CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1801

Scalable group communication supporting configurable levels of consistency

Anders Gidenstam [1], Boris Koldehofe [2], Marina Papatriantafilou [3] and Philippas Tsigas [3,*,†]

[1] *University of Borås, Sweden*
[2] *University of Stuttgart, Germany*
[3] *Chalmers University of Technology, Sweden*

SUMMARY

Group communication is deployed in many evolving Internet-scale cooperative applications such as multiplayer online games and virtual worlds to efficiently support interaction on information relevant to a potentially very large number of users or objects. Especially peer-to-peer based group communication protocols have evolved as a promising approach to allow intercommunication between many distributed peers. Yet, the delivery semantics of robust and scalable protocols such as gossiping is not sufficient to support consistency semantics beyond eventual consistency because no relationship on the order of events is enforced. On the other hand, traditional consistency models provided by reliable group communication providing causal or even total order are restricted to support only small groups.

This article proposes the *cluster consistency* model which bridges the gap between traditional and current approaches in supporting both scalability and ordered event delivery. We introduce a dynamic and fault tolerant cluster management method that can coordinate concurrent access to resources in a peer-to-peer system and can be used to establish *fault-tolerant* configurable cluster consistency with predictable reliability, running on top of decentralised probabilistic protocols supporting scalable group communication. This is achieved by a general two-layered architecture that can be applied on top of the standard Internet communication layers and offers a modular, layered set of services to the applications that need them. Further, we present a *fault-tolerant* method implementing causal cluster consistency with predictable reliability, running on top of decentralised probabilistic protocols supporting group communication.

This paper provides analytical and experimental evaluation of the properties regarding the fault tolerance of the approach. Furthermore, our experimental study, conducted by implementing and evaluating the two-layered architecture on top of standard Internet transport services, shows that the approach scales well, imposes an even load on the system, and provides high-probability reliability guarantees. Copyright © 2011 John Wiley & Sons, Ltd.

Received 12 May 2011; Accepted 27 May 2011

KEY WORDS: Gossiping; Group communication; Optimistic causal order

1. INTRODUCTION

Collaborative environments such as *distributed interactive simulation* [1–3], *virtual worlds* [4] and *massively scalable online multiplayer games* [5,6] allow a possibly large set of concurrently joining and leaving processes to share and interact on a set of common replicated objects. State changes on the objects are distributed among the processes by update messages (also known as *events*). Providing the infrastructure to support such applications and systems places demands for group

*Correspondence to: Philippas Tsigas, Chalmers University of Technology, S-412 96 Göteborg, Sweden.
†E-mail: tsigas@chalmers.se

Contract/grant sponsor: Publishing Arts Research Council; contract/grant number: 98–1846389

communication, with guarantees on reliability, consistency and scalability, even in the presence of failures and variable connectivity of the peers in the system.

Because these properties are in general conflicting, research in group communication has so far investigated two main classes of group communication services: *reliable group communication* and *peer-to-peer-based* group communication. Reliable group communication [7–10] has contributed a lot to provide fault-tolerant group communication services and support for consistency, by addressing both causal order as well as total order delivery semantics. However, their scalability and robustness to frequently changing groups of processes is limited.

In contrast, peer-to-peer-based group communication [11–20] has reduced the requirements with respect to reliability and consistency by mainly focusing on highly dynamic group memberships and dissemination strategies. Although approaches like gossip-based group communication protocols [17–20] are promising to increase their robustness [21], peer-to-peer group communication protocols do not provide any guarantees on the order at which messages are delivered. Hence, they are not suited for stronger guarantees than eventual consistency.

Because the decisions of collaborative applications often depend on the input of multiple events, such applications could highly benefit from an event delivery service that satisfies the causal order relation; that is, it satisfies the 'happened before' relation as originally introduced by Lamport [22]. However, supporting scalable detection of causal relations between events is a challenging task, because the corresponding timestamps induces a large overhead that grows linear with the number of processes eligible to send events [23, 24]. Especially dynamic changes in the group membership require additional mechanisms to coordinate the sending of updates for large groups of processes.

In this paper, we propose a consistency management method denoted by *causal cluster consistency*, providing optimistic causal delivery of update messages to a large set of processes. Causal cluster consistency takes into account that for many applications, the number of processes which are interested in performing updates can be low compared with the overall number of processes which are interested in receiving updates and maintaining replicas of the respective objects. Therefore, the number of processes that are entitled to perform updates at the same time is restricted to n, which also corresponds to the maximum size of the timestamps used. However, the set of processes entitled to perform updates is configurable and may change dynamically.

We address this problem by first proposing a distributed cluster management. A cluster represents a region of interest in a peer-to-peer system, for example, it may consist of a set of resources or objects which processes would like to access. In our context, it is used to coordinate the management of entries to a vector clock by means of a ticketing mechanism. Each peer entitled to disseminate a message will receive a ticket corresponding to a unique vector clock entry. The cluster management ensures that never two processes will perform an action with respect to the same ticket. Still, the cluster management supports a decentralised organisation of the cluster and allows dynamic re-assignment of vector timestamps in the presence of failures and churn; that is, peers can continuously join and leave the cluster.

Furthermore, we present a two-layer architecture implementing causal cluster consistency. The approach can use lightweight communication algorithms like gossiping which can in turn run using standard Internet (or other) transport services. Our method is also designed to tolerate a bounded number of process failures, by using a combined push-and-pull (recovery) method. We also present an implementation and experimental evaluation of the proposed method and its potential with respect to reliability and scalability, by building on recently evolved large-scale and lightweight probabilistic group communication protocols. Our implementation and evaluation have been carried out in a real network and also in competition with concurrent network traffic by other users.

Structure of the article

In Section 2, we discuss related work on resource management in peer-to-peer applications. In Section 3, we describe the problem and introduce notation and definitions. Then we present two algorithms implementing a dynamic cluster management. The protocol of Section 4.1 works in the absence of failures and illustrates the basic idea, whereas Section 4.2 describes and proves a fault-tolerant membership protocol. In Section 5, we introduce a layered architecture for achieving

DOI: 10.1002/cpe

causal cluster consistency and the two-layered protocol implementing it. Section 6 discusses the implementation and experimental evaluation of the proposed protocol running on top of standard Internet transport services. We conclude with a discussion of the presented results and future work.

2. RELATED WORK

A large set of applications can be classified as collaborative environments. Originally deployed in the context of military simulations [1], they have been looked from the perspective of distributed interactive simulations [1–3] and more recently in the context of virtual worlds (e.g. [4]) and massively scalable distributed online games (e.g. [5, 6]). In general, they can be applied in many more settings such as education, social networks, control of business processes and many more. A characteristic for such applications is that the transfer of the full state of objects is costly in terms of bandwidth and latency. Therefore, distributed interactive simulations and distributed online games typically maintain replicas of objects locally and interact by sending update events.

This has early raised the question of the communication paradigm to exchange the updates such that consistency of the replicas is ensured. Although some applications try to resolve inconsistencies at the application level, for instance, by dead reckoning on position information, work in distributed simulation has realised already at an early stage the reliable group communication supporting causal and total order semantics is attractive in establishing such guarantees.

Research in distributed computing has come up with robust solutions for achieving reliable causal delivery in the occurrence of faults [7–10]. However, these approaches are not suitable to cope with variations in needs and behaviour. Further, because the causal order semantic requires that an event is delivered only after all causally preceding events have been delivered, the need to always recover lost messages can lead to long latencies for events, whereas applications often need short delivery latencies. Moreover, the latency in large groups can also become large because a causal reliable delivery service needs to add timestamp information, whose size grows with the size of the group, to every event.

To improve on latency, *optimistic causal order* [25, 26] can be suitable for systems where events are associated with deadlines. In contrast to the strict causal order semantics, optimistic causal order only ensures that no events that causally precede an already delivered event are delivered. Events that have become obsolete do not need to be delivered and may be dropped. Nevertheless, optimistic causal order algorithms aim at minimising the number of lost events. In order to determine the precise causal relation between pairs of events in the system processes can use *vector clocks* [27, 28], which also allow detection of missing events and their origin. However, because the size of the vector timestamps grows linearly with the number of processes in the system, one may need to introduce some bound on the growing parameter to ensure scalability.

Peer-to-peer based application layer multicast and group communication protocols such as [11–15, 17–20] aim to support large groups of dynamically changing processes. In general, these approaches can be classified in groups whose goal is (i) to built in a decentralised manner a dissemination tree [11, 12, 15, 16] or mesh [16] by relying on unstructured peer-to-peer networks; (ii) to embed a dissemination tree into a structured peer-to-peer overlay [13]; or (iii) to rely on a random graph topology [17–20] for robust event dissemination. Although dissemination trees perform most efficient in terms of message overhead, they are vulnerable to message loss during periods of failures. Birman *et al.* [17] have proposed gossiping as a paradigm to ensure reliable and scalable dissemination in the presence of failures. Subsequent work [18, 19] has also shown how to provide decentralised group membership for gossiping by relying on a partial view on the whole system. Moreover, the robustness of gossiping has been intensively analysed (cf. [21, 29] and several improvements accounting for aspects like buffer management [20] and security [30] have been proposed. In [5], it is shown that probabilistic group communication protocols can perform well also in the context of collaborative environments. However, per se, approaches to scalable and reliable group communication do not provide any ordering guarantees. To provide guarantees for ordered delivery, one needs control mechanisms that coordinate the concurrent dissemination of events as achieved by the cluster management protocol of this article.

DOI: 10.1002/cpe

The way structured peer-to-peer systems share information in the system (cf. e.g. [31–35]) has been of relevance and inspiration to this work. Note, however, that uniform hashing, as used in many peer-to-peer systems, is not suitable to solve the cluster management problem because the number of processes is expected to be larger than the number of available tickets in a cluster. Even in the situation of network partitioning, the cluster management needs to ensure that no two processes will create an event with respect to the same ticket.

