Dynamic and fault-tolerant cluster management

Anders Gidenstam* Boris Koldehofe† Marina Papatriantafilou* Philippas Tsigas*

Abstract

Recent decentralised event-based systems have focused on providing event delivery which scales with increasing number of processes. While the main focus of research has been on ensuring that processes maintain only a small amount of information on maintaining membership and routing, an important factor in achieving scalability for event-based peer-to-peer dissemination system is the number of events disseminated at the same time. This work presents a dynamic and fault tolerant cluster management method which can be used to coordinate concurrent access to resources in a peer-to-peer system. In the context of event-based dissemination systems the cluster management can be used to control the number of concurrently disseminated events. We present and analyse an algorithm implementing the proposed cluster management model in a fault-tolerant and decentralised way. The algorithm provides for each cluster a limited set of tickets. A process which has obtained a ticket may send events corresponding to the resources of the cluster. The algorithm guarantees that no two processes ever issue an event corresponding to the same ticket at the same time. The cluster management model on its own has interesting properties which can be useful for many peer-to-peer applications.

1. Introduction

Many applications like collaborative applications rely on an event-based dissemination service, for instance to exchange information on the state of shared replicated objects. For some applications there may be a large number of processes involved. Peer-to-peer dissemination algorithms for structured and unstructured networks have been studied to provide scalable event dissemination for a large number of processes. A lot of work has focused on providing delivery guarantees in the occurrence of dynamical joining and leaving processes by maintaining a low amount of resources locally at each process.

*Department of Computer Science and Engineering, Chalmers University of Technology. {andersg,ptrianta,tsigas}@cs.chalmers.se

†School of Computer and Communication Science, Swiss Federal Institute of Technology in Lausanne (EPFL). boris.koldehofe@epfl.ch

Current peer-to-peer dissemination systems rely on a good behaviour of each peer such that the overall number of events disseminated at the same time remains sufficiently small. A common assumption is that the rate of all incoming events remains constant. The reason is that there exists a limit for the amount information which can be stored locally, but also the amount of information which one can send in a message per time unit is bounded by the physical constraints of computer networks. Since often the dissemination of an event is triggered by local decisions it is a difficult problem to control the amount of events which are disseminated at the same time. Once this rate exceeded the assumptions made by the dissemination system, the dissemination system cannot provide the original guarantees.

Here we address this problem by 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. To coordinate access to the resources, a cluster issues a finite set of enumerated tickets. Processes which received a ticket from the cluster receive the right to perform some action, for instance to disseminate an event corresponding to a resource, or to use a particular entry of a vector clock to issue causally ordered events [9]. In order to prevent conflicts the cluster management needs to ensure in a decentralised fashion that, in spite of continuously joining and leaving as well as failing processes, never two processes perform an action corresponding to the same ticket at the same time. Moreover, one needs to ensure liveness by providing the possibility to reclaim tickets from processes that have crashed.

In this work we present an algorithm which can manage the cluster in the described way. Besides proving the correctness of the algorithm, we also present an analysis of availability of tickets depending on the failure rate and the amount of tickets maintained by non-faulty processes.

Structure of the paper. In Section 2 we describe the problem and introduce notation and definitions. Then we present two algorithms implementing a dynamic cluster management. The protocol of Section 3 works in the absence of failures and illustrates the basic idea, while Section 4 describes and proves a fault-tolerant membership protocol. In Section 5 we discuss related work on resource management in peer-to-peer applications and in the subsequent section we conclude with a discussion of the presented results and future work.

2. Notation and problem statement

Consider a peer-to-peer system supporting a large number processes (each process is considered to be a peer) to join and leave the system dynamically. The processes are said to form a group denoted by $G = \{p_1, p_2, \ldots\}$. Processes in G maintain a set of resources $R = \{r_1, \ldots r_l\}$. We assume the set of resources is partitioned into several disjoint clusters $C_1, C_2, \ldots$ with $\cup_i C_i \subseteq R$. Processes which are interested in certain resources need to join the respective cluster and will be informed afterwards about events corresponding to all resources maintained inside the cluster. A process which wishes to create events corresponding to a resource inside a cluster need to obtain a ticket of the cluster. For a cluster C there exists a maximum of n tickets where n is known to the processes which joined C.

