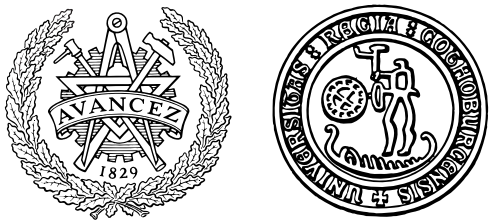
Thesis for the Degree of Doctor of Philosophy

# Distributed Algorithms and Educational Simulation/Visualisation in Collaborative Environments

Boris Koldehofe

CHALMERS | GÖTEBORG UNIVERSITY




Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2005

Distributed Algorithms and Educational Simulation/Visualisation
in Collaborative Environments
Boris Koldehofe
ISBN 91-7291-572-2



Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2254
ISSN 0346-718x

Technical Report no. 1 D
Department of Computer Science and Engineering
Distributed Computing and Systems

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: “Working hand-in-hand” by Juliana Koldehofe

Chalmers Reproservice
Göteborg, Sweden, 2005

**Abstract**

This thesis examines a general class of applications called *collaborative environments* which allow in real-time multiple users to share and modify information in spite of not being present at the same physical location. Two views on collaborative environments have been taken by examining algorithms supporting the implementation of collaborative environments as well as using collaboration to support educational visualisation/simulation environments in the context of distributed computing.

Important algorithmic aspects of collaborative environments are to provide scalable communication which allows interest management of processes and deal with modification of information among collaborators in an efficient and consistent manner. However, to support real-time interactions one may need to trade strong reliability guarantees for efficiency and scalability. This work examines lightweight decentralised peer-to-peer algorithms which can scale to a large number of processes and offer reliability expressed with probabilistic guarantees.

Besides proposing and evaluating peer-to-peer algorithms on support for large scale event dissemination, this work considers the impact of buffer management and membership in achieving stable performance even under critical system parameters. Moreover, we examine consistency management as well as interest management in combination with lightweight peer-to-peer dissemination schemes. Lightweight cluster consistency management allows a dynamic set of processes to perform updates on a set of processes which is observed in optimistic causal order. Topic-aware membership management supports lightweight dissemination schemes to propagate events which correspond only to their interest by providing a fair work distribution are the same time.

The second part deals with collaboration in the context of educational simulation/visualisation. We present and evaluate a visualisation/simulation environment for distributed algorithms called LYDIAN which helps studying distributed algorithms and protocols. Besides looking at what environments such as LYDIAN should provide in order to be successfully used in class, we consider the use of collaboration in providing more interactivity in simulation environments as well as a possibility for new visualisations of learning concepts to evolve.

# Preface

Since the start of the thesis there has been a rapid development in how people connect to the Internet. Every year the amount of Internet users connected via high bandwidth broadband networks increases significantly. Hand-in-hand with this development, demanding applications have evolved which allow a large number of users to cooperate together. Most solutions nowadays are bound to a client/server model, where the server maintains control in updating shared data of the application. This model allows one to easily maintain the applications data consistent, however a server also becomes a single point of failure and a bottleneck which may prevent an application to scale to many users. In order to use the potential of high bandwidth networks, a good distribution of application data is the key to provide scalability. Recent research on peer-to-peer systems has looked for solutions which provide a fair distribution of the work performed by an application in the presence of dynamic connections and disconnection of peers in the system. Each peer may act as server as well as client. Whereas peer-to-peer computing is commonly associated with the location of distributed data, peer-to-peer systems can also be used to perform aggregation or large-scale information dissemination.

As part of this thesis, peer-to-peer based algorithms on support for collaborative environments are examined and proposed. Collaborative environments allow multiple users and processes to interact in a virtual, shared state in real-time. This makes them interesting when building collaborative educational applications, for example, in the context of distance learning. In fact, the original motivation for this thesis stems from an educational application, LYDIAN, which is introduced and evaluated in Part II. LYDIAN is an educational simulation/visualisation environment for learning and teaching distributed algorithms. Collaborative interactions are discussed to increase the level of interactivity with a learning concept.

## Acknowledgements

While being a Ph.D student at Chalmers I had the pleasure and honour to receive support from many people. First of all, I would like to thank Philippas Tsigas, my supervisor, and Marina Papatriantafilou for all their advice and support. Marina and Philippas did not only contribute a lot to my work and my scientific development, but they also offered me personal advice and friendship. It was great to enjoy two excellent supervisors who can also complement one another so well as Marina and Philippas do.

I am also very much honoured by and grateful to Thierry Coquand, the examiner, and Luís Rodrigues, the opponent of the thesis. Thierry is also member of my local Ph.D. committee and followed and commented my work in several meetings.

I am also grateful to my colleagues who took time to discuss my work and who volunteered in offering feedback and comments. Especially the present and former members of the distributed computing and systems research group Anders Gidenstam, Dave Rutter, Håkan Sundell, Niklas Elmqvist, Patrick Eugster, Phuong Ha Hoai, and Yi Zhang helped me a lot. It was a pleasure to work with Anders on consistency management in collaborative environments, as well as with Phuong on supporting LYDIAN. During his stay with us, Patrick gave me useful feedback on my work in the area of group communication and motivated the analytical study of gossiping protocols. Håkan, who is a great office mate and friend, helped me in many occasions with respect to my work and aspects of everyday Swedish life.

I also wish to thank Rachid Guerraoui and the members of his research group at EPFL for kindly hosting me for two months as well as MiNEMA for supporting the visit to EPFL. With Sébastien Baehni, Sidath Handurukande and Oana Jurca we worked together on role-based distributed hash tables.

Further, I would like to thank Sven-Arne Andréasson and Tuomo Takkula. Sven-Arne supported the evaluation of our work on LYDIAN. Tuomo, a very nice fellow and friend to enjoy after-work, provided me with many pointers which facilitated my life enormously.

In general, I would like to thank all the members of the department of Computing Science at Chalmers for providing a friendly and helpful environment. Many members and Ph.D. students also enriched my life outside the department.

Finally, I owe many thanks to my family and friends from Sweden and Germany. Special thanks and kisses to Juliana and Sandrine whose contributions go beyond from what is visible in this thesis.

# Contents

# One

# Overview

Many applications, like collaborative text editing, video conferencing, multiplayer games, or educational applications aim at providing to a potentially large set of users or processes the possibility to interact in a shared, virtual state in real-time. Even though users and processes may be physically distant, they can perform interactions in a shared state to produce something, i.e. *collaboration* happens. For users to be able to collaborate efficiently, it is up to the application to provide a meaningful interface. At a lower level, however, the performance of the application and the semantics it can provide depend on the underlying communication model. As the amount of users and processes grows larger, latency increases and the application may not be able to offer the semantics promised to the user, i.e. a user detects inconsistencies with respect to the shared state.

This work studies distributed systems which aim at providing collaboration to multiple users and processes. In Part I we study system services and algorithms to provide them for a general class of applications called *collaborative environments*. We examine and propose different communication models and algorithms in support for event dissemination and interest management by considering reliability, fault tolerance, scalability, and consistency. In particular the focus is on lightweight peer-to-peer dissemination algorithms which can scale to a large number of processes and users. In addition to their support for fast dissemination, peer-to-peer solutions in a collaborative environment can be used to organise a dynamic and lightweight membership as well as to manage data replication.

Part II is devoted to the original motivation of this thesis, namely offering meaningful interactions for many possible users of an educational simulation/visualisation environment for distributed algorithms. We present an implementation and an evaluation of a learning environment called LYDIAN which gives evidence as to what interactive learning environments such as LYDIAN should provide to be

successfully used in class. Moreover, we examine ways of improving interactivity by means of collaboration as well as new approaches to educational visualisation building on collaboration.

The algorithmic aspects are presented in Part I, while the educational aspects are discussed in Part II. Both parts are based on a collection of papers combined with an introductory chapter for each. The contributions of the thesis are discussed in Chapter 2.6 and Chapter 7.4 with focus on algorithmic and educational aspects respectively. Part III summarises the basic findings of the thesis. In the following we give a brief description of the content of each chapter as well as pointers to the publications the chapter is based on.

Chapters of Part I:

***Foundations and Communication Models.*** Chapter 2 gives background information on the algorithmic aspects of this thesis and looks at collaborative environments from the perspective of consistency and event-based dissemination. Moreover, the contributions regarding the algorithmic aspects of this thesis are discussed in this chapter.

***Simple gossiping with balls and bins.*** Chapter 3 is based on publications which appeared in

- the *Proceedings of the 6th International Conference on Principles of Distributed Systems (OPODIS'02)*, pages 109–118, 2002.
- *Studia Informatica Universalis*, Volume 3, Number 1, pages 43-60, 2004.

The chapter presents an analysis of gossip-based event dissemination using a balls-and-bins paradigm to prove delivery of an event with high probability.

***Buffer management in probabilistic peer-to-peer communication.*** Chapter 4 appeared in

- *the Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS '03)*,pages 76–85, 2003.

Buffer management is an important aspect in the configuration of gossip based dissemination algorithms. This chapter evaluates and analyses buffer management in combination with different dissemination schemes.

***Lightweight causal cluster consistency.*** Chapter 5 is based on a technical report written together with Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. This paper deals with consistency management suitable for lightweight peer-to-peer dissemination protocols.

***A role-based distributed hash table.*** Chapter 6 is based on a technical report written together with Sébastien Baehni, Rachid Guerraoui, Sidath B. Handurukande, and Oana Jurca. The paper describes a membership which can support publish/subscribe schemes in fairly distributing the load of the dissemination according to the interest of processes and considers space efficiency.

Chapters of Part II:

***Collaborative Learning Using Simulation and Visualisation.*** Chapter 7 gives background information on the educational aspects of this thesis and discusses simulation and visualisation to support collaborative learning. The chapter discusses also the contributions with respect to educational simulation/visualisation in collaborative environments.

***LYDIAN.*** Chapter 8 is a journal version based on the following publications:

- Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 2003. Integrating a simulation/visualisation environment in a basic distributed systems course: A case study using LYDIAN. In *Proceedings of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'03)*, J. Impagliazzo, Ed. ACM Press, N.Y., 35–39.
- Holdfeldt, P., Koldehofe, B., Lindskog, C., Olsson, T., Petersson, W., Svensson, J., and Valtersson, L. 2002. Envidia: An educational environment for visualisation of distributed algorithms in virtual environments. In *Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2002)*. ACM press, 226.
- Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 2000. LYDIAN, an extensible educational animation environment for distributed algorithms. In *Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2000)*. ACM Press, 189.
- Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 1999. Distributed algorithms visualisation for educational purposes. In *Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'99)*. ACM Press, 103–106.

- Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 1998. Building animations of distributed algorithms for educational purposes. In *Proceedings of the 3rd Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'98)*. N.Y., 286.

This paper presents an educational environment for learning and teaching distributed algorithms called LYDIAN.

***Using Actors to Teach Self-Stabilisation.*** Chapter 9 is based on following publication:

- Koldehofe, B. and Tsigas, P. 2001. Using actors for an interactive animation in a graduate distributed system course. In *Proceedings of the 6th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2001)*. ACM press, 149–152.

This paper evaluates an animation using dramatisation to teach the concept of self-stabilisation.

# Part I

# Algorithmic Aspects in Collaborative Environments

# Two

# Foundations and Communication Models

*Collaborative environments* are a general class of applications which allow multiple users to interact on a shared state in real-time. Typical examples of such applications are platforms for educational, training or entertaining purposes as well as for distributed monitoring and tuning of networks. In contrast to human face-to-face collaboration, collaborative environments decouple the physical location of the user from the context of the application. Users can share a virtual space in spite of not being physically present at the same location.

A fundamental assumption for allowing user interactions is the access to a communication media. We look at collaborative environments in the context of computer networks. A collaborative process may involve not only human users, but also programs. By using terminology of distributed computing, we refer to "users" and "programs" by "processes" instead.

A *collaborative environment* can be defined as a space in which multiple processes share and modify the state of a set of common objects (information) in real-time. The state of such an environment is formed by a *world* of processes and objects (c.f. Figure 2.1). A collaborative environment offers the following to processes:

- a *membership service* which allows them to dynamically join and leave the world,
- an *interest management* which establishes awareness of objects depending on the context of a process,
- an *event notification service* which informs processes about state changes of

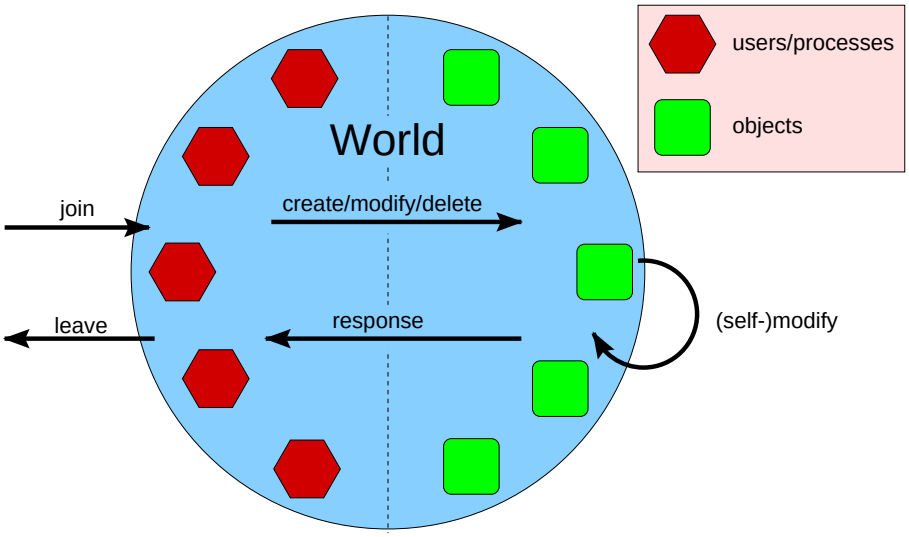



Figure 2.1: The global state of a collaborative environment.

objects and processes and allows processes to perform interleaving modifications to objects.