One may notice some similarity between the problem in this article and the l-exclusion problem [36, 37] which allows at most l concurrent processes to enter the critical section. In contrast, the cluster management allows in a dynamically evolving set of processes to associate each process with a unique ticket. Nevertheless, the solution to the cluster management problem proposed here could also serve as solution basis to the l-exclusion problem.

Our approach integrates and builds on earlier of our work for multipeer systems [20, 29, 38, 39]. The aim of these work has been on providing *predictable reliability*; that is, guaranteeing that a certain condition holds at a set of destinations with high-probability.

3. NOTATION AND PROBLEM STATEMENT

Throughout the article, we will use the following model of a collaborative application (cf. Figure 1). The application is made up of a group denoted by $G = \{p_1, p_2, \ldots\}$ of dynamically joining and leaving processes. Furthermore, $B = \{b_1, b_2, \ldots\}$ denotes the set of replicated objects used by the collaborative application. Processes maintain their replicas of interest locally. Furthermore, we consider that B is partitioned into disjoint clusters $C_1, C_2, \ldots$ with $\cup_i C_i \subseteq B$, where for each cluster C_i all peers are interested in updates messages. Let C denote a cluster and p a process in G, then we write $p \in C$ if p is interested in objects of C. Note that the clustering of objects is very specific to the application. For instance, in a multiplayer game like *Planet* $\Pi 4$ [40], a simple partitioning according to the virtual coordinates of objects is applied. This ensures that state updates of objects like moving planes are only disseminated in a limited region of interest. The scalability of the game is increased by reducing the number of messages within a cluster. In order to be useful in a game, it is important that such a clustering supports dynamically changing interest as well as support the scalable, reliable and ordered dissemination of update messages.

Causal cluster consistency allows any processes in C to maintain the state of replicated objects in C by applying updates in optimistic causal order. However, at most n processes (n is assumed to be known to all processes in C) are eligible to propose updates to the objects in C at the same time. In the aforementioned game, this means, that the application unit sending update message for a specific plane should be a member of the cluster which coordinates the updates of the geographic region in which the plane is currently moving. At most n processes are eligible to send concurrently update

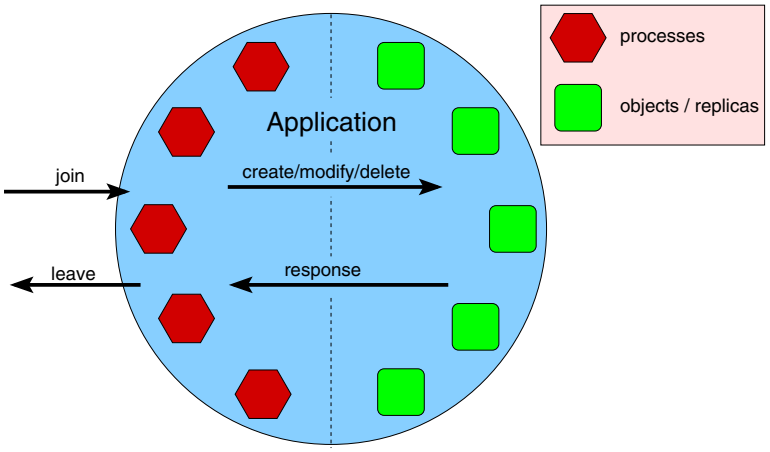


Figure 1. Illustration on the interaction between processes and objects in a collaborative application.

DOI: 10.1002/cpe

messages for objects associated with this region. The updates of the plane can be observed by any process joining the cluster. In particular, update messages will respect the causal order relationship. For instance, given a plane has landed at an airport, then events of passengers leaving the plane will be guaranteed to be delivered after the causally preceding landing event.

Processes which are eligible to propose updates for the objects of a cluster C are called *coordinators* of C. To become a coordinator of C in our cluster model, a process needs to obtain a ticket. Correspondingly, a cluster manages a maximum of n tickets according to the number of coordinators. The set of coordinators of C is denoted by $Core_C$. The set of coordinators is configurable and can change dynamically over time. Throughout the article, we will use the term *events* when referring to update messages sent or received by processes in a cluster.

The propagation of events in each cluster is achieved by multicast communication. It is not assumed that all processes of a cluster will receive an event which was multicast, nor does the multicast need to provide any ordering by itself. Any lightweight probabilistic group communication protocol in the literature [18–20] would be suitable. We refer to such protocols as *PrCast*. PrCast is assumed to provide the following properties:

- an event is delivered to all destinations with high probability; and
- decentralised and lightweight group membership, that is, a process can join and leave a multicast group in a decentralised way and processes do not need to know all the members of the group.

Within each cluster, we apply vector timestamps of the type used in [41]. Let the coordinator processes in $Core_C$ be assigned to unique identifiers in $\{1, \ldots, n\}$ (a process which is assigned to an identifier is also said to *own* this identifier). Then a time stamp t is a vector whose entry $t[j]$ corresponds to the $t[j]$th event send by a process that *owns* index j or a process that owned index j before (this is because processes may leave and new processes may join $Core_C$). A vector time stamp t_1 is said to be smaller than vector timestamp t_2 if $\forall i \in \{1, \ldots, n\}\ t_1[i] \leqslant t_2[i]$ and $\exists i \in \{1, \ldots, n\}$ such that $t_1[i] < t_2[i]$. In this case, we write $t_1 < t_2$.

For any multicast event e, we write t_e for the corresponding timestamp of e. Let e_1 and e_2 denote two multicast events in C, then e_1 causally precedes e_2 if $t_{e_1} < t_{e_2}$, whereas e_1 and e_2 are said to be concurrent if neither $t_{e_1} < t_{e_2}$ nor $t_{e_2} < t_{e_1}$. Further, we denote the index owned by the creator of event e as $index(e)$ and the event id of event e as $\langle index(e), t_e[index(e)]\rangle$.

Throughout the article, it is assumed that each process p maintains for each cluster C, a *cluster-consistency-tailored logical vector clock* (for brevity also referred to as a CCT-*vector clock*) denoted by $clock_p^C$. A CCT-vector clock is defined to consist of a vector time stamp and a sequence number. We write T_p^C when referring to the timestamp and seq_p^C when referring to sequence number of $clock_p^C$. T_p^C is the timestamp of the latest delivered event, whereas seq_p^C is the sequence number of the last multicast event performed by p. In Section 5, when describing the implementation of causal cluster consistency, we explain how these values are used. Note, whenever we look at a single cluster C at a time, we write for simplicity $clock_p$, T_p and seq_p instead of $clock_p^C$, T_p^C and seq_p^C, respectively.

In this article, we seek to address the following two main questions:

- How to manage for a cluster C, such that $Core_C$ can evolve dynamically and robust against process crashes as well as partitions of the network.
- How to design a generic dissemination mechanism in combination with the cluster management in order to provide efficiently the optimistic causal order relation.

4. DYNAMIC CLUSTER MANAGEMENT

In the following, we present a method that allows interleaved *cjoin* and *cleave* operations. The main idea of our approach is to make every process in the core of the cluster the coordinator of a subset of the tickets $\{0 \ldots n-1\}$. We will ensure that there are never two processes that simultaneously own or coordinate the same ticket. The basic idea is first discussed in Section 4.1, where we assume

DOI: 10.1002/cpe

that communication is reliable and processes do not fail. In Section 4.2, we show how to extend the presented ideas under a realistic failure model. Sections 4.3 and 4.4 analyse the algorithm with respect to correctness and liveness properties.

4.1. Basic algorithm

We assume that tickets form a cyclic relation according to their number; that is, the succeeding ticket to ticket i is ticket $i-1 \mod n$, whereas the preceding ticket to ticket i is ticket $i+1 \mod n$. Each process which becomes coordinator of the cluster will own one ticket. Let i be the ticket owned by process p, also denoted as ticket $(p) = i$. The successor of p is the closest process which can be reached by following the chain of succeeding tickets to i. Accordingly, the predecessor of p is the closest process which can be reached by following the chain of preceding tickets. Moreover, we denote q the dth closest successor (predecessor) of p if the process q is reachable in d steps from p by following the chain of successors (predecessors) starting at p.

In order to manage free tickets, the processes which own tickets also become the coordinators of a subset of all tickets maintained in a cluster. We define the set of tickets which is coordinated by a process in terms of successor and predecessor. Let p and q denote two processes owning tickets i and j, respectively and let q be the successor of p. Process p coordinates its own ticket i and all tickets succeeding its own ticket and preceding ticket j. Let the coordinated set, S_p, denote the set of tickets coordinated by p. Formally, we write

$$S_p = \{\, l \quad | \quad l = i - k \mod n,\ 0 \leqslant k < \min\{m \mid j = i - m \mod n,\ m > 0\}\}.$$

Figure 2 gives an example of how processes maintain and coordinate tickets, for example p_2 owns ticket 4 and coordinates the tickets $\{2, 3\}$.

Lemma 4.1
Let C denote a cluster with $Core_C \neq \emptyset$ and no two processes own the same tickets. Then

- for $p, q \in Core_C$ and $p \neq q \Rightarrow S_p \cap S_q = \emptyset$,
- $\cup_{p \in Core_C} S_p = \{0, \ldots, n-1\}$.