Processes which own a ticket 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 messages which where sent with respect to a ticket of the cluster.

An algorithm implementing the dynamic cluster management needs to implement the following operations:

- *Ordinary joining/leaving a cluster.*
- *Coordinator joining/leaving the core of a cluster.*

Any ordinary process in G can perform a join or leave operation on C corresponding to the ordinary join and leave operation of the underlying multicast primitive. With respect to cluster management we will also call these operation *join* and *leave*. An ordinarily joined process will be able to observe events related to resources of a cluster.

In order to become a coordinator in a cluster C, i.e. to become member of $Core_C$ and be able to send events, a process performs an operation called *cjoin*. If process p performs a *cjoin* operation, p becomes assigned to be the owner of a unique ticket of C. When p performs a *cleave* operation it will release its ticket and cannot send events related to resources of the cluster after that. The tickets released by p may then be reused by any other process performing a *cjoin* operation.

For correct cluster management it is essential that there are never two or more coordinators that own the same tickets within the cluster at the same time. The ticket of a process that performed a *cleave* or has failed should eventually be reusable for other processes. Moreover, the cluster management should perform well even if a large number of processes concurrently perform *cjoin* operations.

Using a single process for cluster management is the simplest solution. However, if the cluster manager fails, then no processes can perform *cjoin* or *cleave*. Finding a new coordinator reduces to the agreement problem.

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 [7, 8, 11] would be suitable. We refer to such protocols as *PrCast*. PrCast is assumed to provide i) that an event is delivered to all destinations with high probability, 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.

3. 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. In order to illustrate the basic idea we assume in this Section that communication is reliable and processes do not fail. In Section 4 we show how to extend the presented ideas under a realistic failure model.

We assume that tickets form a cyclic relation according to their number, i.e. the succeeding ticket to ticket i is ticket $i - 1 \bmod n$, while the preceding ticket to ticket i is ticket $i + 1 \bmod n$. Each process which becomes coordinator of the cluster will own one ticket. Let i be the ticket owned by process p. 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 define q the dth closest successor (predecessor) of p, if the process is reachable in d steps from p by following the chain of successors (predecessors) starting at p.

In order to manage tickets, the processes which own tickets become also coordinator of a subset of the 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 S_p denote the set of tickets coordinated by p. Formally, we write

$$S_p = \{\, l \mid l = i - k \bmod n,\ 0 \leq k < \min\{m \mid j = i - m \bmod n,\ m > 0\}\}.$$

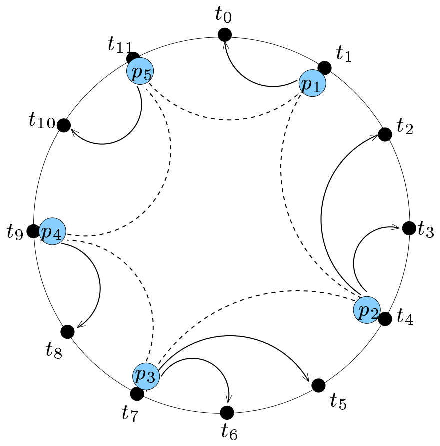

Figure 1. Illustration on how processes maintain and coordinate tickets of a cluster. An arrow from process p_i to a ticket indicates that p_i is the respective coordinator.

Figure 1 gives an example of how processes maintain and coordinate tickets, e.g. p_2 owns ticket 4 and coordinates the tickets $\{2, 3\}$.

Lemma 3.1 *Let C denote a cluster with* $\text{Core}_C \neq \emptyset$ *and no two processes own the same tickets. Then,*

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

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 then p sends 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 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

Algorithm 1 Cluster management in the absence of failures

VAR

Cview_p:	vector of processes
ImmedSucc_p:	immediate successor of p
ImmedPred_p:	immediate predecessor of p
state_p:	state variable

Message types:

CJOIN, CLEAVE, ACKJOIN, ACKSUCC, ACKCLEAVE, REJECT

Init$_p$:

state_p = joining
Send $\langle$CJOIN, $p\rangle$ to a known coordinator in $Core_C$.

