middleware and Appl (DOA), Vilamoura, Portugal, LNCS, vol 5870, pp 449–466

[19] Défago X, Schiper A, Urbán P (2004) Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput Surv 36(4):372–421

[20] Demers AJ, Greene DH, Hauser C, Irish W, Larson J, Shenker S, Sturgis HE, Swinehart DC, Terry DB (1987) Epidemic algorithms for replicated database maintenance. In: 6th ACM Symp on Princ of Distrib Comput (PODC), Vancouver, British Columbia, Canada, pp 1–12

[21] Fernández A, Jiménez E, Cholvi V (2000) On the interconnection of causal memory systems. In: 19th Annual ACM Symp on Princ of Distrib Comput (PODC), Portland, Oregon, USA, pp 163–170

[22] Fidge CJ (1988) Timestamps in message-passing systems that preserve the partial ordering. In: 11th Australian Comput Conf, pp 56–66

[23] Friedman R, Vitenberg R, Chockler G (2003) On the composability of consistency conditions. Inform Process Lett 86(4):169–176

[24] Gilbert S, Lynch N (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2):51–59

[25] Gray J, Helland P, O'Neil PE, Shasha D (1996) The dangers of replication and a solution. In: SIGMOD Conf, pp 173–182

[26] Hadzilacos V, Toueg S (1993) Fault-tolerant broadcasts and related problems. In: Mullender S (ed) Distributed Systems, 2nd edn, ACM Press, chap 5, pp 97–145

[27] Johnson S, Jahanian F, Shah J (1999) The inter-group router approach to scalable group composition. In: 19th Intnl Conf on Distrib Comput Syst (ICDCS), Austin, TX, USA, pp 4–14

[28] Kalantar MH, Birman KP (1999) Causally ordered multicast: the conservative approach. In: 19th Intnl Conf on Distrib Comput Syst (ICDCS), Austin, TX, USA, pp 36–44

[29] Kawanami S, Enokido T, Takizawa M (2004) A group communication protocol for scalable causal ordering. In: 18th Intnl Conf on Adv Inform Netw Appl (AINA), Fukuoka, Japan, pp 296–302

[30] Kawanami S, Nishimura T, Enokido T, Takizawa M (2005) A scalable group communication protocol with global clock. In: 19th Intnl Conf on Adv Inform Netw Appl (AINA), Taipei, Taiwan, pp 625–630

[31] Kshemkalyani AD, Singhal M (1998) Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. Distrib Comput 11(2):91–111

[32] Kshemkalyani AD, Singhal M (2011) Distributed Computing: Principles, Algorithms, and Systems, 2nd edn. Cambridge University Press, New York, USA

[33] Ladin R, Liskov B, Shrira L, Ghemawat S (1992) Providing high availability using lazy replication. ACM T Comput Syst 10(4):360–391

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

[35] Laumay P, Bruneton E, de Palma N, Krakowiak S (2001) Preserving causality in a scalable message-oriented middleware. In: Intnl Conf on Distrib Syst Platf (Middleware), pp 311–328

[36] Liu N, Liu M, Cao J, Chen G, Lou W (2010) When transportation meets communication: V2P over VANETs. In: 30th Intnl Conf Distrib Comput Syst (ICDCS), Genoa, Italy

[37] Lwin CH, Mohanty H, Ghosh RK (2004) Causal ordering in event notification service systems for mobile users. In: Intnl Conf Inform Tech: Coding Comput (ITCC), Las Vegas, Nevada, USA, pp 735–740

[38] Matos M, Sousa A, Pereira J, Oliveira R, Deliot E, Murray P (2009) CLON: Overlay networks and gossip protocols for cloud environments. In: 11th Intnl Symp on Dist Obj, Middleware and Appl (DOA), Vilamoura, Portugal, LNCS, vol 5870, pp 549–566

[39] Mattern F (1989) Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms, North-Holland, pp 215–226

[40] Mattern F, Fünfrocken S (1994) A non-blocking lightweight implementation of causal order message delivery. Lect Notes Comput Sc 938:197–213

[41] Meldal S, Sankar S, Vera J (1991) Exploiting locality in maintaining potential causality. In: 10th ACM Symp on Princ of Distrib Comp (PODC), Montreal, Quebec, Canada, pp 231–239

[42] Mosberger D (1993) Memory consistency models. Operat Syst Review 27(1):18–26