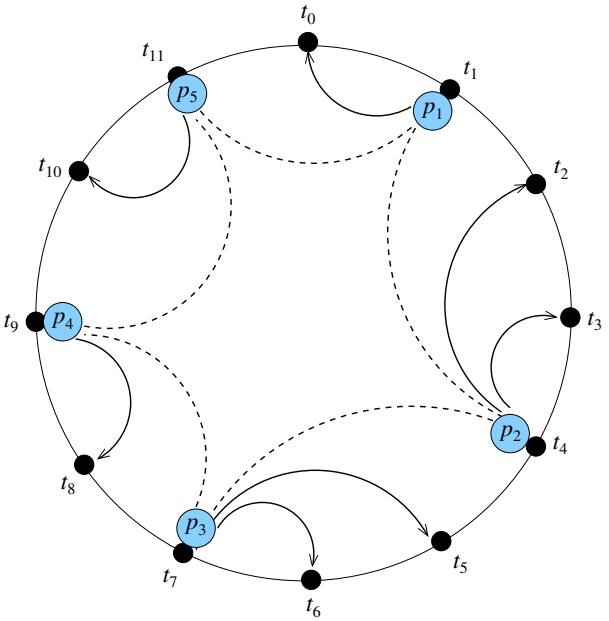


Figure 2. Illustration on how processes maintain and coordinate tickets of a cluster. An arrow from process p_i to a ticket indicates that p_i is the respective coordinator.

DOI: 10.1002/cpe

Proof
The lemma follows immediately from the definition of coordinated set by a process. □

Algorithm 1 presents a decentralised solution which can coordinate the tickets of a cluster if no failures occur. The algorithm ensures that no two processes coordinate the same tickets at the same time; the key to achieve this is by preserving the successor/predecessor relation between coordinators. A process p which wishes to become coordinator in the cluster selects an arbitrary coordinator. To enforce a good load balance of requests to coordinators, the selection by p could take the coordinator of a ticket chosen uniformly at random from the set of available tickets (this can be known by contacting any coordinator in the cluster). Let q be the selected coordinator and p send a *cjoin* message to q. Before responding to p's request, q will first serve all previous *cjoin* and *cleave* operations it received earlier by other processes. In this way, interleaving *cjoin* and *cleave* requests with respect to the same coordinator become serialised. If q has decided to perform a *cleave* operation or does not have any available tickets, it will reply negatively to p. If q is ready to serve the *cjoin* request by p, it will assign a ticket $t \in S_q$ to p (possibly reflecting the random choice when determining q as a suitable coordinator). Let r be q's successor. Process q will send a message *ACKCJOIN* to p with information about t and r to p and will select p as its new successor.

When p receives the message *ACKCJOIN*, p will select q as its predecessor and r as its new successor. In order to allow process r to leave the cluster and maintain its predecessor information correctly, p must, before being able to perform as a coordinator, send a message *NEWSUCC* to process r. If r is not intending to leave the cluster, it will reply by sending an acknowledgement *ACKSUCC* to p and update its predecessor to be p. Process p can then perform as a coordinator of the cluster.

In the case a process r intends to leave the cluster, it first processes all previously received cjoin and cleave requests and sends afterwards a *CLEAVE* message including information of the successor of r, say s, to its predecessor, say q. If r receives afterwards from another process p, a message *NEWSUCC*, it will again send a message *CLEAVE* to p. Process r only leaves the cluster after it has received a message *ACKCLEAVE*.

A process p serves a *cleave* message by r only if r is the current successor of p. In this case, p will send a message *ACKCLEAVE* to r. Thereafter, p sets s as its new successor and sends a message *NEWSUCC* to s. Note that p may have to subsequently serve *CLEAVE* messages from its new successor until finally receiving a message *ACKSUCC* from a successor. However, after each *ACKCLEAVE*, a process coordinates a larger amount of tickets and hence, the number of subsequent *NEWSUCC* messages before a process can perform as a coordinator is bounded.

Once a process may perform as a coordinator, it also PrCasts that it became a coordinator in $Core_C$ and that it owns ticket t. Note that the PrCast operation is only of relevance to inform other processes about p being a coordinator, but it is not necessary to prevent any pair of distinct processes from maintaining the same ticket.

In order to verify correctness of the protocol as stated in Theorem 4.1, recall that according to Lemma 4.1 correctly preserving the relation among successors and predecessors, suffices to guarantee unique assignment of processes to tickets. This is shown in Lemma 4.2.

Lemma 4.2
Let q be a coordinator in $Core_C$ with successor r, serving a *cjoin* operation of p. Then

- any interleaving *cjoin* operation will take effect earliest after processes p and q successfully updated their successor and predecessor,
- an interleaving *cleave* operation of r will successfully be managed at p and therefore preserve the predecessor successor relation of $Core_C$ correctly.

Theorem 4.1
Let $\Sigma := \sigma_1, \ldots, \sigma_m$ denote a sequence of potentially interleaved operations on a cluster C where σ_i corresponds to a *cleave* or *cjoin* operation. If Σ maintains $Core_C$ to include at least one process, the algorithm guarantees for any $p, q \in Core_C$

DOI: 10.1002/cpe

Algorithm 1 Cluster management in the absence of failures

VAR

Cview_p:	vector of processes
ImmedSucc_p:	immediate successor of process p
ImmedPred_p:	immediate predecessor of process p
state_p:	state variable
ticket_p:	the ticket owned by process p

Message types:

CJOIN, CLEAVE, ACKJOIN, ACKSUCC, ACKCLEAVE, REJECT

Init$_p$:

$\text{ticket}_p := \perp$
$\text{state}_p :=$ joining
Send $\langle$CJOIN, $p\rangle$ to a known coordinator in $Core_C$.

Initialisation of variables when cjoin accepted

On *process* p *receives* $\langle ACKCJOIN, i, j, Cview\rangle$ *from process* q
$\text{Cview}_p := \text{Cview}$
p becomes the coordinator for all tickets from i down to $j+1$
$\text{state}_p :=$ coordinator
$\text{ticket}_p := i$
$\text{ImmedSucc}_p := \text{Cview}[j]$
$\text{ImmedPred}_p := q$
Send $\langle$NEWSUCC$\rangle$ to $\text{Cview}[j]$

Successor acknowledged

On *process* p *receives* $\langle ACKSUCC\rangle$ *from process* q
$\text{ImmedSucc}_p := q$

Receiving a cjoin request

On *process* p *being coordinator of tickets* i *down to* $j+1$ *receives* $\langle CJOIN\rangle$ *from process* q
if $\text{state}_p \neq$ coordinator **then**
  Send $\langle$REJECT$\rangle$ to q
else
  Process all previously received CJOIN and CLEAVE requests
  if $|S_p| > 1$ **then**
    Select ticket $t \in S_p \setminus \{i\}$.
    $\text{Cview}[t] := q$
    $\text{ImmedSucc}_p := q$
    Send $\langle$ACKCJOIN, t, j, Cview$\rangle$ to q
  else
    Send $\langle$REJECT$\rangle$ to q
  end if
end if

A new predecessor

On *process* p *being coordinator of tickets* i *down to* $j+1$ *receives* $\langle NEWSUCC\rangle$ *from* q
if $\text{state}_p =$ leaving **then**
  Send $\langle$CLEAVE, $\text{Cview}[j]\rangle$ to q
else
  Send $\langle$ACKSUCC$\rangle$ to q
  $\text{ImmedPred}_p := q$
end if

Leaving the cluster

On *process* p *being coordinator of tickets* i *down to* $j+1$ *decides to leave the cluster*
$\text{state}_p :=$ leaving
Serve all previously received cjoin and cleave requests
Send $\langle$CLEAVE, $\text{Cview}[j]\rangle$ to ImmedPred_p

Receiving a cleave request

On *process* p *being coordinator of tickets* i *down to* $j+1$ *receives* $\langle CLEAVE, r\rangle$ *from* q
if $q = \text{ImmedSucc}_p$ and p is not serving any cjoin and $\text{state}_p \neq$ leaving **then**
  Send $\langle$ACKCLEAVE, $q\rangle$
  Send $\langle$NEWSUCC, $r\rangle$
  $\text{ImmedSucc}_p := r$
end if

Receiving a cleave acknowledgement

On *process* p *being coordinator of tickets* i *down to* $j+1$ *receives* $\langle ACKLEAVE, p\rangle$ *from* q
if $\text{state}_p =$ leaving **then**
  $\text{state}_p :=$ not_a_coordinator
  $\text{ticket}_p := \perp$
end if

DOI: 10.1002/cpe

- unless $p = q$, $S_p \cap S_q = \emptyset$;
- unless $p = q$, p and q maintain different tickets.

4.2. *Supporting link and process failures*

In the following, we present an algorithm which extends the previous framework of Section 4 to deal with link and process failures. It is assumed that processes fail by stopping, we do not consider Byzantine faults. Links may be slow or failing. Communication between pairs of processes is connection oriented. Let δ denote the maximum tolerated message delay and let p and q denote processes. Connection oriented communication guarantees: if p sends a message, M, to q, p expects to receive a status about M not later than time δ. If status of M is *OK*, then q has received M not later than time δ. Otherwise p has no knowledge whether q received the message or not; we say then that p *weakly detects* q *as faulty*. Because the algorithm works in rounds, we also assume that processes have clocks which maintain approximately the same speed. Let T denote a time period larger than the maximum tolerated message delay. If m processes periodically with period T send messages to p , then p will receive $m - \epsilon < m' < m + \epsilon$ messages during any time interval of length T which starts after p has received the messages sent in the previous period by the m sources, when none of the m processes failed.