The demands of collaborative environments make it difficult to scale them to many users and objects. Interactions of an increasing number of users with the world's state require more synchronisation and induce a higher load on the communication traffic. Moreover, the complexity and size of objects play an important role. Therefore, it is often assumed that in the world communication happens by sending updates on the state of objects, instead of sending the whole state of an object. The real-time requirement demands that updates to the shared object should reach all interested group members fast. On the other hand, all group members should have an accurate estimation on the state of shared objects. An accurate state estimation should even be possible if some part of the system is not functioning correctly.

In other words, collaborative environments demand that the state of the system must be handled in a consistent way by offering fast and reliable communication. This requires one to choose both an appropriate consistency model as well as a suitable organisation of the way information is propagated among group members. Clearly, there is a tradeoff between the consistency guaranteed and the response time of an interaction made by a user.

The described problems are addressed by different research communities. Pos-

sible frameworks of organising objects and communication of collaborative environments are present in the area of *collaborative virtual environments* (CVE). CVEs are an interesting example of collaborative environments because CVEs belong to the most demanding application regarding scalability and consistency of the observed state. CVEs may also be the most studied application within the area of collaborative environments. However, the notion of consistency is studied very little. Most approaches provide a best effort to keep the view of the world's objects consistent for group members. The specification of a *consistency model* gives guarantees on how the system behaves on interactions of users and in this way helps to reason on the correctness of an application for such an environment. Consistency models and their performance have been formally analysed in the area of *distributed shared memories*. Although their descriptions often address read/write objects, many of the presented consistency models can be generalised to more complex objects.

In collaborative environments the interest in objects may vary among processes. Therfore, an interest management can help in reducing the amount of unnecessary sent messages. The *publish/subscribe* paradigm allows processes to express an interest by performing an operation called a *subscription*. Based on this subscription a process will be notified on any published *events* corresponding to the expressed interest.

The implementation of consistency models as well as publish/subscribe systems rely on the way events are propagated. In particular, consistency models exploit assumptions on how events, which can be issued from several sources, are propagated at the communication level. Possible requirements can be ordering guarantees for events and the reliability for an event to reach the respective destinations. Another important aspect from the perspective of a collaborative environment is to allow users to dynamically join and leave the system. These problems are examined in the area of *group communication*.

In the following sections we present in more detail aspects and related work of the respective research areas. In Section 2.1 an overview of existing work in collaborative virtual environments is presented. Section 2.2 discusses models known in the area of distributed shared memory. Publish/subscribe in order to provide interest management is introduced in Section 2.3, while Section 2.4 introduces the group communication paradigm. In Section 2.5 peer-to-peer systems and their application to large-scale event dissemination are presented. The algorithmic contributions of subsequent chapters are discussed in Section 2.6.

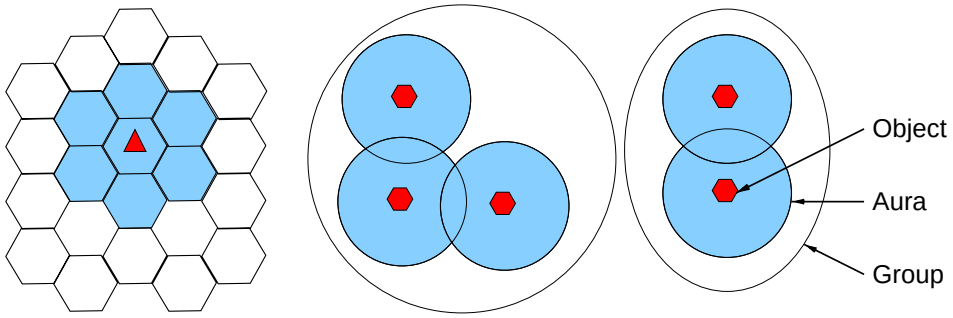
## 2.1 Management of distributed objects in collaborative virtual environments

A *collaborative virtual environment* (CVE) is a distributed multi-user system which provides an immersed three-dimensional view for each user as well as the possibility to interact in real-time with the system such that users have i) a shared sense of space, ii) a shared sense of presence, iii) a shared sense of time, iv) a way to communicate, v) a way to share (as defined in [Singhal and Zyda 1999]). The main difference from collaborative environments is the requirement to compute uniformly 3D-graphics data, i.e. all users have the same representation of the world. Often for collaborative environments each user may choose a different representation and multiple views can coexist at the same time. The state of objects in a CVE is complicated and even updates locally at a user can be computationally costly. CVEs usually favour to communicate on changes of objects by sending updates and try to avoid communication of the object's whole state. In order to scale such a system to many users, a framework for managing objects also requires a good distribution of objects.

Since the beginning of the 90s different frameworks for CVEs have been introduced. SIMNET [Miller and Thorpe 1995] is one of the oldest multi-user real-time environments originally created for military simulations. In order to scale the system well, all processes and objects of the simulation are considered to be autonomous and send continuous updates of changes to all other objects. A process considers only group members which were able to send state update within two local clock pulses. The time delays in the communication between two processes can lead to different views of the same object. SIMNET allows states of objects to be inconsistent among different processes, but tries to reduce the occurrence of inconsistencies. This is achieved by predicting the behaviour of objects and maintaining a history of previous states and updates. If a process observes an inconsistency because of receiving an update late, a process can recalculate a correct state and then behave to converge to the correct state. This method is often referred to as *dead reckoning*.

Much of SIMNETs architecture has influenced the IEEE standard on *distributed interactive simulation (DIS)* [IEEE 1993]. The standard defines events which can be used to interact with objects of a CVE. The objects of the CVE communicate by sending and receiving predefined *protocol data units (PDU)*. Different CVEs often can interact by supporting a subset of the PDUs used in DIS.

Fully distributed approaches like SIMNET can scale well (assuming the broadcasting of updates is handled efficiently), but they do not give guarantees to users as to how their interactions affect the system. Therefore, some CVEs require updates to reach all processes in the same order. For these CVEs the provided ordering




(a) (b)

Figure 2.2: Figure 2.2(a) shows a hexagonal division of the CVE. The dark filled areas symbolise the area of interest of an object represented by a triangle. In Figure 2.2(b) a spatial division of objects into groups is shown which uses the aura associated with an object.

guarantee by the communication media determines the consistency guaranteed.

The ordering of updates can be achieved in various ways. The simplest form is to use a central repository which orders the updates of objects. However, a central repository quickly becomes a bottleneck in such a system. This is why implementations like DIVE [Carlsson and Hagsand 1993; Hagsand 1996], rely on group communication where objects can be physically distributed. The group communication protocol guarantees the order in which updates reach their destinations (see also Section 2.4). The communication of updates in DIVE uses the ISIS [Birman and Joseph 1987a] process groups. All objects are replicated by a process. The processes in DIVE belong to a single group in which group members use the ISIS communication model to propagate updates. Depending on the consistency requirement, different ways of ordering can be chosen, e.g. a total ordering respecting also the local order of events for each process.

Maintaining all objects in a single group facilitates the application of a consistency model as discussed in Section 2.2. However, the approach is only feasible for small systems because every update still involves many messages to be propagated. With increasing number of objects in the system, latency of events will grow too large in order to allow real-time interactions. In order to support better scalability, one has to reduce the amount of updates propagated to the whole group.

MASSIVE [Greenhalgh and Benford 1995; Greenhalgh and Benford 1997] and NPSNET [Macedonia et al. 1995] aim to increase scalability by partitioning the

environment into spatial regions. Objects receive only information on other objects of an area of interest. NPSNET partitions the world in hexagonal regions, while MASSIVE uses volumes to partition the world into groups, supporting several levels of grouping. In order to detect when objects share a common interest, MASSIVE associates the objects of the CVE with an *aura*. The object's aura determines the extent to which interaction is possible with other objects of the world. A detection of intersecting auras takes care that those objects start communicating about each other's state, i.e. they become *aware* of each other. MASSIVE supports different degrees of awareness depending on the subjective importance of related objects. To measure awareness, MASSIVE applies two further properties called *focus* and *nimbus*. The focus determines the attention of the observer while nimbus reflects the visibility and orientation of the observed object. The scalability of MASSIVE depends on the number of objects with intersecting aura.

Although the discussed approaches are only a subset among the existing CVEs, they give a good representation of existing approaches to support large-scale collaborative environments. Approaches like SIMNET and NPSNET achieve a large number of users by sacrificing consistency requirements. Only a best effort is made by using techniques like dead reckoning in order to achieve consistency. Other approaches like DIVE aim for all processes to observe all operations in total order. Since the consistency requirement is very strong, the number of objects that can be supported is much smaller. Other approaches aim for a compromise between the two extreme ways of handling consistency, but do not give a formal definition as discussed in Section 2.2. The discussed approaches also illuminate the importance of efficient group communication and publish/subscribe schemes, which help to inform efficiently other processes about updates in the system, consider interest management and organise at the same time membership of users and objects to the world. MASSIVE and NPSNET exploit group communication by partitioning the world into several regions. Many CVEs rely only on IP-Multicast, but as discussed in Section 2.3 and Section 2.4 interest management, as well as reliability and scalability aspects may lead to other ways of implementing group communication.

## 2.2 Consistency in distributed shared memory

When many processes share common objects which are updated and read concurrently, the correctness of a computation may depend on processes having a consistent view of the shared object. Research in *distributed shared memory* introduced different consistency models for multi-processor systems with processors placed at different locations writing and reading to shared objects. Although the underlying communication to handle updates of the shared objects relies on message

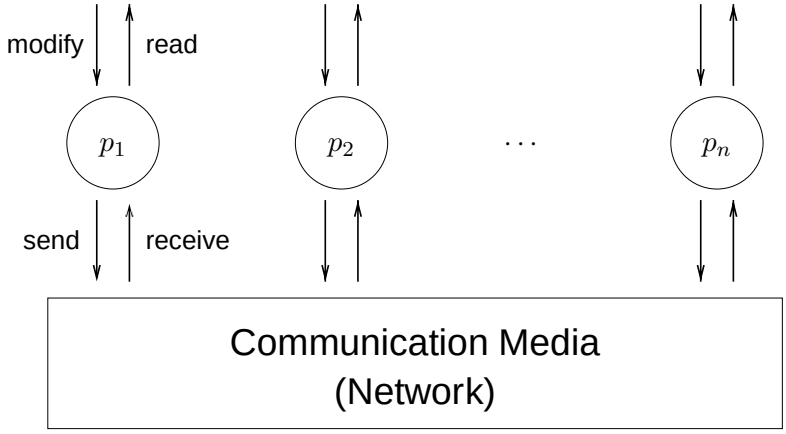



Figure 2.3: Interaction with a distributed shared memory. The application can communicate with familiar operations as read and modify with processes which implement the required memory consistency.

passing, the distributed shared memory gives users the possibility to interact with more familiar operations such as read/write (cf. Figure 2.3). In a distributed shared memory interactions are handled by processes replicating the state of the shared data objects and coordinating their changes with other processes. The consistency model specifies a contract between the user of the system and the communication model which determines how operations to shared objects can be ordered. In order to verify the correctness of programs for a system in which multiple processors modify shared objects, as it happens in collaborative environments, it is essential to specify a consistency model.

In general, the choice of a consistency model has an impact on how a programmer can interact with a distributed system. Programmers prefer to observe the operation as if they would interact with a single processor, i.e. all events happen in sequential order. Observing events in sequential order facilitates the creation of applications significantly because the users do not need to worry about concurrent operations interfering with each other. However, the stricter the model, i.e. the closer it is to provide serialisation of operations, the worse it scales with the number of processes of the system. This is why weaker consistency models, requiring more thinking by the programmer, have evolved.

In this section we introduce important consistency models (c.f. Figure 2.4) which support consistent modification of shared objects (much of the presented material is based on [Mosberger 1993] and [Tanenbaum 1995]). The descriptions of the models use read/write operations, while in collaborative environments often

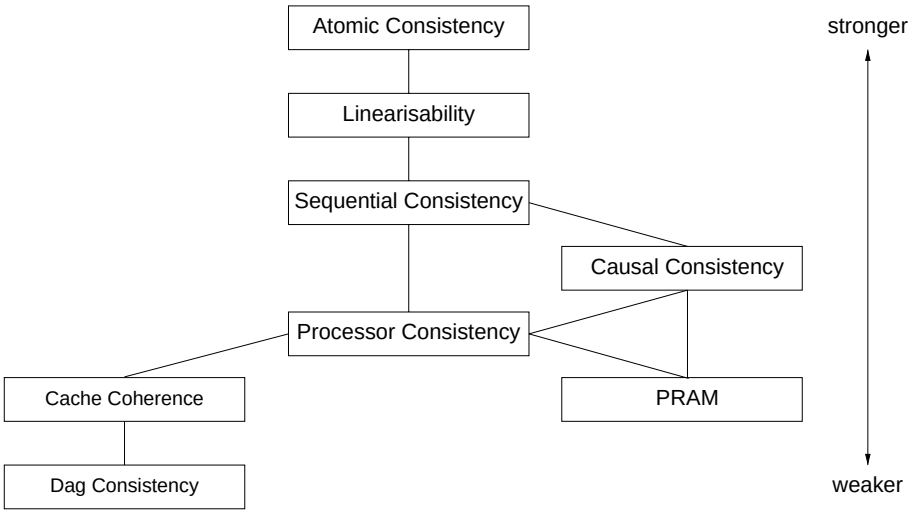



Figure 2.4: A categorisation of consistency models.

only updates of objects are communicated. However, in many cases one can adapt a consistency model to other operations on data objects.

