Lightweight Causal Cluster Consistency

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

[1] Department of Computer Science and Engineering
Chalmers University of Technology
{andersg|ptrianta|tsigas}@cs.chalmers.se
[2] School of Computer and Communication Science, EPFL
boris.koldehofe@epfl.ch

Abstract. Within an effort for providing a layered architecture of services supporting multi-peer collaborative applications, this paper proposes a new type of consistency management aimed for applications where a large number of processes share a large set of replicated objects. Many such applications, like peer-to-peer collaborative environments for training or entertaining purposes, platforms for distributed monitoring and tuning of networks, rely on a fast propagation of updates on objects, however they also require a notion of consistent state update. To cope with these requirements and also ensure scalability, we propose the *cluster consistency* model. We also propose a two-layered architecture for providing cluster consistency. This is a general 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 protocol* implementing causal cluster consistency with predictable reliability, running on top of decentralised probabilistic protocols supporting group communication. 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.

1 Introduction

Many applications like collaborative environments (e.g. [1,2,3]) 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 (a.k.a. *events*). Providing the infrastructure to support such applications and systems places demands for multi-peer communication, with guarantees on reliability, latency, consistency and scalability, even in the presence of failures and variable connectivity of the peers in the system. Applications building on such systems would also benefit from an event delivery service that satisfies the causal order relation, i.e. satisfies the "happened before" relation as described in [4].

The main focus of earlier research in distributed computing dealing with these issues has its emphasis in proving feasible, robust solutions for achieving reliable causal delivery in the occurrence of faults [5,6,7,8], rather than considering the aforementioned variations in needs and behaviour. Further, since the causal order semantics require 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, while 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 the latency, *optimistic causal order* [9,10] can be suitable for systems where events are associated with deadlines. In contrast to the 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* [11], which also allow detection of missing events and their origin. However, since the size of the vector timestamps grow linearly with the number of processes in the system one may need to introduce some bound on the growing parameter to ensure scalability.

Recent approaches for information dissemination use lightweight probabilistic group communication protocols [12,13,14,15,16,17]. These protocols allow groups to scale to many processes by providing reliability expressed with high probability. In [16] it is shown that probabilistic group communication protocols can perform well also in the context of collaborative environments. However, per se these approaches do not provide any ordering guarantees.

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 to 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 vector clocks used. However, the set of processes entitled to perform updates is not fixed and may change dynamically. Our proposed approach is in line with and inspired from recent approaches in multipeer information dissemination [12,13,14], where the aim is at what is called *predictable reliability*, guaranteeing that each event is delivered to all non-faulty destinations with a high-probability guarantee. We present a two-layer architecture implementing cluster consistency that can make use of lightweight communication algorithms which can in turn run using standard Internet 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.

Also of relevance and inspiration to this work is the recent research on peer-to-peer systems and in particular the methods of such structures to share information in the system (cf. e.g. [18,19,20,21,22]), as well as a recent position paper for atomic data access on CAN-based data management [23].

2 Notation and problem statement

Let $G = \{p_1, p_2, \ldots\}$ denote a group of processes, which may dynamically join and leave, and a set of replicated objects $B = \{b_1, b_2, \ldots\}$. Processes maintain replicas of objects they are interested in. Let B be partitioned into disjoint clusters $C_1, C_2, \ldots$ with $\cup_i C_i \subseteq B$. Further, let C denote a cluster and p a process in G, then we write also $p \in C$ if p is interested in objects of C. *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) may propose updates to objects in C at the same time. Processes which may propose updates are called *coordinators* of C. Let $Core_C$ denote the set of coordinators of C. The set of coordinators can change dynamically over time. Throughout the paper we will use the term *events* when referring to update messages sent or received by processes in a cluster.

The propagation of events is done 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 as appears in the literature [13,14,15] would be suitable. We refer to such protocols as *PrCast*. PrCast is assumed to provide the following properties: (i) an event is delivered to all destinations with high probability; and, (ii) decentralised and lightweight group membership, i.e. a process can join and leave a multicast group in a decentralised way and processes do not need to know all members of the group.

Within each cluster we apply vector timestamps of the type used in [24]. 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 time stamp t_2 if $\forall i \in \{1, \ldots, n\}\ t_1[i] \leq t_2[i]$ 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 $t_{e_1} < t_{e_2}$, while 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 paper 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 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 while seq_p^C is the sequence number of the last multicast event performed by p. In Section 3 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.