Algorithm 2 Decentralised and fault tolerant cluster management

VAR

L_p:	set consisting of $2k+1$ predecessors p received from its immediate predecessor
R_p:	set consisting of p and $2k$ predecessors successfully sent to its immediate successor
ALIVE_p:	set of processes which p received an *ALIVE* message from during a round
Cview_p:	vector of processes
ImmedSucc_p:	immediate successor of p
ImmedPred_p:	immediate predecessor of p
TempRounds_p:	indicates the number of rounds for which a process is not sending *UPDATE* messages
state_p:	state variable
ticket_p:	the ticket owned by process p
P_{exclude}:	probability to start exclusion algorithm after weakly detecting a faulty successor

Message types:

CJOIN, ALIVE, UPDATE, ACKJOIN, EXCLUDE, REQCOORD, ACKEXCLUDE

Init$_p$:

$\text{ticket}_p := \perp$
$\text{state}_p :=$ joining
Send $\langle$CJOIN, $p\rangle$ to a known coordinator in $Core_C$.

Main Loop of the coordinator algorithm:

Do in every round (duration longer than PrCast) while $\text{state}_p =$ coordinator
  if $|\text{ALIVE}_p \cap L_p| < k+1$ **then**
    $\text{state}_p :=$ disconnected
    $\text{ticket}_p := \perp$
    exit loop
  end if
  Send $\langle$ALIVE, $p\rangle$ to $2k+1$ closest successors in *Cview*.
  if $\text{TempRounds}_p = 0$ **then**
    $R := \{r \in L_p \mid r$ is among the $2k$ closest predecessors of $p\} \cup p$
    STATUS := Send $\langle$UPDATE, $R\rangle$ to ImmedSucc_p
    if STATUS is OK **then**
      $R_p := R$
    else
      Run exclusion algorithm with probability P_{exclude}
    end if
  else
    $\text{TempRounds}_p := \text{TempRounds}_p - 1$
  end if

DOI: 10.1002/cpe

Algorithm 3 Handling of messages

Initialisation of variables when cjoin succeeds

On *process* p *receives* $\langle ACKCJOIN, L, i, j, Cview\rangle$ *from* q
  $\text{Cview}_p := \text{Cview}$
  $L_p := L$
  $R_p := \emptyset$
  process p becomes the coordinator for all tickets i down to $j+1$
  $\text{ticket}_p := i$
  $\text{ImmedSucc}_p := \text{Cview}[j]$
  $\text{ImmedPred}_p := q$
  $\text{TempRounds}_p := 0$
  Send $\langle$ALIVE, $p\rangle$ to $2k+1$ closest successors in Cview_p.

Handling of *UPDATE* messages

On process p receiving $\langle$UPDATE, $R\rangle$
  $L_p := R$

Receiving a cjoin request

On *process* p *being coordinator of tickets* i *down to* $j+1$ *receives* $\langle CJOIN\rangle$ *from* q
  if $(|S_p| > 1) \wedge (\text{TempRounds}_p = 0)$ **then**
    Select ticket $t \in S_p$.
    $\text{Cview}[t] := q$
    $\text{ImmedSucc}_p := q$
    $R := \{r \in L_p \mid r$ is among the $2k$ closest predecessors of $p\} \cup p$
    STATUS := Send $\langle$ACKCJOIN, $R, t, j, \text{Cview}\rangle$
    if STATUS is OK **then**
      $R_p := R$
    else
      Run exclusion algorithm with probability P_{exclude}
    end if
  else
    Send $\langle$REJECT$\rangle$ to q
  end if

The algorithm performs in rounds, where the time between two consecutive rounds is assumed to be long enough to host a PrCast; that is, to inform members of the cluster C about a successful *cjoin* operation (if any has happened). The fault tolerance of the algorithm is controlled by the parameter k. In a round of the algorithm, a process can tolerate in its $2k+1$ neighbourhood up to k process or communication failures. The algorithm is described in pseudocode (cf. Algorithms 2, 3 and 4) and below, we present the ideas informally. During a round, the algorithm maintains the following two invariants:

- Any nonfaulty process p in $Core_C$ which does not perform a *cleave* operation remains in $Core_C$ as long as p knows that at least $k+1$ of its $2k+1$ closest predecessors have not experienced any process or link failures.
- Failed processes will eventually be excluded from $Core_C$ and processes which perform *cjoin* subsequently may re-use the respective tickets.

The first invariant is achieved by the processes in $Core_C$ sending *ALIVE* messages to their $2k+1$ closest successors in each round. A process that receives less than $k+1$ *ALIVE* messages during a round thinks that it is considered as failed and immediately stops being a coordinator and leaves $Core_C$ (cf. Algorithm 2 and Figure 3(I)–(II)).

In order to manage the exclusion scheme (cf. Algorithms 3, 4 and Figure 3(III)), a process p maintains two sets denoted by L_p and R_p. The set L_p is used to store p's 'knowledge' of its $2k+1$ predecessors (this information is received from its immediate predecessor), whereas R_p contains the information in p's last successful *UPDATE* message to p's immediate successor containing the $2k$ closest predecessors of p and p itself. Both sets are needed to determine whether a range of coordinators can be excluded. When p joins $Core_C$, L_p is initialised by the coordinator performing the *cjoin* operation for p. The set R_p is initially empty. Each process also maintains an array denoted by $Cview_p$ which is p's local view on the set of coordinators $Core_C$; that is, if $Cview_p[i] = q$ holds, then p assumes q to be the coordinator owning ticket i.

DOI: 10.1002/cpe

Algorithm 4 Exclusion Algorithm

Do
  STATUS := FALSE
  while ($p \neq$ succ(ImmedSucc)) $\wedge$ (STATUS is FALSE) **do**
    ImmedSucc := succ(ImmedSucc) {*Finds the next possible successor from* Cview}
    STATUS := Send$\langle$EXCLUDE, $p\rangle$ to ImmedSucc
  end while
  if (STATUS is True) $\wedge$ (p receives $\langle$ACKEXCLUDE, $L_q\rangle$ from q) **then**
    E_{pq} := all tickets succeeding ticket(p) and preceding $ticket(q)$
    Send $\langle$REQCOORD, $E_{pq}\rangle$ to all processes in $L_q \cap R_p$
    Wait for time 2δ for replies of type *ACKCOORD*
    if p receives $\geqslant k+1$ replies of type *ACKCOORD* **then**
      {Do not send *UPDATE* messages while some excluded processes may still be alive}
      TempRounds$_p$:= $dist(p,q)-1$
    else
      state$_p$:= disconnected
      ticket$_p$:= $\perp$
      exit loop
    end if
  else
    state$_p$:= disconnected
    ticket$_p$:= $\perp$
    exit loop
  end if

On q receives $\langle$EXCLUDE, $p\rangle$
  Reply$\langle$ACKEXCLUDE, $L_q\rangle$

On r receives $<$ REQCOORD, $E_{pq}>$
  if $r \notin E_{pq}$ **then**
    Send $\langle$ACKCOORD$\rangle$ to p
    Remove processes in E_{pq} from *Cview*
  end if

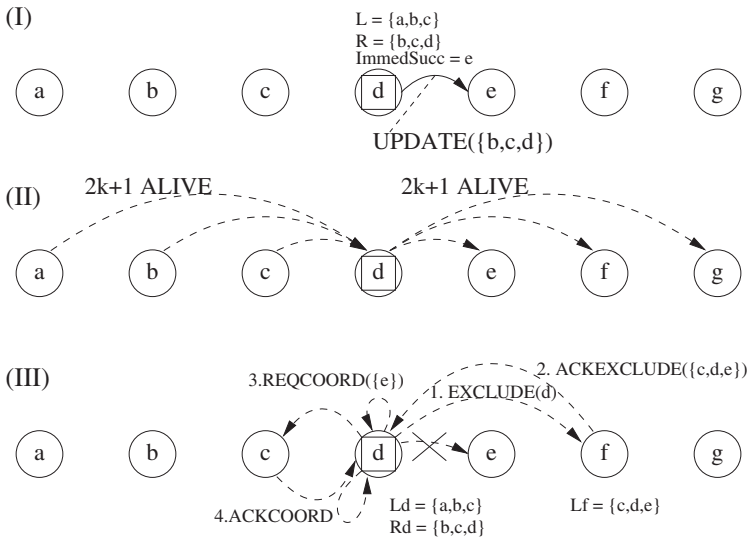


Figure 3. Example of the fault-tolerant cluster management algorithm with $k = 1$ focusing on the process d. (I) and (II) UPDATE messages and ALIVE messages under normal operation. (III) Message exchanges during an exclusion of d's immediate successor e.

In each round, p proceeds if it has received, during a round, at least $k+1$ *ALIVE* messages from processes in L_p, otherwise p thinks that it is considered as failed (cf. below for this case). If p also successfully received an *UPDATE* message from its direct predecessor proposing a new set L'_p, which includes $2k+1$ predecessors of p, then p sets $L_p = L'_p$.

DOI: 10.1002/cpe

If p may proceed, it creates $2k + 1$ *ALIVE* messages and sends them to the $2k + 1$ closest successors known from $Cview_p$. Moreover, it sends an *UPDATE* message to its direct successor containing a set denoted R'_p. The set R'_p contains the $2k$ closest predecessors in L_p and p itself. If p's transmission of the message *UPDATE*(R'_p) to its direct successor is successful, then p will set $R_p = R'_p$.

Assume a process weakly detects its successor r to be faulty, for instance, because it could not establish a connection to r for some time. In order to release the tickets owned and coordinated by r, which is potentially faulty, p will try to contact the next closest successor in $Cview_p$ reachable; that is, not detected as weakly faulty. Let q be the next closest successor reachable by p then q will reply by sending L_q. Process p will request from all processes in $R_p \cap L_q$ to be the new coordinator of all tickets succeeding p and preceding q denoted by E_{pq}. Only if p receives $k + 1$ acknowledgement messages from destinations in $R_p \cap L_q$, p becomes the *temporary coordinator* of the tickets, otherwise p thinks it is considered as failed.