### 2.2.1 Overview on consistency models in distributed shared memory

In the *strict consistency* or *atomic consistency* model any read to a shared object always returns the latest write value in real-time. This model is the strictest among all consistency models and used for performance evaluation, but it is not implementable as long as there is a positive communication delay between processes.

The *sequential consistency* model is based on the work by [Lamport 1979]. A sequence of operations on a set of shared objects is *sequentially consistent* if the operations can be serialised in some global order respecting the local order of operations.

A slightly stronger assumption is to request *linearisability* considering also whether in the real-time ordering of events the response of an operation precedes the call of another operation. In this case, the serialisation of events must also respect the order of those operations. Both consistency schemes are easy to implement on systems providing *imperfect clocks* and *atomic broadcast* [Birman and Joseph 1987b]. Imperfect clocks are clocks which run at the same rate as real time, but are not initially synchronised. They guarantee for the communication delay of a message to be in the range $[d-u, d]$ for some $0 < u \leq d$ where $d$ denotes the worst case deliver time, and $u$ the worst case uncertainty. Atomic broadcast

guarantees that all messages are delivered, and they arrive in the same order to all destinations. Further, the sent order of two messages originating from the same source is preserved.

Attiya and Welch [Attiya and Welch 1994] analyse lower and upper bounds on linearisability and sequential consistency with respect to imperfect clocks and introduce algorithms implementing these consistency models. In particular, it is shown that the cost for one read plus one write is at least $d$. For sequential consistency either read or write operations can be executed locally, while for linearisability always some communication cost is required.

*Causal consistency* requires that all writes that are potentially causally related must be seen in the same order by all processes. The model is a weakening of sequential consistency, i.e. there exists an execution which is causally consistent, but not sequentially consistent, however all executions satisfying sequential consistency are also causally consistent. In [Ahamad et al. 1995] an algorithm is introduced which implements causal consistency by using vector time stamps. Reads and writes are non-blocking operations, i.e. reads and writes can be executed immediately without requiring further communication cost. This is achieved by introducing a priority queue for incoming updates which are ordered according to the time stamps of messages. A time stamp consists of a vector with an entry for each process of the system. With respect to a process's local time stamp, a time stamp for an update is smaller if i) all entries with exception of the entry for the respective process are smaller, and ii) the entry for the process itself is equal to the respective value of the process's time stamp increased by one. An update of the queue is applied only if it is smaller than the local time stamp of a process.

*Pipelined RAM (PRAM) consistency* requires that all writes done by a single process are observed in the same order by all other processes. Writes performed by different processors may be ordered differently. PRAM is easy to implement by adding sequence numbers to each update. Similar to the implementation of causal consistency discussed above, a process maintains a FIFO queue, dispatching updates and adding respective sequence numbers. For incoming messages a process maintains a vector keeping track of the highest sequence number for each process. Incoming messages are stored in a priority queue. Updates are dequeued if the attached tag's value is one more than the value in the local vector. Obviously, PRAM is a weakening of causal consistency. As for causal consistency, all updates can be performed locally, but PRAM can achieve a better bit complexity for the tags associated with each update.

*Coherence* or *local consistency* weakens the sequential consistency by only requiring the order of operations to respect the local order of operations for each process. The weakening is often exploited by a protocol guaranteeing sequential consistency on a smaller entity, for instance a page containing objects.

*Processor consistency* requires an ordering of writes respecting both PRAM and coherence. Writes done by a single process must be observed in the same order by all processes. Writes performed by different processes may occur in different order as long as they do not have the same location.

The definitions so far look at consistency defined by processes acting on memory. An alternative approach is to look from the perspective of the computation [Frigo and Luchangco 1998]. In many practical scenarios it is only important that the shared objects are consistent at certain points in an application. *Weak consistency* [Dubois et al. 1986] introduces in addition to ordinary read and write accesses a special synchronisation access. All synchronisation accesses are sequentially consistent. A synchronisation access may perform if all previous read or write accesses have performed. A read or write access may perform if all previous synchronisation accesses have performed.

*Release Consistency* [Gharachorloo et al. 1990] extends the weak consistency model by dividing the synchronisation access into two primitives called *acquire* and *release* which are required to perform with respect to processor consistency. Read and write accesses may only perform if all previous acquire accesses have performed. A release access may perform if all previous read and write accesses have performed.

*DAG consistency* [Blumofe et al. 1996] considers how a multi-threaded computation affects the result of a computation on shared memory. Hereby, each thread must see the result of updates to shared locations with respect to a serial execution of the invocation and termination of threads, forming a directed acyclic graph. DAG consistency can be implemented by the BACKER [Blumofe et al. 1996] algorithm and is used in CILK [Blumofe et al. 1995], a multi-threaded programming language.

### 2.2.2 Consistency in collaborative environments

Most collaborative environments which implement a consistency model require a total ordering among all operations which respects also the local order of events for each process. This corresponds to sequential consistency. In order to improve the number of objects which can be supported, a weaker consistency model like causal consistency can be of interest. Weaker consistency models allow access to objects at lower communication cost. On the other side, for the weaker consistency models shown in Figure 2.4 it can take a long time until an update which was performed by a process locally can actually be observed by another process. This, however, can be of importance in collaborative environments where the real-time requirement implies that the consistency model should also provide a notion of the freshness with respect to an update. In this case, a mixed consistency model like weak con-

sistency or release consistency could be of interest where synchronisation happens only at special synchronisation points. Otherwise the implementation can focus on propagating updates fast within the world. So far little is known about consistency models which are suitable for collaborative environments. However, collaborative environments which are used to build new applications need to provide a way to reason on the correctness of an application. Defining a consistency model can give a better insight on how changes performed by users will affect the state of the world. Most previous work has dealt with techniques to resolve inconsistencies as dead reckoning and adding *lag* [Mauve 2000] to the communication, i.e. each update is delayed before it may perform in order to detect potential inconsistencies.

## 2.3 Publish/subscribe

As discussed in the context of CVEs (cf. Section 2.1) a collaborative environment may consist of a large number of processes and objects. Coordinating all interactions of the world with respect to a consistency model as discussed in Section 2.2 easily results in performance problems. Instead processes may only need to preserve a small subset of objects consistent. This subset of objects may however change dynamically. The communication model of a collaborative environment needs to support processes in expressing their interest and in return be able to be aware of interesting objects. Moreover, a process needs to be able to perform and receive updates relying on some delivery guarantee.

Publish/subscribe provide an event-based communication paradigm which allows processes to express their interest in form of subscriptions. After successful subscription, processes receive all published events in the system corresponding to their interests. In the context of collaborative environment an event can correspond to updates of an object or the discovery of a relevant object. A process performing an update of an event is called a *publisher*. Publish/subscribe systems support decoupling of space between publisher and subscriber, i.e. publisher and subscriber do not need to have any information about each other. This makes publish/subscribe an interesting communication paradigm for designing collaborative environments.

Various ways of expressing interest in publish/subscribe systems have evolved (cf. [Eugster et al. 2003] for an overview). Most commonly the focus is on *topic-based* and *content-based* publish/subscribe. In *topic-based* publish/subscribe event channels correspond to topics. A process can express its interest by subscribing to a set of topics and after subscription only receive information about these topics. Often topics form a natural containment relation. Topic hierarchies are an acyclic, directed graph capturing the containment relation between events. Each vertex of the graph corresponds to a topic name. After subscribing to a topic, say $A$, in a

topic hierarchy a process will also receive events corresponding to topics $B$ for which there exists a path starting in $A$ and ending in $B$. This way a subscription to a high level topic also gives the subscriber a higher load of events to be processed.

Topics can be used in collaborative environments to specify the region of interest and the names of objects. Hierarchies can also model the relationship between objects. For example in a graphical representation the level of detail perceived can be related to the level in the topic hierarchy a process subscribed. Also the frequency of events received can be controlled by publishing to subtopics in the hierarchy.

A lot of work has dealt with increasing the expressiveness of event notification services. Instead of associating each object with a topic, a process may be interested in events with special properties over a large set of objects. As an alternative way of describing each possible subset of properties by subtopics, events can carry attributes describing the properties of its content. *Content-based* publish/subscribe allows to express the interest of a subscriber by using filters. The subscriber will only receive those events whose property matches the filter. For content-based publish/subscribe programmers have more flexibility in expressing interest. However, this flexibility comes at higher communication cost or requires a more centralised flow of information.

The literature discusses many solutions for implementing publish/subscribe systems. Centralised solutions like JEDI [Cugola et al. 2001] for topic-based and SIENA [Carzaniga et al. 2001] for content-based publish/subscribe distinguish between clients and brokers where brokers provide to clients event channels depending on their interests and manage subscriptions and unsubscriptions. However, there are differences in how brokers themselves are organised. While JEDI considers a hierarchical organisation of brokers, SIENA considers a peer-to-peer organisation among the brokers. Scalable decentralised solutions (e.g. Scribe [Rowstron et al. 2001] and Data-Aware Multicast [Baehni et al. 2004]) using structured and unstructured peer-to-peer systems are discussed in Section 2.5. In general the issues involved in the implementation of topic-based publish/subscribe are close to the concept of group communication as discussed in Section 2.4. In fact, group communication can be used as a fundamental building block in implementing publish/subscribe systems. The main difference to the publish/subscribe paradigm is that process groups are formed corresponding to a single interest. Subscribing to multiple topics corresponds to subscribing to multiple process groups.

## 2.4 Deterministic group communication

So far we have discussed communication models supporting event-based interactions between processes. A fundamental building block is the dissemination of events according to an interest. *Group communication* provides an abstraction to many processes communicating on events so that processes which share a common interest can receive events disseminated by other processes sharing the same interest, while processes which do not share the same interest are not involved in the communication process. Processes sharing a common interest are said to form a *group*. A membership service allows processes to join and leave the group dynamically and every group member is allowed to communicate with other processes by using an event dissemination service.

One distinguishes among three different ways of how processes in a group communicate:

- *unicast communication* considers a peer-to-peer communication where a single process informs another process about an event,
- *multicast communication* considers a single process informing multiple processes about the same event,
- *many-to-many communication* considers multiple processes to multicast an event in a group of processes.

Often multimedia streaming uses an organisation which is restricted to multicast communication, but does not support many-to-many communication. Multiple processes can join a group and receive the respective service offered by a single process. Collaborative environments are not restricted to objects with these properties. They support also updates on objects to be sent from several sources corresponding to changes carried out interactively by multiple processes. This type of communication corresponds to a many-to-many communication model where all or multiple processes of the group can inform anybody else.

An important property of protocols supporting multicasting of events is the reliability criteria provided by the protocol. Formally, a protocol is said to be *reliable* if it can guarantee for an event to inform either all group members or none. Reliable communication also requires one to consider failures of processes or communication links. In this case, the reliability is expressed by requiring the protocol to inform all non-faulty processes or none about any event. Sometimes reliable communication is also associated with ordering guarantees like total ordering, i.e. all processes observe all events in the system in the same order. A group communication protocol which supports total ordering of events is also called *atomic*. Often

atomic multicasting is expressed together with further ordering like FIFO (all operations by a process are observed in the same order) or causal ordering of events. The ordering can be exploited as discussed before in distributed shared memory (see Section 2.2).

[Birman and Joseph 1987a] have introduced a scheme for reliable group communication called *virtual synchrony* which was the basis for the successful ISIS group communication system. The system can support various guarantees regarding the order of events, which hold in the occurrence of failures. On the other hand, a limitation of providing this strong reliability model is the scalability of the approach. Interestingly ISIS was used to support group management in DIVE (cf. Section 2.1). More recent approaches, also often referred to as reliable approaches, deal with various kinds of faults, but in many cases do not provide the full end-to-end reliability, which is essential in the definition of reliability.

An overview of these approaches is presented in [Levine and Garcia-Luna-Aceves 1998]. In order to support reliability, protocols have to detect when an event was not received at a destination. This is achieved by sending acknowledgements or negative acknowledgements. *Sender initiated* protocols require that the source of the multicast has to receive from all receivers an acknowledgement (ACK) before it can release a message from its buffer. In case a message was not received or an error occurred during the transmission, the sender can retransmit the same message to the destination. The problem with this approach is scalability. In such protocols the sender needs to process all acknowledgements. This problem is often referred to as the *acknowledgement implosion* problem. Moreover, within this approach the sender has to know all group members.

In order to reduce the number of acknowledgements that need to be sent and processed, *receiver initiated* protocols like SRM [Floyd et al. 1997] operate with negative acknowledgements (NACKs). If a process observes that a message was lost it multicasts the NACK to all members of the group. Any member can answer by multicasting a repair message with the requested data. Since multiple NACKs could cause multiple multicasts of the same message, SRM uses timers before sending or responding to a NACK. If a process, waiting to send a NACK with respect to a message, observes a NACK to the same message, it will skip its own NACK message. Similarly, a process ready to answer a request will cancel its repair message if it observes a corresponding repair message of another process. Clearly, a good choice of parameters for timers is crucial for the protocol to avoid an implosion of NACKs. If the number of participants is large and the communication delays vary a lot, it is hard to obtain a good choice for timers, which also limits the scalability of the protocol. Another problem with the described approach is that the source ideally needs to keep the message in its buffer forever because it can never be sure that a process may require its retransmission.