[43] Mostéfaoui A, Raynal M (1993) Causal multicast in overlapping groups: Towards a low cost approach. In: 4th Intnl Wshop on Future Trends of Distrib Comp Syst (FTDCS), Lisbon, Portugal, pp 136–142

[44] Mostéfaoui A, Raynal M, Travers C, Patterson S, Agrawal D, El Abbadi A (2005) From static distributed systems to dynamic systems. In: 24th Symp on Rel Distrib Syst (SRDS), Orlando, FL, USA, pp 109–118

[45] Nishimura T, Hayashibara N, Takizawa M, Enokido T (2005) Causally ordered delivery with global clock in hierarchical group. In: ICPADS (2), Fukuoka, Japan, pp 560–564

[46] Parker Jr DS, Popek GJ, Rudisin G, Stoughton A, Walker BJ, Walton E, Chow JM, Edwards DA, Kiser S, Kline CS (1983) Detection of mutual inconsistency in distributed systems. IEEE Trans Software Eng 9(3):240–247

[47] Pascual-Miret L (2014) Consistency models in modern distributed systems. An approach to eventual consistency. Master's thesis, Depto. de Sistemas Informáticos y Computación, Univ. Politècnica de València, Spain

[48] Peterson LL, Buchholz NC, Schlichting RD (1989) Preserving and using context information in interprocess communication. ACM T Comput Syst 7(3):217–246

[49] Pomares Hernández S, Fanchon J, Drira K, Diaz M (2001) Causal broadcast protocol for very large group communication systems. In: 5th Intnl Conf on Princ of Distrib Syst (OPODIS), Manzanillo, Mexico, pp 175–188

[50] Prakash R, Baldoni R (2004) Causality and the spatial-temporal ordering in mobile systems. Mobile Netw Appl 9(5):507–516

[51] Prakash R, Raynal M, Singhal M (1997) An adaptive causal ordering algorithm suited to mobile computing environments. J Parallel Distr Com 41(2):190–204

[52] Raynal M, Schiper A, Toueg S (1991) The causal ordering abstraction and a simple way to implement it. Inform Process Lett 39(6):343–350

[53] Rodrigues L, Veríssimo P (1995) Causal separators and topological timestamping: An approach to support causal multicast in large-scale systems. Tech. Rep. AR-05/95, Instituto de Engenharia de Sistemas e Computadores (INESC), Lisbon, Portugal

[54] Rodrigues L, Veríssimo P (1995) Causal separators for large-scale multicast communication. In: 15th Intnl Conf on Distrib Comput Syst (ICDCS), Vancouver, Canada, pp 83–91

[55] Schiper A, Eggli J, Sandoz A (1989) A new algorithm to implement causal ordering. In: 3rd Intnl Wshop on Distrib Alg (WDAG), Nice, France, pp 219–232

[56] Schiper N, Pedone F (2010) Fast, flexible and highly resilient genuine FIFO and causal multicast algorithms. In: 25th ACM Symp on Applied Comp (SAC), Sierre, Switzerland, pp 418–422

[57] Shapiro M, Preguiça NM, Baquero C, Zawirski M (2011) Convergent and commutative replicated data types. Bulletin of the EATCS 104:67–88

[58] Singhal M, Kshemkalyani AD (1992) An efficient implementation of vector clocks. Inform Process Lett 43(1):47–52

[59] Sotomayor B, Montero RS, Llorente IM, Foster IT (2009) Virtual infrastructure management in private and hybrid clouds. IEEE Internet Computing 13(5):14–22

[60] Stephenson P (1991) Fast ordered multicasts. PhD thesis, Dept. of Comp. Sc., Cornell Univ., Ithaca, NY, USA

[61] Stonebraker M (1986) The case for shared nothing. IEEE Database Eng Bull 9(1):4–9

[62] Wischhof L, Ebner A, Rohling H (2005) Information dissemination in self-organizing intervehicle networks. IEEE T Intell Transp 6(1):90–101

[63] Yavatkar R (1992) MCP: A protocol for coordination and temporal synchronization in multimedia collaborative applications. In: 12th Intnl Conf on Distrib Comput Syst (ICDCS), Yokohama (Japan), pp 606–613

[64] Yen LH, Huang TL, Hwang SY (1997) A protocol for causally ordered message delivery in mobile computing systems. Mobile Netw Appl 2(4):365–372

[65] Zhou S, Cai W, Turner SJ, Lee BS, Wei J (2007) Critical causal order of events in distributed virtual environments. ACM T Mult Comp Comm Appl 3(3)