Although being a temporary coordinator, p behaves like an ordinary coordinator; however, it does not attempt to change L_q by sending an *UPDATE* message and it does not serve *cjoin* requests. All processes in E_{pq} which neither have failed nor think they are considered to have failed are said to be *alive*. Once there does not exist any alive processes in E_{pq}, p behaves like an ordinary coordinator again. Note that the time for a process remaining a temporary coordinator is bounded by at most the distance between p's and q's tickets because in every round, the closest alive process in E_{pq} is guaranteed to think that it is considered to have failed at the end of the round.

Processes which are requested to acknowledge an exclusion interval E_{pq} only acknowledge if their ticket is not contained in E_{pq}. Processes which acknowledged the exclusion of a process will remove processes in E_{pq} from *Cview* and prevent any updates of tickets corresponding to E_{pq} for $\text{dist}(p, q)$ rounds.

4.3. Correctness of the cluster management method

In order to prove correctness of the membership algorithm of Section 4.2, we need to show that even in the occurrence of failures, (i) two processes will never create conflicting events; and (ii) the algorithm invariants are maintained.

In Lemma 4.3, we first consider the behaviour of the algorithm when no failures occur.

Lemma 4.3
Let neither process failures, link failures, or slow links occur and processes always receive sufficiently many *ALIVE* messages. For any sequence of interleaving *cjoin* operations, the membership scheme is equivalent to the membership protocol of Section 4.

Proof
Both algorithms show only different behaviour if p executing Algorithm 2 weakly detects its immediate successor r to be faulty. Because neither processes, nor links do fail p must have detected r as faulty because r thinks it has been considered to be failed. This implies that r did not receive sufficiently many *ALIVE* messages or decided to leave the cluster, which is a contradiction to our assumption. □

The critical case to analyse is after process p initiated the exclusion of E_{pq}. Lemma 4.4 states that during a round, the closest successor in E_{pq} will fail.

Lemma 4.4
Let E_{pq} denote the set of processes to be excluded, where p coordinates the exclusion and q is the new successor of p. Further, let A denote the set of processes which received sufficiently many *ALIVE* messages in the current round. Let r denote the closest process in E_{pq} which is still alive. Then

$$A \cap (L_r - E_{pq} - R_p) = \emptyset.$$

DOI: 10.1002/cpe

Proof
We can associate the passing of an *UPDATE* message with a token. We say process q received a token from p if there is a chain of consecutive *UPDATE* messages originating in p and ending in q. We define a relation $\prec$, where $p \prec q$ if q has received a token from p when it was created (i.e. the time it performed the *cjoin* operation), whereas $p \not\prec q$ if q did not receive a token from p at the time it was created.

Consider case $p \prec r$: In this case, $L_r - E_{pq} - R_p$ is either empty or it contains destinations which where in a previous *Cview* of p. However, when p successfully updated R_p, the respective destinations were guaranteed to be excluded by the predecessors of p. Hence, this case yields

$$A \cap (L_r - E_{pq} - R_p) = \emptyset$$

Let $p \not\prec r$: Any token originated by p and received by q must have been received by r. In particular, if *Cview* of q was influenced by p, also r must have received influence by p. Then we can reason the same as before.

The difficult case remains, where q did not receive any influence from p. We define for two processes p' and q', p' to be the parent of q' if p' coordinated q' to enter the cluster. Further, we define ancestor by the transitive closure of the parent relation. If q did not receive any token from p, but share a common influence, then q must have received a token from an ancestor of p. Let s denote the ancestor of p which succeeded last in sending a token to q.

Case r received the respective token: If r received the respective token, then it shares the same influence as q. Every consecutive token which origins from set E_{pq} has no impact on $A \cap (L_r - E_{pq} - R_p)$. However, every token originating outside E_{pq} by transitivity will affect L_p once p has joined the cluster. Hence, no vertices in $L_r - E_{pq} - R_p$ are alive after p determined its set R_p.
Case r did not receive the respective token: There must be an ancestor which received the respective token. If there was not, we would conclude $E_{pq} = \emptyset$. Then again p on its creation would share all influence by s on the ancestor of r and by transitivity to r itself. Hence, again, all tokens which did not influence p originate from the set E_{pq}. Therefore, no processes in $L_r - E_{pq} - R_p$ are alive, once p has updated R_p.

□

Lemma 4.4 immediately implies Corollary 4.1 which states how long a process p needs to be temporary coordinator until at least i alive processes in E_{pq} have failed.

Corollary 4.1
The ith successor of p in E_{pq} will fail the latest i rounds after p was acknowledged.

Proof
The immediate successor r of p clearly fails because all *ALIVE* messages r can expect according to Lemma 4.4 are from processes inside R_p (suppose p maintains a copy of send L_p) and at most, k messages did not acknowledge p. Assume now that until round $i-1$, the closest $i-1$ successors have failed. Then in round i, the only candidates for sending *ALIVE* messages are in L. However, there are at most k candidates which did not acknowledge the exclusion of the ith successor. □

Theorem 4.2
Algorithm 2 guarantees that two processes never have common tickets; they either own or coordinate.

Proof
Lemma 4.3 shows that only exclusion could cause any such conflicts. Assume that during an execution, two alive processes r and s are processes coordinating common tickets. This implies that one process, say r, failed to be excluded, whereas s was inserted. Let p be the process which failed to exclude r and inserted s.

DOI: 10.1002/cpe

After p initiated the exclusion of E_{pq} with $r, s \in E_{pq}$, p switches state to become temporary coordinator for dist(p, q) rounds. During this time, p could not have inserted s. However, when p switches state to become active coordinator and inserts s, Corollary 4.1 guarantees that r by that time thinks it is considered to have failed, contradicting that both r and s were active. □

4.4. Performance and liveness properties of the cluster management method

Message overhead. Note that the duration of a round is assumed to be longer than the time of a PrCast. PrCast is used to inform all processes which joined a cluster about an event regarding the resources of the cluster. The overhead which is induced by the membership protocol corresponds to the number of sent *ALIVE* messages. In each round, a process sends and receives at most $2k + 1$ messages. Hence, the cluster management protocol can be considered as lightweight; that is, it only adds a low number of additional messages while performing in combination with an application using the cluster management protocol. In addition, every successful ticket acquisition is followed by a PrCast which involves all processes which joined the cluster.

Availability. An interesting performance measure is how well the algorithm manages to grant new processes access to tickets in the occurrence of failures and dependent on the amount of tickets maintained by nonfaulty processes. Let α denote the fraction of tickets taken by nonfaulty processes. Moreover, let p_f denote the probability for a process to fail in a round. Whenever a process q fails, the predecessor, say p, is trying to reclaim the tickets maintained by q. Although running the exclusion algorithm, p performs as a temporary coordinator and does not release any tickets.

Observe that $Core_C$ consists of the processes which have not been excluded and processes which perform correctly, that is, we know $|Core_C| \geqslant \alpha n$. Because there exists at most n tickets, the expected number of tickets maintained by each coordinator of $Core_C$ is smaller or equal to $1/\alpha$. Hence, the time to reclaim tickets from a failing process is expected to take time less or equal to $1/\alpha$.

Assume that (i) α remains constant and (ii) the exclusion algorithm needs $1/\alpha$ rounds. Then the expected number of failing processes which needs to be excluded is $p_f n$ because in each round, $\alpha p_f n$ processes are expected to fail. By applying the Chernoff bound [42], one can bound the probability that in a round of the algorithm's execution, there exist more than $2p_f n$ faulty processes. The probability is strictly smaller than $(e/4)^{2p_f n}$. That means a process which attempts to acquire a ticket succeeds w.h.p. if $p_f < \frac{1}{2}(1 - \alpha)$.

5. LAYERED ARCHITECTURE FOR OPTIMISTIC CAUSAL DELIVERY

This section proposes a layered protocol for achieving optimistic causal delivery. Here, we assume that the set of coordinators of a cluster is maintained with the fault-tolerant configurable cluster management method described earlier. Using this method, each coordinator is assigned to a unique entry in the vector clock, and furthermore, all coordinators know each other via the cluster management protocol.

5.1. Method for clustered causal delivery

The first of the two layers uses *PrCast* in order to multicast events inside the cluster (cf. pseudo-code description Algorithm 5). The second layer, the causality layer, implements the optimistic causal delivery service. The causal delivery protocol is inspired by the protocol by Ahamad *et al.* [41] and

DOI: 10.1002/cpe

is adapted and enhanced to provide the optimistic delivery service of the causal cluster consistency model and the recovery procedure for events that may be missed because of *PrCast*.

Each process in a cluster interested in observing events in optimistic causal order (which is always true for a coordinator), maintains a queue of events denoted by H_p^C. For any arriving event e, one can determine from T_p^C and the event's timestamp t_e whether there exist any events which (i) causally precede e; (ii) have not been delivered; and (iii) could still be deliverable according to the optimistic causal order property. More precisely, we define this set of not yet delivered deliverable events as

$$to\_deliver\_before(e) = \{e' \mid t_{e'} < t_e \wedge \neg(t_{e'} < T_p^C)\}$$

and their event ids, which can be used for recovery, can be calculated as follows

$$\begin{aligned} to\_deliver\_before\_ids(e) = \{\langle i, j\rangle \quad | \quad & (\forall i \neq index(e)\ .\ T_p^C[i] < j \leqslant t_e[i]) \\ & \vee\ (i = index(e) \wedge T_p^C[i] < j < t_e[i])\}. \end{aligned}$$

If there exist any such events, e will be enqueued in H_p^C until those events have been delivered or e is about to become obsolete at which point e will be delivered. (Prior to that process, p may 'pull' missing events.) Otherwise, p delivers e to the application.