Memory requirements can be overcome by using a hierarchical organisation like RMTP [Lin and Paul 1996]. The group is divided into several subgroups, of which a local coordinator is in charge to acknowledge successful delivery to the source or another coordinator in the hierarchy. In this way one can avoid the source processing too many ACKs or NACKs. The method requires one to compute a tree structure in which the source has no knowledge of all members. If coordinators fail, non-faulty processes may not receive a message and thus violate the reliability criteria. Peaks in the communication traffic can overload the coordinators that will become hotspots in the multicast tree. Especially on a larger scale, hierarchical approaches are likely to suffer from perturbed network traffic. Moreover, the maintenance of the multicast tree by subgroups joining and leaving adds an additional burden for the scalability of these protocols.

## 2.5 Peer-to-peer membership and dissemination

Whereas the aforementioned deterministic approaches (cf. Section 2.4) originally aimed at providing a robust reliable service, *peer-to-peer (P2P) communication* is suited to manage large sets of entities in a decentralised way which is highly resilient to failures. However, the strong reliability for event dissemination is traded with the strength of the reliability guarantee which is often expressed in terms of probability for an operation to succeed.

Peer-to-peer systems provide a communication infrastructure called an *overlay* which is decoupled from the physical interconnection of processes although the physical interconnection may be considered when constructing the overlay. The infrastructure is designed to enforce a good distribution of the load of the work performed by the system, i.e. all processes may perform both as client and server. Similar to the group communication model, the infrastructure is maintained by processes performing join and leave operations. In order to join the structure it suffices to know an arbitrary member of the peer-to-peer system.

The literature distinguishes between *structured* and *unstructured* peer-to-peer systems. *Structured peer-to-peer systems* provide a lookup service which binds any given key to a location maintained by one process of the system. Any process of the system can efficiently route for a given key to the respective location. Peer-to-peer systems which provide this property are also said to implement a *distributed hash table (DHT)*. DHTs commonly use a technique, called uniform hashing [Karger et al. 1997], i.e. a known hash function is used by all processes. Typically the hash function is designed such that keys are well distributed among locations of the peer-to-peer system. If an adversary would select a pair of keys a collision is unlikely to happen. Many existing solutions choose a cryptographic hash function

as SHA1 [NIST 2002] which maps keys of bits to values of bits. DHTs can be powerful in constructing distributed services such as persistency management and publish/subscribe.

*Unstructured peer-to-peer systems* do not rely on a lookup service, but rather maintain a set of communication partners which may change dynamically over time. Typically, such systems are constructed application-specific, e.g. for implementing aggregation, large-scale data dissemination, or publish/subscribe.

In the following, an overview of related work of structured and unstructured peer-to-peer systems is presented with focus on membership management and large-scale event dissemination.

### 2.5.1 Structured peer-to-peer systems

Recent research has proposed a variety of structured peer-to-peer systems which are also referred to as DHTs. As a main characteristic, a DHT offers a lookup service to its members. Every process which joins the DHT is mapped to at least one location. When the membership performs stable, a lookup of a valid key is uniquely mapped to a single location, and the process performing this lookup can efficiently route to this location.

A typical application of a lookup service is an object location service as used in many file-sharing systems and in persistency management. Each object can be associated with a unique identifier, e.g. by using consistent hashing. Let $\Sigma$ and $I$ be alphabets and $\mathit{hash} : \Sigma^* \rightarrow I^l$ denote a hash function which is collision free in a cryptographic sense, i.e. it is difficult for an adversary to find $\omega_1 \in \Sigma^*$ and $\omega_2 \in \Sigma^*$ such that $\mathit{hash}(\omega_1) = \mathit{hash}(\omega_2)$. Then $\mathit{hash}(\text{object})$ determines the location where the object should be stored. Any process which knows the name of the object can use *hash* and perform a lookup to determine the location and perform operations such as read or modify on the object. Besides object location, the lookup service of a DHT can be used to support other services such as large-scale event dissemination and publish/subscribe with the help of application level multicast trees (cf. Section 2.5.2).

Almost all algorithms in the literature provide an efficient lookup service. Let $N$ denote the number of locations of a DHT, then a lookup can be implemented using $O(\log N)$ routing steps by using for each process a routing table of $O(\log N)$. For some algorithms such as CAN [Ratnasamy et al. 2001] space can be traded against time complexity as an algorithm parameter. In spite of sharing many similarities, algorithms often differ in how locality and failure resilience are supported. A good categorisation of DHT based algorithms is presented in [Gummadi et al. 2003] considering the geometry of DHTs and the effect of geometry on proximity and failure resilience. In the following, the ideas behind some representative

algorithms categorised according to their geometry are briefly introduced.

**Ring.** In Chord [Stoica et al. 2001] locations maintained by processes are aligned according to a cyclic predecessor successor relation. Each process in a Chord ring can identify its location by performing uniform hashing. Assume that $I^l = \mathbb{Z}_M$. Let $p$ denote a process and $l_p$ its location, then processes maintain a routing table of a maximum number $i \in \{1, \ldots, \log M\}$ in addition to its successor/predecessor pair locations closest to $l_p + 2^i \bmod M$. A lookup request of $x$ is forwarded to $l_p + 2^j \bmod M$ if there exists $k > 0$ such that $l_p + 2^j < x + M^k < l_p + 2^{j+1}$. Under the assumption of a uniform distribution, a lookup needs $O(\log N)$ hops to succeed. Chord also proposes a self-stabilisation protocol to deal with wrong routing table entries.

**Hypercube.** In CAN [Ratnasamy et al. 2001] a location and key are a $d$-dimensional vector mapped to a $d$-torus, i.e. $I^l = \mathbb{Z}_M^d$. The $d$-torus is divided into zones associated with a single location. A lookup is forwarded to the locations along the line between source and destination in the Cartesian space. Assume all locations are perfectly distributed. In this case each location is expected to have $2d$ neighbouring zones determining the space requirement of the routing information. Moreover, for each dimension the maximum path length is bounded by $n^{-d}$. Hence, a lookup involves at most $O(dn^{-d})$.

**Tree combined with ring.** In the aforementioned approaches locality was not a design issue. However, each hop in a peer-to-peer network can take a long distance in the underlying physical network. For example, in a geographical setting a path of a lookup may travel between two continents several times although the source and the destination of the lookup are close.

PRR-tress proposed in [Plaxton et al. 1997] were originally designed to minimise the cost of a lookup on a set of replicated objects. Although the focus of the work was not on providing a dynamic assignment of processes to locations in combination with a lookup service, PRR-trees have influenced the design of the routing table maintained in Pastry [Rowstron and Druschel 2001] and Tapestry [Zhao et al. 2004]. Let in the following $I = \{0, \ldots, b-1\}$. Both schemes use a prefix-based routing mechanism in which a lookup visits in every hop a location sharing a larger prefix with the key. Therefore, each location maintains for every prefix of length $i$ $b-1$ locations which differ for a prefix of length $i+1$, and this way the common prefix can be increased when forwarding a lookup request. A lookup is terminated if the location is closest to the key. In order to decide closeness, processes maintain also a set of children of closest locations. Some of the latter have larger value and

some are required to have lower value than the value of the maintained location – similar to the cyclic predecessor/successor relation of Chord.

For a prefix of length $i$ there are potentially many more than $b - 1$ possible locations. This freedom is used by Pastry in choosing locations according to best locality properties.

### 2.5.2 Event dissemination in structured peer-to-peer systems

Besides object location services, structured peer-to-peer systems are of interest for event dissemination systems such as group communication and publish/subscribe. In combination with formerly mentioned structured peer-to-peer systems there exist dissemination services such as the one described in [Ratnasamy et al. 2001; Rowstron et al. 2001], developed for CAN and Pastry, respectively. A common technique in supporting event dissemination services is the construction of *application-level multicast trees*. An application-level multicast tree is a tree embedded in the structure of the peer-to-peer system connecting a set of members forming a group. A typical issue in constructing and maintaining application-level multicast trees is support for self organisation, i.e. processes which are interested in receiving an event service are responsible for being included in the multicast tree. The only requirement is that a process needs to know the identifier of the multicast tree's root.

Given an application-level multicast tree it is easy to provide topic-based publish/subscribe. Each topic can be associated with a location e.g. by using consistent hashing. This location becomes the root of the tree and all published events with respect to the topic are forwarded via the root.

A way of constructing an application-level multicast tree, as it is proposed in Scribe [Rowstron et al. 2001], is to perform a lookup to the root of the application-level multicast tree. Let $r$ denote the root of an application-level multicast tree and $p$ denote a process which likes to join the application-level multicast tree formed by $r$. When performing a lookup to $r$, $p$ takes the first process on the path to $r$ as its father. No other processes on the path towards the root become involved. Routing towards the root of the multicast tree allows a decentralised self-organisation since in the occurrence of failures a process only needs to start another lookup towards the root. However, if the group size is small compared to the overall size of the peer-to-peer network than this approach is expected to lead to a fairly unbalanced tree where nodes close to the root are expected to have a fairly large number of children.

In the approach taken by Bayeux [Zhuang et al. 2001], $p$ first performs an ordinary lookup to $r$. Afterwards $r$ uses the multicast tree to perform a lookup to $p$. At the point where a process, say $q$, cannot route any closer to $p$, $p$ is inserted as

a child. This technique allows a better balance among processes, however the root of the multicast tree is involved in every join operation. Bayeux uses the idea of multiple root processes.

An alternative way of achieving a better balance and decreasing the involvement of processes in each join is to combine the approaches taken by Scribe and Bayeux by performing a reverse lookup from the first process on the path between $p$ and $r$.

A disadvantage of the presented solutions is that the selected root node is not necessarily interested in being a group member, but is the most involved node in maintaining the tree as well as in forwarding messages. Moreover, there is a fairness issue since processes which join the multicast tree at a lower level are more involved as forwarders than the majority of processes which are only consuming messages.

A way to address the latter problem is to split the multicast data into several items as suggested by SplitStream [Castro et al. 2003] and Bullet [Kostić et al. 2003]. Bullet considers bandwidth requirements of processes and tries to pair processes according to already delivered content. This is achieved by performing a protocol on top of an application-level multicast tree discovering possible pairing. SplitStream disseminates events using a forest of multicast trees instead of a single multicast tree. Before performing a multicast the data is split into $k$ items and propagated via $k$ sources. Dissemination using multiple multicast tree sources balances the work in forwarding messages if for the different multicast trees the set of processes performing the work in forwarding differs. This is likely because of the uniform distribution of locations typically achieved in structured peer-to-peer systems. Splitting of data can also be used to provide different services suited to the demands of a process. A process which is interested in obtaining information which requires need for higher bandwidth needs to subscribe to more multicast trees. This is especially useful for multimedia data where data may be transmitted using various levels of quality.

In cases were the size of data is small, splitting data could be used to achieve redundancy, but it is not useful to share the load while disseminating the data. Instead one may try to balance the load induced by sending a series of events, e.g. by performing random choices in order to select which of the multiple multicast sources to use for dissemination of an event.

### 2.5.3 Unstructured peer-to-peer systems for large-scale dissemination

Structured solutions to support event dissemination organise processes in some hierarchical form e.g. an application-level multicast tree. Although in a faultless execution a hierarchical organisation can give good performance, a failure close

to the top of the hierarchy can disconnect a large set of processes. To deal with failures we have to introduce some form of redundancy such as maintaining multiple application-level multicast trees. Moreover, the protocols require some form of continuous maintenance of data structures to achieve a good and fair balance of the work.

In the following, we consider unstructured solutions for event dissemination with good scalability guarantees and good resilience to failures. Instead of requiring the strong end-to-end reliability as provided by virtual synchrony (cf. Section 2.4), we look at reliability which guarantees the dissemination of an event to all non-faulty processes expressed with *high probability*, which is with probability $1 - O(n^{-c})$ for some positive constant $c$.

Unstructured solutions for event dissemination follow a flat communication model where each process maintains a possibly dynamic set of group members called its *view*. Consider the directed graph formed by inserting for each process $p$ an edge from $p$ to all members of $p$'s view, If the directed graph is connected then *flooding*, i.e. forwarding events to all neighbours, succeeds in disseminating an event to all processes of a group. Alternatively, when performing *gossiping*, communication happens with a subset of the communication partners of a view during a round. The number of communication partners, called the *fanout*, is a tuning parameter of gossiping protocols. The actual subset of communication partners is typically chosen uniformly at random from the whole view. In the context of event dissemination, communication can be used to either push an event, pull an event, or perform combined push and pull.

Gossiping was originally introduced into distributed computing in the context of data replication [Demers et al. 1987]. Since the introduction of bimodal multicast [Birman et al. 1999] gossiping has received a lot of attention in group communication.

The gossiping paradigm has been used under different assumptions depending on the underlying network. While in a wired network (which we assume in the context of this thesis) the view is decoupled from the physical interconnection forming an overlay network, for sensor networks the view is formed by the direct communication partners in the range of the sensor. For sensor networks, gossiping is a way of reducing the overall work performed during a dissemination.

In wired networks, the view is an additional tuning parameter to determine the reliability of a dissemination. A random selection of processes inside a view and communication partners during a round distributes the communication load evenly between the processes of the group. In contrast to hierarchical approaches where a failure of a process or link can affect a huge part of group members, a failure of a process or link in gossiping protocols has only a small effect on the overall statistical reliability provided.

The statistical reliability of gossiping depends on the rules used for achieving event dissemination. This comprises not only the view and the way communication partners are selected during a round, but also the decision criteria for propagating an event and terminating the dissemination of an event. Often the mathematical analysis of epidemics and rumour spreading is applied to analyse the statistical reliability of gossiping protocols. Hereby, an event is associated with a disease or a rumour with the goal to infect as many processes as possible (see also Section 3.2).