Initialisation of variables when cjoin accepted

On *p receives $\langle ACKCJOIN, i, j, Cview\rangle$ from q*
  Cview_p = Cview
  p becomes the coordinator for all entries Cview$[i]$ until Cview$[j-1]$
  ImmedSucc_p = Cview$[j]$
  $\text{ImmedPred}_p = q$
  Send $\langle$NEWSUCC$\rangle$ to Cview$[j]$

Successor acknowledged

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

Receiving a cjoin request

On *p being coordinator of Cview$[i]$ until Cview$[j-1]$ receives $\langle CJOIN\rangle$ from 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 $t \in S_p$.
      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 *p being coordinator of Cview$[i]$ until Cview$[j-1]$ receives $\langle NEWSUCC\rangle$ from q*
  if state_p = leaving **then**
    Send $\langle$CLEAVE, Cview$[j]\rangle$ to q
  else
    Send $\langle$ACKSUCC$\rangle$ to q
    $\text{ImmedPred}_p = q$
  end if

Leaving the cluster

On *p being coordinator of Cview$[i]$ until Cview$[j-1]$ decides to leave the cluster*
  state_p = leaving
  Serve all previously received cjoin and cleave requests
  Send $\langle$CLEAVE, Cview$[j]\rangle$ to ImmedPred_p

Receiving a cleave request

On *being coordinator of Cview$[i]$ until Cview$[j-1]$ receives $\langle CLEAVE, r\rangle$ from q*
  if p is not serving another cjoin and $\text{state}_p \neq$ leaving **then**
    state_p = exclusion
    Send $\langle$ACKCLEAVE, $q\rangle$
    Send $\langle$NEWSUCC, $r\rangle$
    $\text{ImmedSucc}_p = r$
  end if

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 sent 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 sent 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 3.1, recall that according to Lemma 3.1 correctly preserving the relation among successors and predecessors, suffices to guarantee unique assignment of processes to tickets. This is shown in Lemma 3.2.

Lemma 3.2 *Let q be a coordinator in* Core_C *with successor r, serving a* cjoin *operation of p. Then*

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

Theorem 3.1 *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* $p, q \in \mathrm{Core}_C$

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

4. Supporting link and process failures

In the following we present an algorithm which extends the previous framework of Section 3 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, say 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*. Since 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.

The algorithm performs in rounds, where the time between two consecutive rounds is assumed to be long enough to host a PrCast, i.e. 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. Algorithm 2, Algorithm 3, and Algorithm 4), and below we present the ideas informally. During a round the algorithm maintains the following two invariants:

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

The first invariant is achieved by the processes in $Core_C$ sending *ALIVE* messages to their $2k + 1$ 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 leaves $Core_C$.

In order to manage the exclusion scheme, a process p maintains two sets denoted by L_p and R_p. The set L_p is used to store p's "knowledge" on its $2k + 1$ predecessors (this information is received from its immediate predecessor), while R_p contains the information on p's last successful transmission to p's immediate successor consisting of

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 sent an *ALIVE* message to p 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
P_{exclude}: probability to start exclusion algorithm after weakly detecting a faulty successor

Message types:

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

Init$_p$**:**

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)
  if $|$ALIVE$_p \cap L_p| < 2k+1$ **then**
    ThinkIamDisconnected $= true$
    exit loop
  end if
  Send $\langle$ALIVE, $p\rangle$ to $2k+1$ closest successors in *Cview*.
  if 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
    TempRounds$_p$ = TempRounds$_p - 1$
  end if

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 $Core_C$, i.e. if $Cview_p[i] = q$ holds, then p assumes q to be a coordinator owning ticket i .

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 received a successfully transmitted *UPDATE* message from its direct predecessor proposing a new set L'_p, which includes $2k+1$ predecessors of p, then p sets $L_p = L'_p$.