When a process p delivers an event e referring to cluster C, the CCT-vector clock $clock_p^C$ is updated by setting $\forall i\ T_p^C[i] = \max(t_e[i], T_p^C[i])$. Process p also checks whether any events in H_p or recovered events now can be dequeued and delivered. Before a coordinator p in $Core_C$, owning the jth vector entry, multicasts an event it updates $clock_p^C$ by incrementing seq_p^C by one. The event is then stamped with a vector timestamp t such that $t[i] = T_C^p[i]$ for $i \neq j$ and $t[j] = seq_p^C$.

Although PrCast delivers events with high probability, a process may need to recover some events. The recovery procedure, which is invoked when an event e in H_p is close to become obsolete, sends recovery messages for the missing events that precede e. The time before e becomes obsolete depends the amount of time because the start of the dissemination of e is assumed to be larger than the duration of a PrCast (which is estimated by the number of hops that an event needs to reach all destinations with w.h.p.) and the time it takes to send a recovery message and receive an acknowledgement. At the time $e \in H_p$ becomes obsolete, p delivers all recovered events and events in H_p that causally precede e and e in their causal order. A simple recovery method is to contact the sender of the missing event. For this purpose, the sender has a *recovery buffer* which stores events until no more recovery messages are expected (this is e.g. the case if $\forall i\ t_e[i] < T_p^C[i]$). In the following, we will present and analyse another recovery method that enhances the throughput and the fault tolerance.

5.2. *Properties of the basic protocol and fault-tolerant extensions*

The PrCast protocol provides a delivery service that guarantees that an event will reach all its destinations with high probability; that is, PrCast can achieve high message stability. When an event needs recovery, the number of processes that did not receive the event is expected to be low. Thus, a process multicasting an event is expected to receive a low number of recovery messages.

First, consider the case when there are no failures in the system. If there are no process, link or timing failures, reliable point to point communication succeeds in recovering all missing events, and thus provides causal order without any message loss. The following lemma is straightforward, following the analysis in [41].

Lemma 5.1
An execution of the two-layer protocol guarantees causal delivery of all events disseminated to a cluster if neither processes nor links are slow or fail.

When there are timing or other failures, the method can deal with them in the following way, which not only guarantees fault tolerance but also influences the throughput in a positive way. To introduce redundancy in the recovery protocol, all processes are required to keep a history of some of the observed events, so that a process only needs to contact a fixed number of other processes to

DOI: 10.1002/cpe

Algorithm 5 Two-Layer protocol

VAR
- H_p: set of received events that can not be delivered yet
- R: set of recovered events that can not be delivered yet
- B: fixed size recovery buffer with FIFO replacement.
- seq_p^C: sequence number of the last event created by a process owning the identifier p in $Core_C$.
- T_p^C: vector timestamp indicating the causal present for p.

On process p in $Core_C$ creates the event e
- $seq_p^C := seq_p^C + 1$; $t_e := T_p^C$; $t_e[p] := seq_p^C$ /* Create timestamp t_e */
- PrCast($\langle e, t_e \rangle$)
- Insert e into recovery buffer B

On process p receives $\langle e, t_e \rangle$
- Insert e into recovery buffer B
- **if** e can be delivered **then**
  - deliver(e)
  - for all $e' \in H_p \cup R$ that can be delivered
    - deliver(e')
- **else**
  - **if** e is not delivered or obsolete **then**
    - delay(e, time_to_terminate)
  - **end if**
- **end if**

On timeout(e, time_to_terminate)
- for all $eid \in to\_deliver\_before\_ids(e)$ not in $H_p \cup R$ and eid not already under recovery
  - send($\langle RECOVER$, eid$\rangle$) to source(eid) or to k arbitrary processes in cluster
- delay(e, time_to_recover)

On timeout(e, time_to_recover)
- for all $e' \in to\_deliver\_before(e) \cap (H_p \cup R)$ that can be delivered
  - deliver(e')
- deliver(e)
- for all $e' \in H_p$ that can be delivered
  - deliver(e')

On process p receives $\langle RECOVER$, eid$\rangle$ from process q
- **if** p has e with identifier eid in its buffer **then**
  - respond($\langle ACKRECOVER, e, t_e \rangle$) to process q
- **end if**

On process p receives $\langle ACKRECOVER, e, t_e \rangle$
- Insert e into recovery buffer
- **if** e can be delivered **then**
  - deliver(e)
  - for all $e' \in R \cup H_p$ that can be delivered
    - deliver(e')
- **else**
  - **if** e is not delivered or obsolete **then**
    - $R := R \cup \{e\}$
  - **end if**
- **end if**

On deliver(e)
- $\forall i$ $T_p^C[i] := \max(t_e[i], T_p^C[i])$ /* Update T_p^C */
- Remove e from R and H_p
- Deliver e to the application

recover an event. Further, such redundancy could help the recovery of a failed process. As it is desirable to bound the size of this buffer, we analyse the recovery buffer size and number of processes to contact such that the recovery succeeds with high probability.

Following [20], we describe a model suitable to determine the probability for availability of events that are deliverable and may need recovery in an arbitrary system consisting of a cluster C of n processes that communicate using the two-layer protocol. Let $\mathcal{C}$ denote this system, and T denote the time determined by the number of rounds an event stays at most in $\mathcal{C}$. Note the similarity of the buffer system to a single-server queueing system, where new events are admitted to the queue as a random process. However, unlike common queueing systems, the service time (time needed for all processes in C to get the event using the layered protocol) in this model depends on the arrival times of events. The service time is such that every event stays in the queue at least as long as it needs to stay in the buffer of $\mathcal{C}$ in order to guarantee delivery/recovery (i.e. whether the queue is stable is not

DOI: 10.1002/cpe

an issue here). In the following, we estimate the probability that the length of the queue exceeds the choice of the length for the recovery buffer of $\mathcal{C}$.

If a_i denotes the arrival time of an event e_i, the 'server' processes each event at time $s_i = a_i + T$. Observe that if the length of the buffer in $\mathcal{C}$ is greater than the maximum length of the queue within the time interval $[a_i, s_i]$, then $\mathcal{C}$ can safely deliver e_i.

Consider $[t_a, t_s]$ denoting an interval of length T and the random variable $X_{i,j}$ denoting the event that at time $t_a + i$, process j inserts a new event in the system. Further, assume that all $X_{i,j}$ occur independently, and that $\mathbf{Pr}[X_{i,j} = 1] = p$ and $\mathbf{Pr}[X_{i,j} = 0] = 1 - p$. The number of admitted events in the system can be represented by the random variable $X := \sum_{j=1}^{n} \sum_{i=1}^{T} X_{i,j}$; hence, the random process describing the arrival rate of new events is a binomial distribution and the expected number of events in the queue in an arbitrary time interval $[t_a, t_s]$ equals $\mathbf{E}[X] = npT$. Clearly, the length of the recovery buffer must be at least as large as $\mathbf{E}[X]$, or we are expected to encounter a large number of events that cannot be recovered.

Now, using the Chernoff bound [20, 42], we bound the buffer size so that the probability of an event that needs recovery not to be present in the recovery buffer of any arbitrary process becomes low.

Theorem 5.1
Let e be an event admitted to a system $\mathcal{C}$ executing the two-layered protocol, where each event is required to stay in $\mathcal{C}$ for T rounds. Each of the n processes in the system admits a new event to $\mathcal{C}$ in a round with probability p. Then $\mathcal{C}$ can guarantee the availability of e in the recovery buffer of an arbitrary process with probability strictly greater than $1 - \left(\frac{e}{4}\right)^{npT}$ if the size of the buffer is chosen greater than or equal to $2npT$.

Proof
Following the Chernoff bound for binomial distributions, for any $\delta > 0$ it is the case that $\mathbf{Pr}[X > (1+\delta)npT] < \left(\frac{e^{\delta}}{(1+\delta)^{\delta+1}}\right)^{npT}$. By choosing $\delta = 1$, the result follows. □

To estimate T, we can use the estimated duration of a PrCast, for example as in [20]. Let PrCastTime denote this time. An event e is likely to be needed in $\mathcal{C}$ for (i) PrCastTime rounds (to be delivered to all processes with high probability); (ii) plus PrCastTime rounds, if missed, to be detected as missing by the reception of a causally related event (note that this is relevant under high load, because in low loads PrCast algorithms are even more reliable); (iii) plus the time $time\_to\_terminate + time\_to\_recover$ spent before and after requesting recovery.

Further, because processes may fail, a process that needs to recover some event(s) should contact a number of other processes to guarantee recovery with high probability. Assume that processes fail independently with probability p_f, and let X_f be the random variable denoting the number of faulty processes in the system. Then $\mathbf{E}[X_f] = p_f n$. By applying the Chernoff bound as in Theorem 5.1, we get:

Lemma 5.2
If, in a system of n processes where each one may fail independently with probability p_f, we consider an arbitrary process subset of size greater than or equal to $2np_f$, with probability strictly greater than $1 - \left(\frac{e}{4}\right)^{np_f}$, there will be at least one nonfailed process in the subset.

Proof
Following the Chernoff bound for binomial distributions, for any $\delta > 0$, it is the case that $\mathbf{Pr}[X > (1+\delta)np_f] < \left(\frac{e^{\delta}}{(1+\delta)^{\delta+1}}\right)^{np_f}$. By choosing $\delta = 1$, the result follows. □

This implies that if a process requests recovery from $R = 2p_f n$ processes, then w.h.p., there will be at least one nonfaulty to reply.

DOI: 10.1002/cpe

Theorem 5.2
In a system of n processes, where each one may fail independently with probability $p_f \leqslant k/(2n)$ for fixed k, an arbitrary process that needs to recover events according to the two-layer protocol will get a reply with high probability if it requests recovery from k processes.