Analytical approaches often assume that the view of a process consists of all group members. In the setting of group communication the view corresponds to the membership information stored at a process. Storing information about all group members limits scalability supported by a protocol. Therefore, it is important to initialise views in a decentralised manner such that a view contains only a reasonable small subset of the group members. The membership schemes of *lpbcast* [Eugster et al. 2001a] and *SCAMP* [Ganesh et al. 2001] are two representative approaches. In lpbcast the size of the view is a system parameter and the view changes dynamically by exchanging continuously group members in each communication step of the gossip protocol. This shuffling technique is intended to allow the selection of processes from the view as if the selection took place uniformly at random over all group members. Let $n$ denote the number of processes inside a group and $C$ a constant, then SCAMP automatically adapts the average size of its view approximately to $\log n + C$. However, if the group membership remains static the view also remains the same. In [Kermarrec et al. 2003] an analysis shows that a fanout of $O(\log n)$ suffices to inform all processes w.h.p.

The scalability properties have been successfully demonstrated for collaborative multi-player applications. NEEM [Pereira et al. 2003] combines gossiping with semantic knowledge about events to reduce the number of actively disseminated events. The protocol has been tested with a fairly large number of spectators in a distributed flight simulation.

Gossip groups have also been suggested to implement publish/subscribe. Data Aware Multicast [Baehni et al. 2004] proposes a protocol implementing topic-based publish/subscribe by also supporting topic hierarchies. The delivery guarantee for a subscriber is related to the deliver guarantee provided by SCAMP.

## 2.6 Contributions

In the following chapters we look at various algorithmic aspects towards building large-scale collaborative environments. The focus is on failure resilient and decentralised peer-to-peer algorithms supporting large-scale event dissemination and interest management. For achieving event dissemination we consider structured

and unstructured peer-to-peer systems using gossip-based event dissemination and application-level multicast trees. For gossip-based event dissemination this thesis presents:

- an analysis of gossip-based event dissemination,
- an evaluation of resource management,
- an evaluation of termination strategies,
- comparison of dissemination strategies,
- experimental evaluation using simulation as well as network experiments,
- consistency management supporting causal ordering,
- bootstrapping of membership in order to provide random choices.

In the context of publish/subscribe systems we consider application-level multicast trees as well as gossiping. The main focus here is to support

- a fair work distribution according to processes interest,
- well balanced trees.

**Analysis of gossiping.** Chapter 3 proposes a method for analysing gossip-based group communication systems in order to provide an end-to-end reliability guarantee expressed with high probability. Typically, the analysis of gossiping is based on an approximation modelling an epidemic using differential equations or by using random graphs. In this thesis, gossiping is analysed by modelling the dissemination as a balls-and-bins game. Based on this method, a dissemination scheme has been proposed which can be combined with lightweight membership algorithms. The dissemination scheme has been evaluated in combination with lightweight resource management and consistency management.

**Resource management.** Limitations on the resources have an effect on the delivery guarantee of gossip-based protocols. Apart from membership information stored at a process, gossip protocols keep a history for events which have been already delivered. This avoids multiple delivery of the same event to the application, but is also used by some protocols as a way of termination. Since the number of events can be large, the size of the buffer is bounded. Chapter 4 provides an analysis for estimating a safe buffer size so that there is a low probability for events being delivered multiple times. In a dynamic system the analysis can help to adapt the

buffer size based on observable parameters. Besides the size of the buffer, the order in which events are inserted and removed are also important and are evaluated with different dissemination schemes.

**Termination strategies.** Buffer management also has an impact on the termination strategy. Often the history buffer is also used as a criterion to terminate the dissemination of events. Chapter 4 shows that even for almost safe buffer sizes this termination policy can lead to low reliability and a large number of multiple deliveries. Since some events may propagate a long time, it is suggested to explicitly terminate the dissemination by considering the number of hops an event performed in the dissemination system.

**Comparison of dissemination.** In the traditional analysis of rumour spreading (cf. [Pittel 1987]), a process communicates with only one process at a time. Each process needs to gossip about an event received for as many rounds as it takes until every process of the group has received the event w.h.p. This dissemination strategy may limit the amount of events which can be disseminated at the same time if many concurrent events exist in the system.

Approaches like *pbcast* [Birman et al. 1999] and *lpbcast* [Eugster et al. 2001a] suggest instead to disseminate an event only the first time it is observed by providing at the same time a higher fanout. However, in order to provide guarantee w.h.p. the fanout needs to be at least in $\Omega(\log n)$.

According to the *balls and bins dissemination* proposed in Chapter 3 a process propagates an event whenever it receives it unless the event has travelled the maximum number of allowed hops.

Chapter 4 evaluates both dissemination schemes in combination with a framework which supports the dissemination of concurrent events. Balls and bins dissemination has been shown to provide a better message stability by using the same amount of traffic and requiring a low number of multiple deliveries.

**Consistency management.** Gossip-based protocols per se do not provide any ordering guarantees. Chapter 5 proposes a lightweight dynamic consistency management which can be combined with lightweight dissemination algorithms. The proposed consistency management implements an optimistic causal ordering of messages. The key to provide scalability is to limit the number of concurrent updater by providing a dynamic, decentralised and fault-tolerant cluster management protocol. The delivery guarantee of gossip-based dissemination gives good performance results in combination with the causal ordering protocol since only a small amount of communication deals with recovery messages and the likelihood of overloading

a process is reduced. The evaluation also shows that the ordering scheme adds only a low overhead.

**Experimental evaluation.** In order to support the evaluation of dissemination schemes presented in this thesis, a simulation as well as a group communication framework have been developed. The simulation considers a synchronous execution model. In each round all processes receive incoming messages which were sent in the previous round, perform a computation, and send messages to be delivered in the next round. The communication assumes an overlay in which a process can exchange information with any other process it is aware of. On top of this model, all discussed variations for achieving dissemination and buffer management discussed in Chapter 4 were implemented in combination with a lightweight dynamic membership algorithm similar to lpbcast. The simulation has been used to follow the lifetime of events in the dissemination system and could reveal the importance of event termination.

In order to evaluate the behaviour in a networked environment an object-oriented group communication framework has been implemented supporting multiple multi-threaded layers. The basic layer, the group communication interface, provides basic point-to-point communication by using UDP or TCP as the transport protocol. The group communication layer is a group object which implements membership services as well as multicast on top of the basic layer. Within the framework it is possible to maintain multiple group objects using different implementations at the same time. The group interface will propagate events to the correct group object. It is possible to establish several layers on top of a group object.

The experiments in Chapter 5 use the balls-and-bins dissemination in combination with the general group communication framework described in Chapter 3 and a lightweight dynamic membership protocol. The causal ordering algorithm is implemented as a layer which can be also used in combination with other group objects. The network experiments served to measure the performance of the dissemination system regarding throughput, latency, message size, and the impact of causal event ordering. Moreover, the experiment revealed several issues in tuning the dissemination algorithm: using TCP as a transport does not show significantly worse performance to UDP. Moreover, in combination with timeouts TCP provides feedback on how to set parameters as to the duration of a gossip round. If processes perform gossiping by using approximately the same round duration then the protocol gives a higher reliability.

**Bootstrapping of membership.** An assumption made for pbcast and balls and bins dissemination is that a process can select $K$ random processes from its view

as if the processes have been chosen uniformly at random among all processes of the system. In an unstructured peer-to-peer network one can use the shuffling technique of views proposed by lpbcast to obtain an approximate random view. However, it is not clear whether an algorithm using this approximation is required to maintain a larger fanout and how large the view needs to be maintained.

In Chapter 6 the bootstrap problem is approached in a structured peer-to-peer system by using a technique called random routing. This allows one to maintain a view as small as the required fanout for the dissemination.

**Fair work distribution.** Achieving publish/subscribe in peer-to-peer systems by using application-level multicast trees may distribute the work in forwarding unevenly among processes and also involve processes which do not share an interest in the event they disseminate. Chapter 6 proposes a topic-aware peer-to-peer membership which allows one to route between processes sharing the same interest without involving any processes which do not share the same interest.

**Balanced trees.** Application-level multicast trees as discussed in Section 2.5.2 often are not well balanced or the maintenance for balancing a tree frequently involves the root. By using random routing in a role-based distributed hash table, Chapter 6 shows how one can construct a forest of well balanced application-level multicast trees.

# Three

# Simple Gossiping with Balls and Bins

Boris Koldehofe

**Abstract**

Recent research suggests decentralised probabilistic protocols on support for multipeer communication. The protocols scale well and impose an even load on the system. They provide statistical guarantees for the reliability, i.e. an information sent from an arbitrary source will reach all its destinations. Analysing the reliability is based on modelling the propagation of events as an epidemic process often referred to as gossiping or rumour spreading.

This work[1] provides a new method for analysing such protocols, by representing the propagation of information as a balls-and-bins game. The method gives a simple relation between the number of hops a gossip message is propagated and the reliability provided. This way it can facilitate the analysis of the multiple delivery problem i.e. to prevent multiple deliveries of the same message to the application layer. By introducing a new protocol it is shown how existing approaches can be adapted to the balls-and-bins approach. Furthermore, the proposed method is applied to analyse the performance of this protocol.

[1] A version of this paper has been published in the proceedings of the 6th International Conference on Principles of Distributed Sytems (OPODIS'02)

## 3.1 Introduction

*Multipeer communication* applies to all scenarios where multiple senders and multiple receivers, associated with physically distributed processes, communicate information of common interest. Processes sharing a common interest form a group, in which they exchange messages on information. Every piece of information is embedded in an *event*, which processes deliver to the application layer of the protocol. Supporting collaboration for large groups of users can be seen as one application relying on multipeer communication. Group members require a fast and reliable propagation of events within such an environment. Moreover, the group needs mechanisms that allow an efficient handling of membership, i.e. ongoing joining and leaving of group members should not add a significant overhead to the group communication.

General aspects of interest in group communication concern the *reliability* of the communication and the *scalability* of the protocol. Reliable group communication protocols provide guarantees that group events reach all their destinations. They may also be required to give ordering guarantees, which are needed in many cases where shared data objects are modified with respect to some consistency model. Reliable group communication protocols must also be robust against faults and bursts in the communication traffic. Apart from providing guarantees that an event will be delivered, some models are also concerned with prohibiting multiple deliveries of events to the application. Protocols that lack this property are said to suffer from the *multiple delivery* problem. The *scalability* of a group communication protocol is determined by how many group members can be supported and which reliability mechanisms are provided.

A common categorisation used in [Wittmann and Zitterbart 1999] divides protocols supporting group communication services in three different classes.

*Reliable group services* guarantee that if an event is delivered to one destination it will eventually be delivered to all operating destinations. Further, ordering guarantees for events are provided. However, these requirements imply a high synchronisation overhead for such protocols. The high communication cost let those protocols scale only to a low number of processes.

*Unreliable group services* focus on best effort strategies which scale well. Under ideal conditions some protocols even give reliability guarantees. However, under perturbed conditions the protocols can show unpredictable behaviour and thus cannot be used for time critical applications.

The focus of this paper is on group services that provide *predictable reliability guarantees* which scale well and hold even under perturbed conditions. In particular, we focus on those protocols which are based on *gossiping* or *rumour spreading* where processes exchange in every round messages on events according to a prob-

abilistic scheme. By processes communicating with random destinations, hot spots are avoided, compared to the typically hierarchical flow of events within reliable group services. In the context of gossiping protocols, an event is also called a *rumour*. This notation is commonly used in the analysis of gossiping protocols.

**Organisation of this paper.** Section 3.2 introduces the *epidemic model*, which is the mathematical foundation for most existing gossiping protocols, followed by a presentation of known theoretical results for this model. Further, an overview is presented on related protocols based on gossiping in the area of multipeer communication. Some of the protocols use an approximation of the theoretical results presented. In the next sections it is shown how to modify the epidemic model so that the balls-and-bins analysis can be applied. A definition of a *balls-and-bins-compliant gossiping protocol* is introduced in Section 3.3, where the balls-and-bins approach is described. In Section 3.4 a new protocol with this property is described. The framework of the protocol is based on recently proposed protocols for this problem, such as *pbcast* [Birman et al. 1999] and *lpbcast* [Eugster et al. 2001a]. Moreover, Section 3.5 presents an analysis based on the balls-and-bins model. Hereby, the reliability of the protocol is determined as a statement with high probability that all processes will receive an event propagated within such a system after gossiping this event for a fixed number of rounds. The event can originate from any arbitrary process. Section 3.6 discusses the applicability of the given approach in the context of the *multiple delivery problem*.

**Remarks on the notation.** Throughout the paper $\log$ and $\ln$ denote the logarithms in basis of 2 and basis of $e$, respectively. For any other basis, say $a$, the logarithmic function is expressed as $\log_a$. In the analysis the expression "with high probability", i.e. with probability $1 - O(n^{-k})$ for a constant $k > 0$, is abbreviated by writing w.h.p.

## 3.2 Related work and models

**The epidemic model.** The idea of applying the mathematical theory of general epidemics [Bailey 1975] to rumour spreading in distributed systems was first adapted in [Demers et al. 1987]. In this approach, the protocol models the spreading of a rumour as an epidemic process, with the purpose to infect as many other processes as possible. Processes can be

1. *susceptible* to the rumour, i.e. they have not received the rumour yet,

2. *infected* by the rumour, i.e. the rumour is received and processes are spreading this rumour to other processes,

3. *resistant*, i.e the process has received the rumour, but does not participate in the propagation of the rumour.