3 Layered architecture for optimistic causal delivery

This section proposes a layered protocol for achieving optimistic causal delivery. Here we assume that coordinators of a cluster are assigned to vector entries and that the coordinators of a cluster know each other. To satisfy these requirements we choose a decentralised and fault-tolerant cluster-management protocol [25] which can map a process to a unique identifier in the CCT-vector clock in a decentralised way and can inform all processes in $Core_C$ about this mapping.

Protocol description

The first of the two layers uses *PrCast* in order to multicast events inside the cluster (cf. pseudo-code description Algorithm 1). 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. [24] and is adapted and enhanced to provide the optimistic delivery service of the cluster consistency model and the recovery procedure for events that may be missed due to *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

$$to\_deliver\_before\_ids(e) = \\ \{\langle i,j\rangle \mid (\forall i \neq index(e)\ .\ T_p^C[i] < j \leq t_e[i]) \vee (i = index(e) \wedge T_p^C[i] < j < t_e[i])\}.$$

If there exist any such events, e will be enqueued in H_p^C until it becomes obsolete (prior to that process p may "pull" missing events —see below). 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 $t[j] = seq_p^C$.

Since 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 since the start of the dissemination of e, and 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 $\forall i \; t_e[i] < T_p^C[i]$). Below we will present and analyse a another recovery method that enhances the throughput and the fault-tolerance.

Properties of the protocol. The PrCast protocol provides a delivery service that guarantees that an event will reach all its destinations with high probability, i.e. 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. If there are no process, link or timing failures, reliable point to point communication succeeds in recovering all missing events, and thus provide causal order without any message loss. The following lemma is straightforward, following the analysis in [24].

Lemma 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.*

Event recovery procedure, fault-tolerance and throughput

The throughput and fault-tolerance of the protocol can be increased by introducing redundancy in the recovery protocol. All processes could be 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 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 [15], 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