Proof
From Lemma 5.2, we know that if a process requests recovery from at least $2p_f n$ processes, then w.h.p., there will be at least one nonfaulty process among them, which can answer. So we choose $k \geqslant 2p_f n$, thereby for a given k high probability guarantee holds for $p_f \leqslant k/(2n)$. □

Note that requesting recovery only once and not propagating the recovery messages is good because in cases of high loss because of networking problems, we do not flood the network with recovery messages. Compared with recovery by asking the originator of an event, this method may need k times more recovery messages. However, the advantages are tolerance of failures and process departures, as well as distributing the load of the recovery in the system.

Regarding replacement of events in the recovery buffer, the simplest option is first-in first-out replacement. Another option is an aging scheme, for example, based on the number of hops the event has made. As shown in [20], an aging scheme may improve performance from the reliability point of view. However, to employ such a scheme here, we need to sacrifice the separation between the consistency layer and the underlying dissemination layer to access this information. Instead, note that using a dissemination algorithm such as the *Estimated-Time-To-Terminate-Balls-and-Bins*(ETTB)-gossip algorithm [20] that uses an aging method to remove events from process buffers and guarantees very good message stability implies that the reliability is improved because fewer processes may need to recover events.

6. EXPERIMENTAL EVALUATION OF THE TWO-LAYER PROTOCOL

In this section, we investigate the scalability of causal cluster consistency and the reliability and throughput effects of the optimistic causality layer in the two-layer protocol. We refer to a message/event as lost if it was not received or could not be delivered without violating optimistic causal order.

6.1. System and implementation

The evaluation of the two-layer protocol was done on 125 networked computers at Chalmers University of Technology. The computers were Sun Ultra 10 and Blade workstations running Solaris 9 and PC's running Linux distributed over a few different subnetworks of the university network. The average round-trip-time for a 4KB IP-ping message was between 1ms and 5ms. As we did not have exclusive access to the computers and the network, other users might potentially have made intensive use of the network concurrently with the experiments.

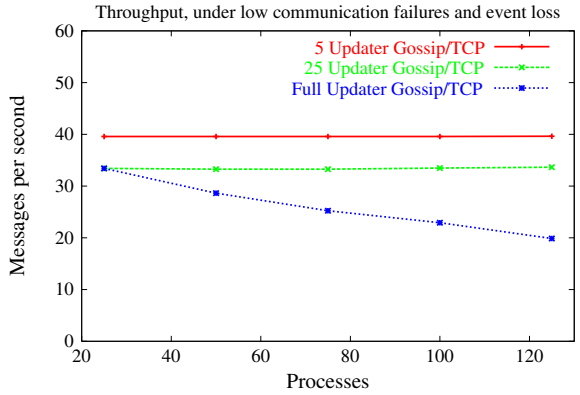
The Two-Layer protocol is implemented in an object oriented, modular manner in C++. The implementation of the causality layer follows the description in Section 5.1 and can be used with several group communication objects within our framework. Our PrCast is the ETTB-dissemination algorithm described in [20] together with the membership algorithm of lpbcast [43]. TCP was used as message transport (UDP is also supported). Multi-threading allows a process to send its gossip messages in parallel and a timeout ensures that the communication round has approximately the same duration for all processes.

6.2. Scalability results

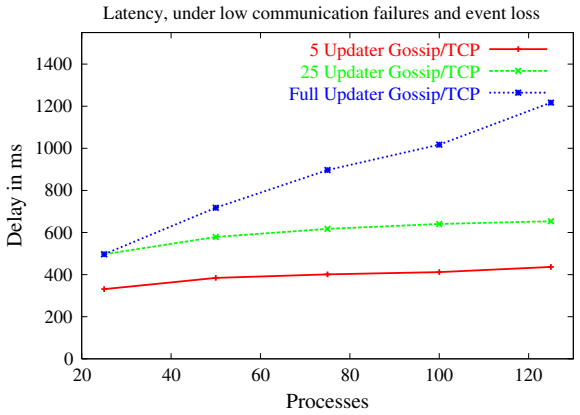
Our first experiment evaluates how the number of coordinators affects throughput, latency and message size. In our test application a process acts either as a coordinator, which produces a new event with probability p in each PrCast round, or as an ordinary cluster member. The product of the number of coordinators and p was kept constant (at 6).

DOI: 10.1002/cpe

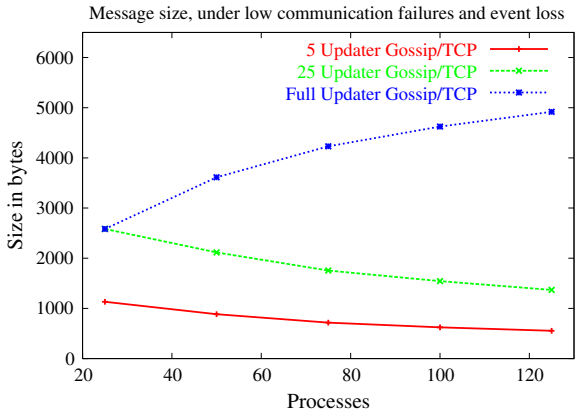
To focus on the performance of the causality layer the PrCast was configured to satisfy the goal of each event reaching 250 processes w.h.p. (the fan-out was 4 and the event termination time was 5 hops). PrCast was allowed to know all members to avoid side effects of the membership scheme. The maximum number of events transported in each gossip message was 20. The size of the history buffer was 40 events, which according to [20] is high enough to prevent w.h.p. PrCast from delivering the same event twice. The duration of each PrCast round was tuned so that all experiments had approximately the same rate of TCP connection failures (namely 0.2%). Figure 4 compares three instances of the Two-Layer protocol: the full-updater instance where all processes



(a) Throughput



(b) Latency



(c) Message size

Figure 4. Throughput and latency with increasing number of cluster members

DOI: 10.1002/cpe

act as coordinators, the 5-updater and the 25-updater instances with 5 and 25 coordinators, respectively. The causality layer used the first recovery method, described in Section 5.1. The results show the impact of the size of the vector clock on the overall message size and throughput. For the protocols using a constant number of coordinators message sizes even decreased slightly with growing group size since the dissemination distributes the load of forwarding events better then, i.e. for large groups a smaller percentage of processes performs work on an event during the initial gossip rounds. However, for the full updater protocol messages grow larger with the number of coordinators which influences the observed latency and throughput. For growing group size the protocols with a fixed number of coordinators experience only a logarithmic increase in message delay and throughput remains constant while for the full-updater protocol latency increases linearly and throughput decreases.

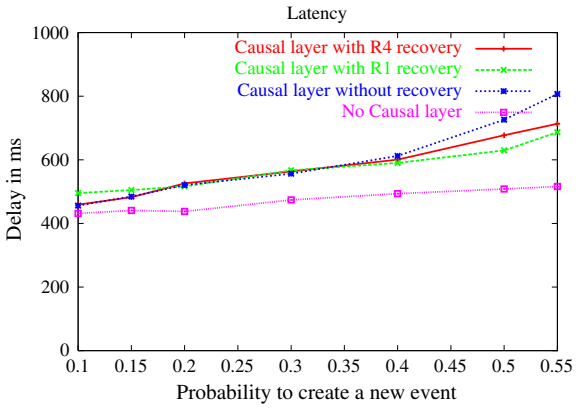
6.3. Comparison of recovery schemes

Our second set of experiments studies the effects of the causality layer and the recovery schemes in the Two-Layer protocol. Figure 5 compares the gossip protocol and the Two-Layer protocol with and without recovery. The recovery is done in two ways, both described in Section 5.2: (i) from the originator (marked "R1 recovery") and (ii) from k arbitrary processes (marked "R4 recovery" as the recovery fan-out k was 4). The recovery buffer size follows the analysis in Section 5.2, with the timeout-periods set to the number of rounds of the PrCast. Unlike the first experiment, the number of coordinators and processes was fixed to 25; instead varying values of p were used, to study the behaviour of the causality layer under varying load. Larger p values imply increased load in the system; at the right edge of the diagrams approximately $n/2$ new events are multicast in each round. As the load increases, more events are reordered by the dissemination layer and message losses begin to occur due to buffer overflows, thus putting the causality layer protocols under stress. The results in Figure 5(b) show that the causality layer significantly reduces the amount of lost (ordered) events, in particular when the number of events disseminated in the system is high. With the recovery schemes almost all events could be delivered in optimistic causal order. With increasing load latency grows only slowly (cf. Figure 5(a)), thus manifesting scalability. The causality layer adds a small overhead by delaying events in order to respect the causal order. The recovery schemes do not add much overhead with respect to latency, while they significantly reduce the number of lost events. At higher loads the recovery schemes even improve latency since by recovering missing events causally subsequent events in H_p can be delivered before they time out. Figure 5(c) shows the success rate for the recovery attempts. The number of recovery attempts increase as the load in the system increases, when the load is low very few events need recovery (cf. the event loss without the causality layer in Figure 5(b)). There are three likely causes for a recovery to fail: (i) the reply arrives too late; (ii) the process(es) asked did not have the event; and (iii) the reply or request(s) messages were lost. The unexpectedly low success rate during low load for the R4 method could be because a PrCast may reach very few processes when a gossip message is lost early in the propagation of an event. Also note that as the load is low the number of missing events and recovery attempts is very small. However, as load and the number of recovery attempts increase the success rate converges towards the predicted outcome.