Let $s, i, r$ denote the fraction of susceptible, infected and resistant processes at time $t$ such that $s+i+r = 1$. Assume that within the time fraction $\Delta t$ the infectious rate denoted by $\alpha$ is proportional to the number of infected processes and to the number of susceptible processes, respectively. Then, the fraction of new infection is given by $\alpha i s \Delta t$. On the other hand, the constant removal rate denoted by $\beta$ needs to be considered, such that the fraction of new resistant processes is given by $\beta i \Delta t$. The resulting differential equations are given by

$$\begin{aligned} \frac{\delta s}{\delta t} &= -\alpha s i \\ \frac{\delta i}{\delta t} &= \alpha s i - \beta i \\ \frac{\delta r}{\delta t} &= \beta i. \end{aligned}$$

By $i(t) = i(t(s))$ we can express $i$ as a function of $s$. Hence,

$$\begin{aligned} \frac{\delta i}{\delta s} &= \frac{\alpha s i - \beta i}{-\alpha s i} \\ &= -1 + \frac{\beta}{\alpha}\frac{1}{s}. \end{aligned}$$

Let $s_0$ denote the initial fraction of susceptible processes while $i_0$ denotes the initial fraction of infected processes. The anti-derivative for $-1 + \frac{\beta}{\alpha}\frac{1}{s}$ is given by $-s + \frac{\beta}{\alpha}\ln s + C$. Applying $i_0 = -s_0 + \frac{\beta}{\alpha}\ln s_0 + C$, we obtain

$$i(s) = i_0 + s_0 - s + \frac{\beta}{\alpha}\ln\frac{s}{s_0}.$$

The maximum number of infected processes at the same time is given by $i(\frac{\beta}{\alpha})$. The term $\frac{\beta}{\alpha}$ also represents the threshold for the ratio of susceptible processes needed to start an epidemic. The epidemic process terminates if $i(s) = 0$, which is the case if

$$s = s_0\, e^{\frac{\alpha}{\beta}(s - i_0 - s_0)}.$$

Initially, only one process knows about the rumour and $s_0 + i_0 = 1$. Thus for large groups,

$$s \approx e^{-\frac{\alpha}{\beta}(1-s)}.$$

**Theoretical bounds.** Theoretical work presents results on how to bound the number of rounds and the number of transmissions needed to infect every process with high probability.

B. Pittel [Pittel 1987] analyses a simplified model, in which processes never loose interest, i.e. the removal rate is zero. Let $n$ denote the group size and assume that an infected process informs only one other process in a round. In probability the number of rounds needed to infect all group members is determined by

$$\log n + \log_{10} n + O(1) \quad \text{as} \quad n \to \infty$$

Karp et al. [Karp et al. 2000] introduce a combined push and pull scheme in which processes cease to push a rumour when the expansion of the rumour has exceeded a certain threshold. Since the basic scheme is based on an exact estimation on the number of informed processes, an algorithm for terminating the spreading of the rumour is introduced. The complete scheme is address-oblivious, i.e. it does not depend on the state of neighbour processes. The number of transmissions of the scheme is $O(n \log \log n)$. Further, it is shown that any such scheme needs at least $\Omega(n \log \log n)$ transmissions independent of the number of rounds.

**Rumour spreading in group communication.** Rumour spreading (gossiping) has received a lot of attention in group communication, especially after the introduction of *pbcast* [Birman et al. 1999]. The arguments in favour of rumour spreading are that those protocols introduce an even load on the system and are robust against bursts in the network traffic. Further, they usually provide mechanisms that support fault tolerance and scale better than deterministic approaches. The reliability is based on an approximation of the epidemic model, in which each event associated with a rumour will be spread to all other processes with high probability. However, approaches like *pbcast* rely on global knowledge of all group members and often on a centralised organisation of membership. This way the organisation of a dynamic membership, i.e. processes can join a group or leave a group, requires a lot of synchronisation overhead and a lot of memory resources affecting the scalability. In contrast, recent work [Ganesh et al. 2001; Eugster et al. 2001a] suggests protocols that support decentralised handling of membership where processes only have a partial view on the system.

In *lightweight probabilistic broadcast* (*lpbcast*) [Eugster et al. 2001a] the size of the view, i.e. the set of group members known to a process, as well as the size of buffers storing recent events, are statically fixed. When propagating an event, a process communicates only with a fixed number of processes randomly chosen from the local view. The reliability is derived by an approximation of the theoretical result presented in [Pittel 1987]. With respect to the approach introduced in this

paper, it provides the interesting property that the view converges to a situation where it is as if members of the local view have been selected uniformly randomly among all group members.

Similar to *lpbcast SCAMP* [Ganesh et al. 2001] presents a scheme supporting decentralised group services. However, instead of using a fixed view size, the membership scheme takes care that the view automatically adapts depending on the group size. The local view size converges to approximately $\log n + C + 1$ where $n$ denotes the group size and $C$ is a constant determining the fault tolerant behaviour of the protocol. The reliability of the event transmission is related to the connectivity in the random graph model and is analysed in [Kermarrec et al. 2003].

## 3.3 The balls-and-bins model

The occupancy problem considers a random allocation of $m$ indistinguishable objects denoted by balls on $n$ destinations denoted as bins. Many problems in computer science can be modelled using the occupancy problem. For example, in distributed computing the random allocation problem has received a lot of attention. One of the many issues in the analysis is to bound the maximum number of balls that can fall into a single bin.

In the setting of spreading a rumour among $n$ processes, we associate each process with one bin. Let $H_n = \sum_{k=1}^{n} \frac{1}{k}$ denote the $n$th harmonic number. If the rumour is carried within balls which are randomly placed into bins, it is known that $nH_n$ balls are expected to be placed until every process has received at least one ball. To obtain a statement with high probability, i.e. with probability at least $1 - n^{-k}$ for some positive constant $k$, we look at the random variable $X$ denoting the number of empty bins and the respective expectation value

$$\mathbf{E}[X] \;=\; n\left(1 - \frac{1}{n}\right)^{m}.$$

Since for $n \geq 1$ the inequality $\left(1 - \frac{1}{n}\right)^{n} \leq e^{-1}$ holds, the expectation value can be bounded by

$$\mathbf{E}[X] \;\leq\; ne^{-m/n}.$$

Indeed, we can also write $\mathbf{E}[X] \sim ne^{-m/n}$. By choosing $m$ and $k$ such that $m = cn \log n$ and $k \geq c + 1$ for a constant $c > 1$, one can derive $\mathbf{E}[X] \leq n^{-k}$. From Markovs inequality it is known that $\mathbf{Pr}[X \geq 1] \leq \mathbf{E}[X]$, which gives

$$\mathbf{Pr}[X = 0] \;\geq\; 1 - n^{-k}.$$

This reasoning, summarised in theorem 3.3.1, is the basis for the approach of analysing the rumour spreading as a balls and bins game.

**Theorem 3.3.1** *Let $c > 1$ be a constant and let $m$ denote the number of balls that are placed into $n$ bins chosen uniformly and independently at random. If $m \geq cn \log n$ then every bin contains at least one ball with high probability.*

Now, the balls-and-bins model is introduced by defining what a *balls-and-bins-compliant gossiping protocol* is. Hereby, it is required to send a rumour to $O(n \log n)$ destinations which are uniformly and independently chosen at random. Since balls representing a rumour can only be placed by processes that have received the rumour before and initially only one process knows the rumour, it is allowed that the rumour is propagated in multiple rounds. This way, processes which received balls in previous rounds can create themselves new balls. As long as balls are placed in each round independently of the round the ball was created, the protocol can terminate after overall $O(n \log n)$ balls have been placed. The resulting scheme of a balls-and-bins-compliant protocol, as defined in definition 3.3.1, is illustrated in Figure 3.1.

**Definition 3.3.1** *A protocol for spreading a rumour is said to be balls-and-bins-compliant if for some fixed integer $r > 0$ and a constant $c > 1$*

- *the protocol provides a scheme that creates $m_i > 0$ balls associated with the rumour in consecutive rounds $1 \leq i \leq r$,*
- *the protocol guarantees that $\sum_{i=1}^{r} m_i \geq cn \log n$,*
- *the destination of each ball is chosen uniformly at random and independently of the destination of the other balls,*
- *and the protocol terminates after $r$ rounds, i.e. $m_i = 0$ for all $i > r$.*

**Corollary 3.3.1** *Any balls-and-bins-compliant protocol terminating after $r$ rounds informs all processes about a rumour with high probability.*

With respect to a rumour and a set of balls associated with the rumour, in this scheme a process is

- *susceptible* when no ball has fallen into the bin associated with the process for the first time,
- *infectious* whenever it creates balls that are placed randomly into some bins,

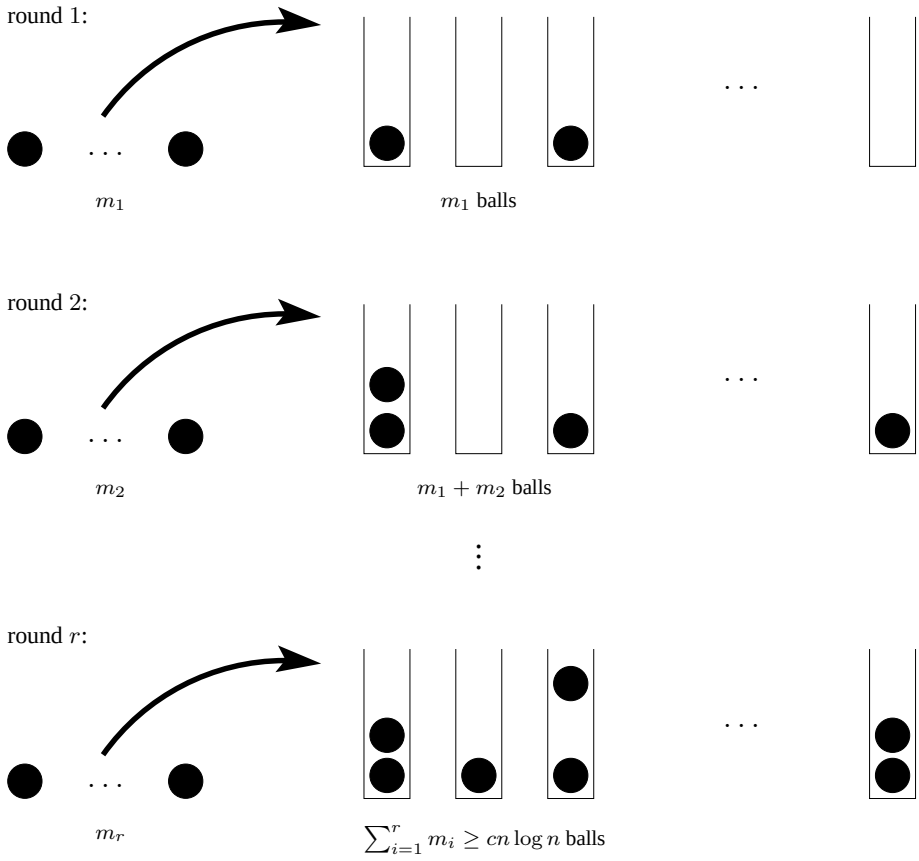



Figure 3.1: A scheme for spreading a rumour that terminates after $r$ rounds and guarantees that at least $cn \log n$ balls where placed into random bins.

- and *resistant* when the process knows about the rumour, but does not create balls.

Compared to the state description of the epidemic model shown, the description based on balls and bins is more flexible regarding the state changes of a process. The balls-and-bins model allows processes to change several times between being infectious and being resistant without affecting the underlying analysis.

In the next section a scheme implementing a balls-and-bins-compliant protocol is introduced. Further, the number of rounds the protocol needs to terminate is analysed for a fixed fan-out value.

## 3.4 A balls-and-bins-compliant protocol

The balls-and-bins compliant protocol obtained in this section considers a framework which also applies to epidemic protocols. Note that such a framework is not only intended to propagate new events fast, but must also be able to organise the membership of a group, i.e. it must deal with members joining and leaving the group.

Hereby, a group $G = \{p_1, \ldots, p_n\}$ is defined as a set of processes. Processes exchange in every round *gossip* messages where each gossip message includes information about members that joined the group, members that left the group, and events to be delivered to the application layer of the gossip protocol. In order to keep track of possible communication partners, each process $p_i$ maintains a view $V_i \subset G$. Let $K$ denote the fan-out, i.e. the number of communication partners chosen in a round. Each process $p_i$ communicates with $K$ communication partners chosen uniformly at random from its view $V_i$. The framework of the protocol is based on every process doing the following computation in every synchronous round (see also the basic protocol in Figure 3.2):

1. evaluate received gossip messages by processing joining and leaving members and events

2. create a new gossip message and send it to $K$ randomly chosen neighbours known from the view

For the later analysis it may be helpful to keep in mind that the destination of a gossip message will determine the destination of a ball associated with an event.

The first part of the framework is concerned with the maintenance of the local view of processes. The evaluation of members which either join the group or leave the group determines the new view of the process. In the following we assume that the view of a process allows the selection of $K$ randomly chosen elements from the whole group. If the process maintains only a partial view, we consider only schemes as [Ganesh et al. 2001; Eugster et al. 2001a] where the views converge to a uniform random distribution.

The second part of the framework deals with the propagation of events such that every process can receive the event. Therefore, we provide a scheme in order to place sufficiently many balls associated with an event into distinct bins. By using a set denoted by *newevents*, a process keeps track of the events which have been delivered in the current round. Another set denoted by *history* contains the events which have been delivered to the application layer. An event $e$ with $e \in$ *newevents* and $e \notin$ *history* will be added to *history* and delivered to the application layer.