Algorithm 1 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.
On p creates e
  t_e := T_p^C; t_e[p] := seq_p^C; seq_p^C := seq_p^C + 1 /* Create timestamp t_e */
  PrCast(⟨e, t_e⟩)
  Insert e into recovery buffer B
On p receives ⟨e, t_e⟩
  Insert e into recovery buffer B
  if e can be delivered then
    deliver(e)
    for all e' ∈ H_p ∪ R that can be delivered
      deliver(e')
  else
    if e is not delivered or obsolete then
      delay(e, time_to_terminate)
On timeout(e, time_to_terminate)
  for all eid ∈ to_deliver_before_ids(e) not in H_p ∪ R  and eid not already under recovery
    send(⟨RECOVER, eid⟩) to source(eid) or to k arbitrary processes in cluster
  delay(e, time_to_recover)
On timeout(e, time_to_recover)
  for all e' ∈ to_deliver_before(e) ∩ (H_p ∪ R) that can be delivered
    deliver(e')
  deliver(e)
  for all e' ∈ H_p that can be delivered
    deliver(e')
On p receives ⟨RECOVER, source(e'), eid⟩
  if p has e with identifier eid in its buffer then
    respond( ⟨ACKRECOVER, e, e_t⟩ )
On p receives ⟨ACKRECOVER, e, e_t⟩
  Insert e into recovery buffer
  if e can be delivered then
    deliver(e)
    for all e' ∈ R ∪ H_p that can be delivered
      deliver(e')
  else
    if e is not delivered or obsolete then
      R := R ∪ {e}
On deliver(e)
  ∀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
```

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 at least as long in the queue 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 an issue here). Below 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 [15,26], 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 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 $2npT$.*

Due to space constraints, please see our technical report [27] for the proofs. To estimate T, we can use the estimated duration of a PrCast, e.g. as in [15]. 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, since 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, since 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 1 we get:

Lemma 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 non-failed process in the subset.*

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

Theorem 2. *In a system of n processes where each one may fail independently with probability $p_f \leq k/(2n)$ 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.*

Note that requesting recovery only once and not propagating the recovery messages is good because in cases of high loss due to networking problems we do not flood the network with recovery messages. Compared to 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 FIFO replacement. Another option is an aging scheme, e.g. based on the number of hops the event has made. As shown in [15], 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 [15] that uses an aging method to remove events from process buffers and guarantees very good message stability, implies that the reliability is improved since fewer processes may need to recover events.

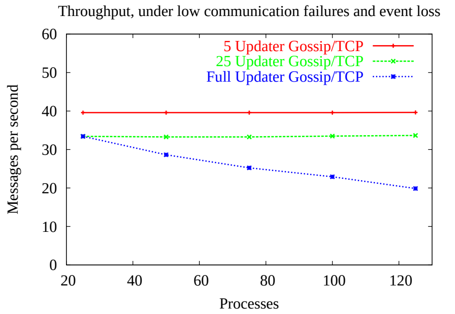
4 Experimental evaluation

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.

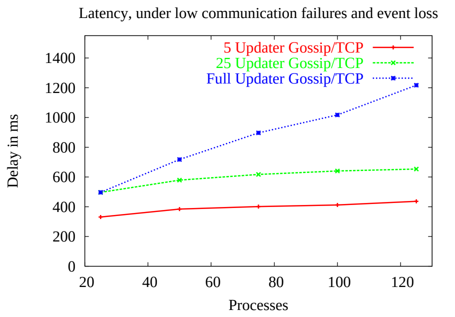
The evaluation of the Two-Layer protocol was done on 125 networked computers at Chalmers University of Technology. The computers were Sun Ultra 10 and Blade workstations running Solaris 9 and PC's running Linux distributed over a few different subnetworks of the university network. The average round-trip-time 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 3 and can be used with several group communication objects within our framework. Our PrCast is the ETTB-dissemination algorithm described in [15] together with the membership algorithm of lpbcast [13]. 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.

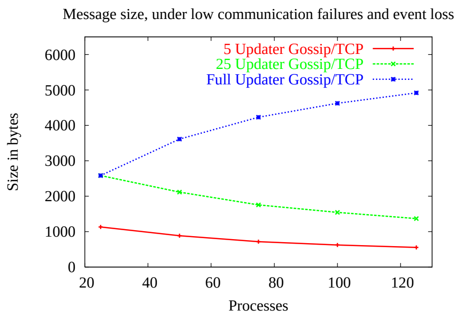
Our first experiment evaluates how the number of coordinators affect throughput, latency and message size. In our test application a process acts either as a coordinator, which produces a new event with probability p in each PrCast round, or as an ordinary cluster member. The product of the number of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



(a) Throughput

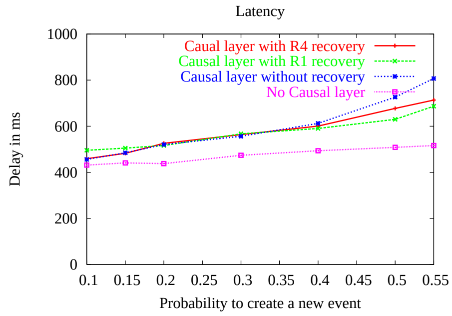


(b) Latency

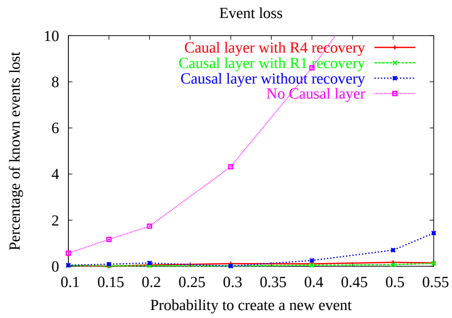


(c) Message size

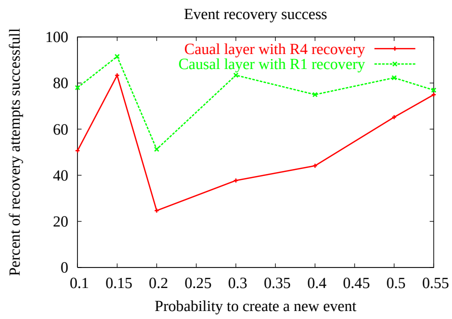
Fig. 1. Throughput and latency with increasing number of cluster members



(a) Latency



(b) Message loss



(c) Percentage of successful recovery attempts

Fig. 2. Event latency, loss and recovery behaviour under varying load with and without the causality layer.

message was 20. The size of the history buffer was 40 events, which according to [15] 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%). Fig. 1 compares three instances of the Two-Layer protocol: the full-updater instance where all processes 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 3. 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.