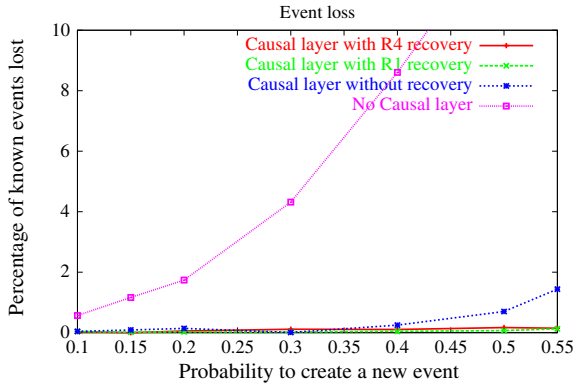
7. CONCLUSION

We have proposed lightweight causal cluster consistency, a hierarchical layer-based structure for multi-peer collaborative applications. To this end, we have presented and analysed a solution for a dynamic and fault-tolerant cluster management for event-based peer-to-peer dissemination systems. Since the protocol guarantees that never two processes perform some action corresponding to the same ticket of a cluster, the protocol is not only suitable to support the consistent management for distributed objects, but also for other common coordination tasks, such as resource management, controlling the number of concurrently disseminated events. The cluster management can be integrated at low cost in the two-level architecture. The duration of a round is longer than the time of a

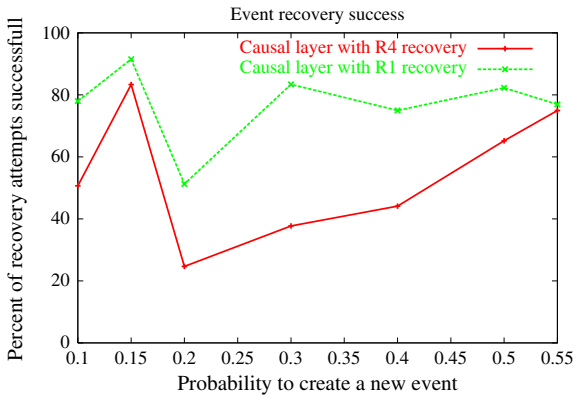
DOI: 10.1002/cpe



(a) Latency



(b) Message loss



(c) Percentage of successful recovery attempts

Figure 5. Event latency, loss and recovery behaviour under varying load with and without the causality layer.

multicast and in each round only a low number of messages are sent. Moreover we have shown how the protocol guarantees access to tickets in spite of failing processes.

The presented two-level architecture is a general architecture that can be applied on top of the standard Internet transport-layer services, and offers a layered set of services to the applications that need them. We presented a two-layer protocol for causal cluster consistency running on top of decentralised probabilistic protocols supporting group communication. Our experimental study, conducted by implementing and evaluating the proposed architecture as a two-layered protocol that uses standard Internet transport communication, shows that the approach scales well, imposes an even load on the system, and provides high-probability reliability guarantees.

DOI: 10.1002/cpe

REFERENCES

1. Miller DC, Thorpe JA. SIMNET:the advent of simulator networking. *Proceedings of the IEEE* 1995; **83**(8): 1114–1123.
2. Greenhalgh C, Benford S. A multicast network architecture for large scale collaborative virtual environments. In *Proceedings of the Second European Conference on Multimedia Applications, Services and Techniques*, Vol. 1242. Springer: Heidelberg, 1997; 113–128.
3. Carlsson C, Hagsand O. DIVE - a multi-user virtual reality system. *Proceedings of the IEEE Annual International Symposium on Virtual Reality*, Seattle, 1993; 394–400.
4. Rymaszewski M, *et al. Second life: The official guide*, 2nd edn. Sybex, 2007.
5. Pereira J, Rodrigues L, Monteiro MJ, Kermarrec AM. NEEM: Network-friendly epidemic multicast. *Proceedings of the 22nd symposium on reliable distributed systems*, 2003; 15–24.
6. Lehn M, Triebel T, Leng C, Buchmann AP, Effelsberg W. Performance evaluation of peer-to-peer gaming overlays. *Proceedings of Peer-to-Peer Computing*, 2010; 1–2.
7. Birman KP, Joseph TA. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems* 1987; **5**(1):47–76.
8. Birman KP, Schiper A, Stephenson P. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 1991; **9**(3):272–314.
9. Raynal M, Schiper A, Toueg S. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters* 1991; **39**(6):343–350.
10. Kshemkalyani AD, Singhal M. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing* 1998; **11**:91–111.
11. Chu Y-h, Rao SG, Zhang H. A case for end system multicast (keynote address). In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*. ACM Press: New York, NY, 2000; 1–12.
12. Jannotti J, Gifford DK, Johnson KL, Kaashoek MF, O'Toole JW, Jr. Overcast: reliable multicasting with on overlay network. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. USENIX Association: Berkeley, CA, USA, 2000; 14.
13. Rowstron AIT, Kermarrec A-M, Castro M, Druschel P. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop*, Vol. 2233. Springer-Verlag: London, UK, 2001; 30–43.
14. Zhuang SQ, Zhao BY, Joseph AD, Katz RH, Kubiatowicz JD. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM: New York, NY, USA, 2001; 11–20.
15. Tran DA, Hua KA, Do TT. A peer-to-peer architecture for media streaming. *IEEE Journal on Selected Areas in Communications* 2004; **22**(1):121–133.
16. Kostić D, Rodriguez A, Albrecht J, Vahdat A. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Vol. 37, SOSP '03. ACM Press: New York, Bolton Landing, NY, USA, 2003; 282–297.
17. Birman KP, Hayden M, Ozkasap O, Xiao Z, Budiu M, Minsky Y. Bimodal multicast. *ACM Transactions on Computer Systems* 1999; **17**(2):41–88.
18. Eugster PT, Guerraoui R, Handurukande SB, Kermarrec AM, Kouznetsov P. Lightweight probabilistic broadcast. *ACM Transaction on Computer Systems* 2003; **21**(4):341–374.
19. Kermarrec AM, Massoulié L, Ganesh AJ. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems* 2003; **14**(3):248–258.
20. Koldehofe B. Buffer management in probabilistic peer-to-peer communication protocols. *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS '03)*, 2003; 76–85.
21. Alvisi L, Doumen JM, Guerraoui R, Koldehofe B, Li H, van Renesse R, Tredan G. How robust are gossip-based communication protocols?. *ACM SIGOPS Operating Systems Review* 2007; **41**(5):14–18.
22. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
23. Charron-Bost B. Concerning the size of logical clocks in distributed systems. *Information Processing Letters* 1991; **39**(1):11–16.
24. Ådahl J. Shared resource for collaborative editing over a wireless network. *Master's Thesis*, Chalmers University of Technology, 2011.
25. Baldoni R, Prakash R, Raynal M, Singhal M. Efficient Δ-causal broadcasting. *International Journal of Computer Systems Science and Engineering* 1998; **13**(5):263–271.
26. Rodrigues L, Baldoni R, Anceaume E, Raynal M. Deadline-constrained causal order. *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, 2000; 234–241.
27. Mattern F. Virtual time and global states of distributed systems. *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989; 215–226.
28. Fidge CJ. Timestamps in message-passing systems that preserve partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 1988; 56–66.
29. Koldehofe B. Simple gossiping with balls and bins. *Studia Informatica Universalis* 2004; **3**(1):43–60.

DOI: 10.1002/cpe

30. Li HC, Clement A, Wong EL, Napper J, Roy I, Alvisi L, Dahlin M. Bar gossip. *Proceedings of the 7th symposium on Operating systems design and implementation OSDI '06*, 2006; 191–204.
31. Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H. Chord: A scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*. ACM Press: New York, August 2001; 149–160.
32. Alima LO, Ghodsi A, Brand P, Haridi S. Multicast in DKS(N; k; f) overlay networks. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS '03)*, Vol. 3144. Springer-Verlag, 2003; 83–95.
33. Ratnasamy S, Francis P, Handley M, Karp R, Shenker S. A scalable content-addressable network. *ACM SIGCOMM Computer Communication Review* 2001; **31**:161–172.
34. Rowstron A, Druschel P. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Vol. 2218. Springer-Verlag: London, UK, 2001.
35. Zhao BY, Huang L, Stribling J, Rhea SC, Joseph AD. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications* 2004; **22**(1):41–53.
36. Abraham U, Dolev S, Herman T, Koll I. Self-stabilizing l-exclusion. *Theoretical Computer Science* 2001; **266**(1-2):653–692. DOI: 10.1016/S0304-3975(00)00325-X.
37. Afek Y, Dolev D, Gafni E, Merritt M, Shavit N. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Languages and Systems* 1994; **16**(3):939–953.
38. Gidenstam A, Koldehofe B, Papatriantafilou M, Tsigas P. Dynamic and fault-tolerant cluster management. *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, 2005; 237–244.
39. Gidenstam A, Koldehofe B, Papatriantafilou M, Tsigas P. Lightweight causal cluster consistency. In *Proceedings of the Conference on Innovative Internet Community Systems (I^2CS '05)*, Vol. 3908, Lecture Notes in Computer Science. Springer: Heidelberg, 2005; 17–28.
40. Koch GG, Tariq MA, Koldehofe B, Rothermel K. Event processing for large-scale distributed games. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*. ACM Press: New York, NY, 2010.
41. Ahamad M, Neiger G, Kohli P, Burns JE, Hutto PW. Casual memory: Definitions, implementation and programming. *Distributed Computing* 1995; **9**(1):37–49.
42. Motwani R, Raghavan P. *Randomized algorithms*. Cambridge University Press: Cambridge, 1995.
43. Eugster PTh, Guerraoui R, Handurukande SB, Kermarrec A-M, Kouznetsov P. Lightweight probabilistic broadcast. *Proceedings of the International Conference on Dependable Systems and Networks*, 2001; 443–452.

DOI: 10.1002/cpe