In order to provide termination for the propagation of an event, each event is associated with a *tag value*, which indicates the number of rounds the event has

*Basic Protocol:*

**for all** rounds in which $p_i \in G$ **do**
  evaluate received gossip messages by processing joining and leaving members and events
  create a new gossip message $M$
  **for** $l = 1$ to $K$ **do**
    choose $j \in V_i$ uniformly at random
    send $M$ to $p_j$
  **end for**
**end for**

*Evaluation of events:*

**for all** events $e$ received in a round **do**
  **if** $e \notin$ *history* **then**
    deliver($e$)
  **end if**
  tag $=$ gettag$(e)$
  **if** $tag < r$ **then**
    $M = M \cup (e, tag + 1)$  */* add the event to the gossip message $M$ */*
  **end if**
**end for**

Figure 3.2: The balls-and-bins compliant protocol consists out of the basic protocol and the evaluation of events. It can be adapted to various schemes determining the membership of a group.

been propagated. A new gossip message contains, in addition to the information on members joining and leaving the group, all events $e \in$ *newevents* whose tag value is smaller than a parameter $r$, indicating the maximum number of rounds an event needs to be propagated.

The complete scheme including the basic protocol and the evaluation of events is illustrated in Figure 3.2. Only the evaluation of members joining and leaving the group is omitted since the protocol is intended to serve many different ways to organise the membership.

Now, one can look at the spreading of each event as a separate balls and bins game, where the placing of the balls is determined by the random destination of the gossip message. Whenever a process receives an event which has stayed less

than $r$ rounds in the system, the process will create $K$ balls, which are sent with $K$ different gossip messages to their destinations. Hence, by choosing $r = cn \log n$ for $c > 1$ we can derive theorem 3.4.1.

**Theorem 3.4.1** *Let $K$ denote the fan-out and let $r$ denote the maximum number of rounds an event is propagated with respect to the provided gossiping scheme. Then, for any $K \geq 1$ there exists a fixed $r$ such that the provided scheme becomes balls-and-bins-compliant.*

Note that the introduced scheme makes explicit use of the possibility to allow processes to change states between being infectious and being resistant several times. In the following section a performance analysis is presented, which bounds the value of $r$ depending on the fan-out $K$.

## 3.5 Performance analysis

The focus of this analysis is to bound the number of rounds indicated by parameter $r$ such that the introduced balls-and-bins-compliant protocol of the previous section can be applied for practical purposes. The reliability of the protocol is expressed as a guarantee that an event created by a process will reach all other processes with high probability. As a main result it is shown that for a small fan-out $K = \lceil 2e \ln n / \ln \ln n \rceil$, it is sufficient to choose $r = O(\log n)$ to guarantee the delivery of an event to all destinations w.h.p. Further, an analysis for constant fan-out proves that $r = O(\sqrt{n})$ is sufficient.

The analysis is based on the balls-and-bins model introduced in Section 3.3 and uses in particular corollary 3.3.1. We must ensure that within $r$ rounds sufficiently many balls associated with an event are created. This is achieved by examining the number of balls which are placed within a round $i$ denoted by $m_i$ and determining $r$ such that

$$\sum_{i=1}^{r} m_i \;\geq\; cn \log n \quad \text{w.h.p.}$$

Hereby, we aim to relate the number of balls which are placed into random bins within two consecutive rounds, i.e. we are looking for a statement such that for a positive constant $d > 1$

$$m_{i+1} \;\geq\; d \cdot m_i \quad \text{w.h.p.} \tag{3.1}$$

From such a statement it is shown how one can derive the number of rounds until a minimum number of balls is created in every round, where the number of rounds needed is required to be logarithmic in the minimum number of balls. Therefore, similar to the random allocation problem we look into the maximum number of balls that may fall into any single bin.

**Theorem 3.5.1** *Let $m$ denote the number of balls, which are placed into $n$ bins chosen uniformly and independently at random. If $m \leq n$ holds, then every bin contains no more than $e\frac{\ln n}{\ln \ln n}$ balls with probability $\frac{1}{n}$.*

The proof is not presented since a detailed proof of this theorem is given in [Motwani and Raghavan 1995] (see pages 44 ff.). A generalisation of theorem 3.5.1 can be found in [Raab and Steger 1998].

Let now $m_i$ denote the number of balls placed into random bins in the $i$th round. Applying theorem 3.5.1 leads to the following corollary, which gives an estimate of how much time is needed until we can expect at least $n$ balls to be placed into bins within a round.

**Corollary 3.5.1** *If $K \geq 2e\frac{\ln n}{\ln \ln n}$ and $m_i \leq n$, then $m_i \geq 2m_{i-1}$ with probability $1 - \frac{1}{n}$.*

Finally, from theorem 3.5.2 we can conclude that the protocol needs $O(\log n)$ rounds to terminate.

**Theorem 3.5.2** *Let $K$ denote the fan-out of the presented balls-and-bins compliant protocol. If $K \geq 2e\frac{\ln n}{\ln \ln n}$, then after $O(\log n)$ rounds every process of the group is informed about an event w.h.p.*

**Proof.** We show that after $O(\log n)$ rounds an overall of $cn \log n$ balls have been placed into bins.
Let $\xi_i$ denote the event that the number of balls did not double in round $i$ for which $m \leq n$. Then, by corollary 3.5.1 it follows that

$$\mathbf{Pr}[\xi_i] \leq \frac{1}{n}.$$

Therefore, the occurrence of any event can be expressed as the union of $\log n$ events $\xi_i$:

$$\mathbf{Pr}[\cup_{i=1}^{\log n} \xi_i] \leq \sum_{i=1}^{\log n} \xi_i \leq n^{-1/2}$$

and thus after $\log n$ rounds the number of balls placed into random bins in a round is at least $n$ with probability $1 - n^{-1/2}$.
Using the same argument for the next $c \log n$ rounds, the number of created balls in a round will not decrease below $n$ with probability $1 - c\, n^{-1/2}$.
Hence, with probability $1 - o(1)$, $cn \log n$ balls have been created and placed into random bins after $(c+1) \log n$ rounds. □

Note that $\ln n / \ln \ln n$ grows very slowly, so that the fan-out value can be regarded as sufficiently small for all practical purposes. Further, the maximum

number of messages a process receives in a round is expected to be smaller than $O(\frac{\log n}{\ln \ln n})$ since the number of balls which are created within one round is clearly bounded by $Kn$, and placing $O(n \log n)$ balls randomly into bins gives a maximum of $O(\log n)$ balls in each single bin.

If we assume that the fan-out $K$ is constant, one needs $\Omega(\log n)$ rounds as we cannot expect to place more than $Kn$ balls in a round. Further, as stated in theorem 3.5.3, it is shown in the following that $O(\sqrt{n})$ rounds are sufficient. To determine the result, we examine once again for which values it is possible to achieve a statement as given by equation 3.1.

**Lemma 3.5.1** *If $m \leq \sqrt{n}\log_{20}(n)$ holds, then every bin contains no more than 6 balls with probability $\frac{1}{n}$.*

**Proof.** For $m = \sqrt{n}\log_{20}(n)$ the probability that bin $i$ receives exactly $j$ balls is given by

$$\begin{aligned} & \binom{m}{j}\left(\frac{1}{n}\right)^j\left(1-\frac{1}{n}\right)^{m-j} \\ \leq\ & \left(\frac{em}{jn}\right)^j \\ =\ & \left(\frac{e\log_{20}(n)}{j\sqrt{n}}\right)^j. \end{aligned}$$

Let $\xi_i(k,m)$ denote the event that bin $i$ has $k$ or more balls in a bin after m balls have been tossed. Then for $k \geq 6$,

$$\begin{aligned} \mathbf{Pr[}\xi_i(k,m)\mathbf{]} &\leq \sum_{j=k}^{n}\left(\frac{e\log_{20}(n)}{j\sqrt{n}}\right)^j \\ &\leq \left(\frac{e\log_{20}(n)}{k\sqrt{n}}\right)^k \frac{1}{1-\left(\frac{e\log_{20}(n)}{k\sqrt{n}}\right)} \\ &\leq 2\left(\frac{e\log_{20}(n)}{k\sqrt{n}}\right)^k. \end{aligned}$$

In particular, for the event $\xi_i(6,m)$ we obtain

$$\mathbf{Pr[}\xi_i(6,m)\mathbf{]} \leq 2\left(\frac{e\log_{20}(n)}{6\sqrt{n}}\right)^6.$$

By applying $\log_{20}(n) \leq n^{1/3}$, it follows that

$$\mathbf{Pr[}\xi_i(6,m)\mathbf{]} \leq n^{-2}.$$

Taking the union of the events $\xi_i(6, m)$, for $i = 1, \ldots, n$, yields the desired result. □

Using the same proof technique as for theorem 3.5.2 one can derive theorem 3.5.3.

**Theorem 3.5.3** *Let $K$ denote the fan-out of the presented balls-and-bins compliant protocol. If $K \geq 12$ holds, the protocol needs $O(\sqrt{n})$ rounds in order to inform every process of the group about an event w.h.p.*

## 3.6 Discussion

Although the epidemic model has been shown to be useful to estimate the performance of various protocols, it is a hard problem to obtain a precise bound if one assumes limitations on the resources available to processes. As motivated by *lpbcast* [Eugster et al. 2001a] and *SCAMP* [Ganesh et al. 2001] the next generation of protocols will exactly deal with these requirements.

To cope with limited resources, the epidemic model needs to consider the possibility that a resistant process may become susceptible again. Although in general this scenario is undesirable, it occurs when protocols limit the size of the history buffer. The history buffer itself determines whether a process can stay resistant against an event. If a process deletes an event too early from its history buffer it is susceptible again and potentially delivers the event multiple times to the application layer. In this case the protocol is said to suffer from the *multiple delivery problem*. Even worse an epidemic process which almost stopped could be started once over again, thus preventing the epidemic process to terminate. In fact, simulation results confirm that even a small infection rate of new events can lead to a significant overhead.

This scenario requires to consider more differential equations and needs a far more complex analysis. Therefore, the motivation for this work was to provide a simpler model as an alternative. The analysis as presented in Section 3.5 is based on elementary mathematics and can this way facilitate the buffer analysis by examining the collisions of balls containing different events.

## 3.7 Conclusion

This work has shown an alternative approach of analysing gossiping based on the occupancy problem. With respect to the balls-and-bins model, a new protocol is presented based on existing practical approaches [Eugster et al. 2001a; Birman et al. 1999]. The maintenance of the group membership can be adapted to various schemes which allow processes to select uniformly random group members from its

view. The reliability of the protocol is proven as a statement with high probability that an event, created by an arbitrary process, will reach all group members. The presented analysis shows that an event needs to remain for $O(\log n)$ rounds in the system if the fan-out is $\lceil 2e \ln n / \ln \ln n \rceil$, and $O(\sqrt{n})$ rounds if a constant fan-out is used.

The future work focuses on analysing the tightness of the presented bounds. Moreover, an analysis on the average number of gossip messages imposed by the spreading of an event is needed. Note that only a lower bound can be derived from the fact that at least an overall of $cn \log n$ messages are needed in order to ensure that every process receives the event with high probability. Finally, work in progress looks into ways to manage buffer space so as to improve the multiple delivery problem discussed in Section 3.6.

## Acknowledgements

Thanks a lot to Patrick Thomas Eugster, Anders Gidenstam, Marina Papatriantafilou and Philippas Tsigas for discussions and comments that contributed to this work.

# Four

# Buffer Management in Probabilistic Peer-to-Peer Communication Protocols

Boris Koldehofe

**Abstract**

In multipeer communication decentralised probabilistic protocols have received a lot of attention because of their robustness against faults in the communication traffic and their potential to provide scalability for large groups. These protocols provide a probabilistic guarantee for a propagated event to reach every group member. Recent work aims to improve the scalability of such protocols by reducing memory requirements. In saving memory resources, the history buffer, which is used to "remember" received events and to prevent multiple deliveries of events to the application, plays a very significant role. In this paper[1] we examine how the buffer size should be chosen to challenge the multiple delivery problem. Further, we propose and evaluate several methods of organising the dissemination of events in order to provide high reliability and reduce the number of multiple deliveries at the same time.

**Keywords:** *peer-to-peer communication, multipeer communication, gossiping, networking, evaluation, fault tolerant communication, fault tolerant protocols*

[1] A version of this paper has been published in the proceedings of the Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS '03)

## 4.1 Introduction

Many distributed applications like multi-user collaboration rely on multipeer communication. In multipeer communication processes sharing a common interest form a group. Inside a group multiple senders can communicate with multiple receivers on a peer-to-peer basis (many-to-many communication) by sending information embedded in an *event*. Multipeer communication must

- provide a management of group membership to support processes joining and leaving a group,
- give *reliability* guarantees of the communication,
- provide *scalability* which is determined by the number of processes supported and the reliability provided,
- and be robust against multiple deliveries of the same event to the application.

The focus of this paper is on lightweight probabilistic peer-to-peer dissemination protocols like lpbcast [Eugster et al. 2001a; Eugster et al. 2001b], SCAMP [Ganesh et al. 2001], and [Koldehofe 2002]. While previous approaches require processes to maintain a full view of all group members, these protocols aim to save memory resources by introducing for processes a partial view of the whole group.