If p may proceed, it creates $2k+1$ *ALIVE* messages and sends them to the $2k+1$ closest successors known from *Cview*. Moreover, it sends to its direct successor an *UPDATE* message consisting of a set denoted R'_p. R'_p contains

Algorithm 3 Handling of messages

Initialisation of variables when cjoin succeeds

On *p receives $\langle$ACKCJOIN, L, i, j, Cview$\rangle$ from q*
  Cview$_p$ = Cview
  $L_p = L$
  $R_p = \emptyset$
  p becomes the coordinator for all entries Cview$[i]$ until Cview$[j-1]$
  ImmedSucc$_p$ = Cview$[j]$
  ImmedPred$_p = q$
  TempRounds$_p = 0$
  Send $\langle$ALIVE, $p\rangle$ to $2k+1$ closest successors in Cview$_p$.

Handling of *UPDATE* messages

On receiving $\langle$UPDATE, $R\rangle$
  $L_p = R$

Receiving a cjoin request

On *p being coordinator of Cview$[i]$ until Cview$[j-1]$ receives $\langle$CJOIN$\rangle$ from q*
  if $(|S_p| > 1) \wedge$ (TempRounds$_p = 0$) **then**
    Select ticket $t \in S_p$.
    Cview$[t] = q$
    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, 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

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**
    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 $\geq k+1$ 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
      ThinkIamDisconnected = true
      exit loop
    end if
  else
    ThinkIamDisconnected = true
    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

the $2k$ closest predecessors in L_p and p itself. If p succeeds in sending *UPDATE(R'_p)* to its direct successor, 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* reachable, i.e. not detected 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 entries preceding q and succeeding p denoted by E_{pq}. Only if p receives $k+1$ messages from destinations in $R_p \cap L_q$ acknowledging the request, p becomes the *temporary coordinator*, otherwise p thinks it is considered as failed.

While being 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 to at most the distance from p's to q's ticket since in every round the closest alive process in E_{pq} is guaranteed to think 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 entries corresponding to E_{pq} for $\mathrm{dist}(p, q)$ rounds.

4.1. Correctness

In order to prove correctness of the membership algorithm of Section 4, 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 4.1 we first consider the behaviour of the algorithm when no failures occur.

Lemma 4.1 *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 3.*

The critical case to analyse is after process p initiated the exclusion of E_{pq}. Lemma 4.2 states that during a round the closest successor in E_{pq} will fail.

Lemma 4.2 *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*

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

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), while $p \not\prec q$ 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 4.2 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 latest i rounds after p was acknowledged.*

Theorem 4.1 *Algorithm 2 guarantees that two processes never have common tickets they either own or coordinate.*

Proof. Lemma 4.1 shows that only exclusion could cause any such conflicts. Assume that during an execution two alive processes r and s, are two processes coordinating common tickets. This implies that one process, say r was failed to be excluded, while s was inserted. Let p be the process which failed to exclude r and inserted s.

After p initiated the exclusion of E_{pq} with $r, s \in E_{pq}$, p switches state to become temporary coordinator for $\text{dist}(p, q)$ 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 thinks it is considered to have failed, contradicting that both r, and s were active. □

4.2. Performance and liveness properties

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, i.e. 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 non-faulty processes. Let α denote the fraction of tickets taken by non-faulty 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. While 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, i.e. we know $|Core_C| \geq \alpha n$. Since 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 [13], 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 $p_f < 1/2(1-\alpha)$.

5. Related Work

Many distributed applications like collaborative environments (e.g. [12, 10, 6]) use event-based dissemination to interact on a distributed shared state. In order to perform well for many processes, such systems rely on a middleware which provides scalable group communication, supports maintenance of membership information according to processes interest as well as fast dissemination of events in the system.

Recent approaches for information dissemination use lightweight probabilistic group communication protocols [5, 7, 8, 11, 14, 4]. These protocols allow groups to scale to many processes by providing reliability expressed with high probability. In [14] it is shown that probabilistic group communication protocols can perform well also in the context of collaborative environments. However, to guarantee a delivery with high probability one needs a control mechanism for the number of concurrently disseminated events as achieved by the cluster management protocol. In [9] we show how the cluster management protocol can be used to implement such a control mechanism which also provides causal delivery of events disseminated in the cluster.