The second experiment studies the effects of the causality layer and the recovery schemes in the Two-Layer protocol. Fig. 2 compares the gossip protocol and the Two-Layer protocol with and without recovery. The recovery is done in two ways, both described in Section 3: (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 3, 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 Fig. 2(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. Fig. 2(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. Fig. 2(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 Fig. 2(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.

5 Discussion and future work

We have proposed lightweight causal cluster consistency, a hierarchical layer-based structure for multi-peer collaborative applications. This is a general architecture, 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 also 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.

Future work include complementing this service architecture with other consistency models such as total order delivery with respect to objects. Object ownership and caching are other topics that is worth studying.

References

1. Miller, D.C., Thorpe, J.A.: SIMNET:the advent of simulator networking. In: Proc. of the IEEE. Volume 8 of 83. (1995) 1114–1123
2. Greenhalgh, C., Benford, S.: A multicast network architecture for large scale collaborative virtual environments. In: Proc. of the 2nd European Conf. on Multimedia Applications, Services and Techniques. Volume 1242 of LNCS., Springer-Verlag (1997) 113–128
3. Carlsson, C., Hagsand, O.: DIVE - a multi-user virtual reality system. In: Proc. of the IEEE Annual Int. Symp. (1993) 394–400
4. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In: Communications of the ACM. Volume 7 of 21. (1978) 558–565
5. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failure. ACM Transactions on Computer Systems **5** (1987) 47–76
6. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems **9** (1991) 272–314
7. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Information Processing Letters **39** (1991) 343–350
8. Kshemkalyani, A.D., Singhal, M.: Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. Distributed Computing **11** (1998) 91–111
9. Baldoni, R., Prakash, R., Raynal, M., Singhal, M.: Efficient Δ-causal broadcasting. Int. Journal of Computer Systems Science and Engineering **13** (1998) 263–269

10. Rodrigues, L., Baldoni, R., Anceaume, E., Raynal, M.: Deadline-constrained causal order. In: Proc. of the 3rd IEEE Int. Symp. on Object-oriented Real-time distributed Computing. (2000)
11. Mattern, F.: Virtual time and global states of distributed systems. In: Proc. of the Int. Workshop on Parallel and Distributed Algorithms. (1989) 215–226
12. Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Transactions on Computer Systems **17** (1999) 41–88
13. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kermarrec, A.M., Kouznetsov, P.: Lightweight probabilistic broadcast. In: Proc. of the Int. Conf. on Dependable Systems and Networks. (2001) 443–452
14. Ganesh, A.J., Kermarrec, A.M., Massoulié, L.: Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In: Proc. of the 3rd Int. COST264 Workshop. Volume 2233 of LNCS., Springer-Verlag (2001) 44–55
15. Koldehofe, B.: Buffer management in probabilistic peer-to-peer communication protocols. In: Proc. of the 22nd Symp. on Reliable Distributed Systems, IEEE (2003) 76–85
16. Pereira, J., Rodrigues, L., Monteiro, M., Kermarrec, A.M.: NEEM: Network-friendly epidemic multicast. In: Proc. of the 22nd Symp. on Reliable Distributed Systems, IEEE (2003) 15–24
17. Baehni, S., Eugster, P.T., Guerraoui, R.: Data-aware multicast. In: Proc. of the 5th IEEE Int. Conf. on Dependable Systems and Networks. (2004) 233–242
18. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: Proc. of the ACM SIGCOMM 2001 Conf., ACM Press (2001) 149–160
19. Alima, L.O., Ghodsi, A., Brand, P., Haridi, S.: Multicast in DKS(N; k; f) overlay networks. In: Proc. of the 7th Int. Conf. on Principles of Distributed Systems. Volume 3144 of LNCS., Springer-Verlag (2003) 83–95
20. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: ACM SIGCOMM Computer Communication Review. Volume 31. (2001) 161–172
21. Rowstron, A., Druschel, P.: Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In: Proc. of the 18th IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware). Volume 2218 of LNCS., Springer-Verlag (2001)
22. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22** (2004) 41–53
23. Lynch, N., Malkhi, D., Ratajczak, D.: Atomic data access in distributed hash tables. In: Proc. of the 1st Int. Workshop on Peer-to-Peer Systems. Volume 2429 of LNCS., Springer-Verlag (2002) 295–305
24. Ahamad, M., Neiger, G., Kohli, P., Burns, J.E., Hutto, P.W.: Casual memory: Definitions, implementation and programming. Distributed Computing **9** (1995) 37–49
25. Gidenstam, A., Koldehofe, B., Papatriantafilou, M., Tsigas, P.: Dynamic and fault-tolerant cluster management. In: Proc. of the 5th IEEE Int. Conf. on Peer-to-Peer Computing, IEEE (2005)
26. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
27. Gidenstam, A., Koldehofe, B., Papatriantafilou, M., Tsigas, P.: Lightweight causal cluster consistency. Technical Report 2005-09, Computer Science and Engineering, Chalmers University of Technology (2005)