Alternatively, recently proposed dissemination systems implement the publish/subscribe paradigm in combination with structured peer-to-peer systems [17, 20] For each region of interest the protocols construct an application level multicast tree. Also these protocols assume a maximum number of concurrently disseminated events. Otherwise the dissemination system may overload the source of a multicast-tree and perform unstable thereafter.

The way structured peer-to-peer systems share information in the system (cf. e.g. [18, 3, 15, 16, 19]) 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 since 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 paper and the l-exclusion problem [1, 2]. However, to the best of our knowledge, the solutions to the l-exclusion problem do not satisfy the cluster management problem requirements. Nevertheless, the solution to the cluster management problem proposed here could also serve as solution basis to the l-exclusion problem.

6. Discussion and future work

This paper 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 suitable for several coordination tasks, such as resource management, controlling the number of concurrently disseminated events, as well as consistency management for replicated distributed objects. The cost of combining the presented solution with an application is low since the duration of a round is longer than the time of a 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.

Current and future work deals with integrating the cluster management with existing peer-to-peer dissemination algorithms in order to increase reliability as well as achieve decentralised ordering of messages by maintaining small distributed vector timestamps.

References

[1] U. Abraham, S. Dolev, T. Herman, and I. Koll. Self-stabilizing l-exclusion. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 48–63. Carleton University Press, 1997.

[2] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):939–953, May 1994.

[3] L. O. Alima, A. Ghodsi, P. Brand, and S. Haridi. Multicast in DKS(N; k; f) overlay networks. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS '03)*, volume 3144 of *LNCS*, pages 83–95. Springer-Verlag, 2003.

[4] S. Baehni, P. T. Eugster, and R. Guerraoui. Data-aware multicast. In *Proceedings of the 5th IEEE International Conference on Dependable Systems and Networks (DSN 2004)*, pages 233–242, 2004.

[5] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[6] C. Carlsson and O. Hagsand. DIVE - a multi-user virtual reality system. In *Proceedings of the IEEE Annual International Symposium*, pages 394–400, Seattle, Sept. 1993.

[7] P. T. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001)*, pages 443–452, July 2001.

[8] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the Third International COST264 Workshop*, volume 2233 of *LNCS*, pages 44–55. Springer-Verlag, 2001.

[9] A. Gidenstam, B. Koldehofe, M. Papatriantafilou, and P. Tsigas. Lightweight causal cluster consistency. In *Proceedings of the Conference on Innovative Internet Community Systems (I^2CS'05) (To appear)*. Springer-Verlag, June 2005.

[10] C. Greenhalgh and S. Benford. A multicast network architecture for large scale collaborative virtual environments. In *Proceedings of the Second European Conference on Multimedia Applications, Services and Techniques - ECMAST'97*, volume 1242 of *LNCS*, pages 113–128. Springer-Verlag, 1997.

[11] B. Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS '03)*, pages 76–85. IEEE, Oct. 2003.

[12] D. C. Miller and J. A. Thorpe. SIMNET:the advent of simulator networking. In *Proceedings of the IEEE*, volume 8 of *83*, pages 1114–1123, Aug. 1995.

[13] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, june 1995.

[14] J. Pereira, L. Rodrigues, M. Monteiro, and A.-M. Kermarrec. NEEM: Network-friendly epidemic multicast. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS '03)*, pages 15–24. IEEE, Oct. 2003.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 161–172, 2001.

[16] A. Rowstron and P. Druschel. 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)*, volume 2218 of *LNCS*. Springer-Verlag, Nov. 2001.

[17] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop*, volume 2233 of *LNCS*, pages 30–43. Springer-Verlag, 2001.

[18] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, New York, Aug. 2001. ACM Press.

[19] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, and A. D. Joseph. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[20] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. 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*, pages 11–20. ACM Press, 2001.