In this paper we study another important and memory intensive resource of lightweight probabilistic dissemination protocols, the *history buffer*. While deterministic approaches maintain a history of events in order to handle retransmission of events (a study of such a kind of buffer management is presented in [Xiao et al. 2002]), the history buffer in lightweight probabilistic dissemination protocols serves a different purpose. Lightweight approaches use the history buffer in order to detect whether an event, received by a message, is *new*, i.e. it is delivered for the first time. Since the flow of information is non-hierarchical and we aim for many-to-many communication, one *cannot rely on global sequence numbers* which could be used to detect new events. Instead, a structure like the history buffer is needed to make the protocol robust against the *multiple delivery problem* which is the occurrence of multiple deliveries of the same event to the application. The information whether an event is new is sometimes also exploited in order to decide whether to end the dissemination of an event.

Limited resources, for instance in embedded systems [Davis et al. 2000; Gupta et al. 2001], indicate the need for restrictions on the size of the history buffer. However, a process which provides only a buffer of bounded size becomes vulnerable to multiple deliveries. As shown in the study of lpbcast [Eugster et al. 2001b] the

size of the buffer can affect the reliability of the protocol. Events may stay a long time in a system and reduce the probability of successful delivery for other events.

It is a difficult problem for processes to estimate a suitable size of their buffers in order to avoid multiple deliveries. Processes themselves are not aware of having delivered the same event multiple times. However, the occurrence of multiple deliveries themselves may change the settings of the system, for instance the time during which events stay in the system may increase. At the time, in order to estimate a suitable buffer size, processes can mainly rely on

- observations of the frequency at which new events are created,
- an estimation of the group size,
- and an estimation of the time events are disseminated in the system in order to provide reliability.

So far only little work has been carried out to study the buffer management of lightweight probabilistic dissemination protocols. In [Kouznetsov et al. 2001] a framework is shown in which events are removed from their buffer depending on their age, which is determined by the number of hops an event has performed. The study evaluates on the example of pbcast [Birman et al. 1999] and lpbcast [Eugster et al. 2001a] possible improvements of the approach in throughput and message stability (i.e. all members of a group have received the event).

In this paper we study buffer management and dissemination with respect to the multiple delivery problem by combining reliability with the avoidance of multiple deliveries for an event. We present a model which relates the buffer system to a queueing model. Based on this model we show an analysis suitable to estimate the vulnerability to multiple deliveries by determining from known parameters in the system when the size of the buffer becomes critical. For critical buffer sizes we show that events need an explicit way to terminate the dissemination. Building on these results and previous work [Koldehofe 2002], we propose and evaluate several ways to organise buffer space in order to provide high reliability *and* a low rate of multiple deliveries.

**Organisation of the paper.** In Section 4.2 background information on probabilistic multipeer communication is presented. Section 4.3 relates the buffer system to a queueing model and proves a bound on the occurrence of multiple deliveries in any dissemination system. Further, in Section 4.4 the impact of the dissemination on multiple deliveries is studied based on common ways to disseminate events and a general framework which supports multiple event dissemination and management of group membership. It is shown that events need to provide an explicit way to

terminate the dissemination. Building on this result, Section 4.5 proposes several approaches to manage buffer space and provide termination to the dissemination. In Section 4.6 the proposed approaches are evaluated together with the introduced dissemination systems.

## 4.2 Background

This section gives some background information on the existing literature on probabilistic multipeer communication protocols, as well as reliability and scalability properties.

Generally in group communication, reliability is the guarantee that an event will inform all group members. One can express reliability guarantees for group services (see also [Wittmann and Zitterbart 1999]) by distinguishing among group services which are i) *reliable*, i.e. either all non faulty processes or no process will be informed about an event sent by a process, or ii) *predictable reliable*, i.e. the protocol gives a probabilistic guarantee that events will reach all non faulty group members.

Virtual synchrony [Birman and Joseph 1987a], a deterministic approach, satisfies the (strong) reliability criteria in the presence of faults, but provides only very limited scalability. More recent deterministic approaches like receiver-initiated and hierarchical approaches improved scalability a lot. However, receiver initiated approaches like SRM [Floyd et al. 1997] require unbounded memory and synchronisation of timers, while hierarchical approaches like RMTP [Lin and Paul 1996] do not provide, as required in the definition of reliability, full end-to-end reliability in the occurrence of faults.

Alternatively, one can reason on predictable reliability. Protocols based on *gossiping* provide predictable reliability and they can scale at the same time to large groups. Gossiping protocols in group communication have received a lot of attention since the introduction of pbcast [Birman et al. 1999]. The communication in each round is based on a process selecting its communication partners at random. Because of the non hierarchical flow of information a failure of a single process affects the overall reliability only very little.

The predictable reliability of gossiping protocols is often related to an approximation of the epidemic model [Bailey 1975], which was first adapted to distributed computing in the context of replicated databases [Demers et al. 1987]. The dissemination of events is handled to infect as many processes as possible. Analytical evaluation of several variants of the epidemic model [Pittel 1987; Karp et al. 2000] allows to give a probabilistic guarantee of the number of processes that will receive an event as well as to bound the number of rounds to terminate the protocol. Ide-

ally, the guarantee is given with high probability, i.e. with probability $O(1-(n^{-k}))$ where $k$ is a constant and $n$ denotes the number of processes.

While previous approaches usually consider a process to know all members of the group, recent work like lpbcast [Eugster et al. 2001a] and SCAMP [Ganesh et al. 2001] showed how to provide for processes a partial view on the group. Further, the protocols have a decentralised membership scheme inherent. The membership scheme allows processes to join and leave the group arbitrarily by contacting any process of the group. The reliability of lpbcast is based on maintaining the distribution of members inside a view such that a process can choose its communication partners as if they were chosen uniformly at random among all members. In SCAMP the reliability is based on modelling the possible choice of communication partners as a random graph.

The communication of gossiping protocols usually happens by processes sending gossip messages to $K$ destinations in a round, where $K$ denotes the *fan-out* of the protocol. In an asynchronous setting, a process sends gossip messages to its destination periodically after a fixed time interval has passed. Selecting communication partners at random makes these protocols robust against processes and link failures. Moreover, the load of the communication is evenly distributed among all processes. The *dissemination system* of a gossiping protocol decides which events a process includes in a gossip message and how the destinations are chosen. In addition, it decides when to deliver an event to the application. To provide reliability most dissemination systems reflect the epidemic model. An important aspect for analysing the performance of gossiping protocols is to obtain a good estimation on the number of rounds, denoted by $r$, until a predictable reliability is given for an event to inform all members of the group. Such a bound can also be useful to terminate the dissemination of events, as we discuss in Section 4.4 and Section 4.5. In this case events that stay longer than $r$ in the system will not be considered by the dissemination system.

## 4.3 Modelling and analysing the history buffer

The history buffer, as used in many gossiping protocols, consists of a set of identifiers, each identifier referring to an event. We also say an event is contained in the history buffer if and only if its identifier is contained in the history buffer. Since events are propagated in a randomised way, they are likely to reach destinations several times. Whenever a process receives an event, the process can check the history buffer whether the event is new and in this way avoid multiple deliveries of the same event to the application. However, a process which removed the identifier of an event too early, may suffer from multiple deliveries. If an event is delivered

at most once to all locations it is said to be delivered *safe*.

A bounded buffer size in a dynamic system where new members join and leave the group is always vulnerable to multiple deliveries since there is no bound on the number of events which are disseminated at the same time and events may be observed out of order. Nevertheless, a process can estimate based on local observations the probability that a new event which is not contained inside the history buffer was delivered before. By using a probabilistic approach one can distinguish between safe and critical buffer sizes. We define the size of the buffer to be *safe* with respect to a safety-parameter denoted by $\lambda$ if the size of the buffer is sufficiently large to ensure safe delivery for an event with probability greater than $1 - \lambda$. Otherwise the size of the buffer is called *critical*.

In the following we consider a general model and analysis suitable to determine the probability for multiple deliveries for an arbitrary dissemination system denoted by $\mathcal{D}$. Let $m$ denote the time determined by the number of rounds an event stays at most in $\mathcal{D}$ and $n$ the number of processes communicating with $\mathcal{D}$. We relate the buffer system to a queueing model using a single server where new events are admitted to the queue as a random process. However, contrary to ordinary queueing-theory the service time in this model depends on the arrival times of events. The service time is chosen such that every event stays at least as long in the queue as it needs to stay in the buffer of $\mathcal{D}$ in order to guarantee safe delivery. Hence, the question whether the queue is stable is not an issue here. Instead, we investigate the probability that the length of the queue exceeds the choice of the length for the buffer of $\mathcal{D}$.

Let $a_i$ denote the arrival time of an event $e_i$. Then the server processes each event at time $s_i = a_i + m$. Hence, if the length of the buffer in $\mathcal{D}$ is greater than the maximum length of the queue within the time interval $[a_i, s_i]$ then $\mathcal{D}$ can deliver $e_i$ safe.

Let $[t_a, t_s]$ denote an interval of length $m$ and the random variable $X_{i,j}$ denote the event that at time $t_a + i$ process $j$ admits a new gossiping event to the system. Further, we assume that all $X_{i,j}$ occur independently and each event yields $\mathbf{Pr}[X_{i,j} = 1] = p$ and $\mathbf{Pr}[X_{i,j} = 0] = 1 - p$. The number of admitted gossiping events can be represented by the random variable

$$X := \sum_{j=1}^{n} \sum_{i=1}^{m} X_{i,j}.$$

Then 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

$$\mathbf{E}[X] = pnm.$$

Clearly, the length of the buffer must be chosen at least as large as **E[**$X$**]**. Otherwise we are expected to encounter a large number of multiple deliveries.

As a next step we bound the buffer size so that the probability for multiple deliveries of an event becomes low by applying a Chernoff bound for binomial distributions as stated in [Motwani and Raghavan 1995].

**Theorem 4.3.1** *Let $e_i$ be an event admitted to a gossiping algorithm with dissemination system $\mathcal{D}$ where each event is required to stay in $\mathcal{D}$ for $m$ rounds. Further, for each of the $n$ processes in the system let $p$ denote the probability that the single process admits a new event to $\mathcal{D}$ in a round. Then $\mathcal{D}$ can prevent the multiple delivery of $e_i$ with probability strictly greater than*

$$1 - \left(\frac{e}{4}\right)^{nmp}$$

*if the size of the buffer is chosen greater than or equal to $2nmp$.*

**Proof.** Applying the Chernoff bound for binomial distributions yields for any $\delta > 0$

$$\textbf{Pr[}X > (1+\delta)nmp\textbf{]} < \left(\frac{e^{\delta}}{(1+\delta)^{\delta+1}}\right)^{nmp}.$$

By choosing $\delta = 1$, the result follows. □

Note that a similar result can be achieved for the Poisson distribution. It is also possible to use the more general assumption that each process $i$ admits a new event to the system with probability $p_i$. If

$$\mu = \sum_{i=1}^{n} p_i$$

a buffer size of length $2\mu m$ guarantees safe delivery of an event with a probability strictly greater than $1-(\frac{e}{4})^{\mu m}$. In the experimental study, described in the following sections, we use the simpler formula as described in Theorem 4.3.1 in order to distinguish between safe and critical buffer sizes.

Theorem 4.3.1 provides an easy way to relate safe buffer sizes to local observations by a process, as stated in Corollary 4.3.1.

**Corollary 4.3.1** *Let $\mathcal{D}$ denote any dissemination system with parameters $m$, $n$, and $p$, denoting the number of rounds an event needs to stay in $\mathcal{D}$, the number of processes, and the arrival rate of new events respectively. The size of the history buffer is safe with safety-parameter $(e/4)^{nmp}$ if the size of the buffer is chosen greater than or equal to $2nmp$.*

## 4.4 Dissemination in relation to multiple deliveries

In the following we examine the impact of the dissemination system on buffer sizes. Towards that we first introduce a general framework for dissemination of multiple events, possible ways to implement dissemination systems, and study the impact of event termination on the multiple delivery problem.

### 4.4.1 A general framework for dissemination of multiple events

In order to disseminate multiple events and manage processes joining and leaving a group gossiping protocols have to use a framework which allows to combine group management with the dissemination of events. The evaluation of buffer management in this paper uses a general framework, which can be combined with existing ways of group management for lightweight approaches. Let a group, denoted by $G$, consist of a set of processes, i.e. $G = \{p_1, p_2, \ldots, p_n\}$. In every round processes of the group exchange gossip messages. A gossip message contains information on members which joined or left the group and events to be delivered to the application. In order to select communication partners a process manages a view $V \subset G$ from which it can choose processes of the whole group uniform at random. In every round a process evaluates the gossip messages it receives from other group members. A process may also receive a new event from the application which must be propagated. Based on the information which a process obtains in a round from other communication partners and the application, a process creates a new gossip message with respect to the dissemination scheme and propagates it to $K$ neighbours chosen uniformly at random from its view. The framework used for all protocols introduced in the paper is illustrated in Figure 4.1.

### 4.4.2 Dissemination systems in gossiping

Let $r$ denote the number of rounds until a predictable reliability is given. A possible way to obtain $r$ is based on an approximation of the results presented in [Pittel 1987]. The analysis uses a simplified model in which during the life time of an event processes never loose interest in propagating this event. Let $n$ denote the group size and assume that a process which has received an event informs only one other process in a round. The number of rounds needed to inform all group members with high probability is bounded by

$$r = \log n + \log_{10} n + O(1).$$

However, the analysis does not match exactly the requirements of gossiping protocols in the spirit of pbcast [Birman et al. 1999] because the analysis assumes that

*Basic Framework:*

**for all** rounds in which $p_i \in G$ **do**
  evaluate received gossip messages by processing joining and leaving members and events
  create a new gossip message $M$
  **for** $l = 1$ to $K$ **do**
    choose $j \in V_i$ uniformly at random
    send $M$ to $p_j$
  **end for**
**end for**

Figure 4.1: A general framework for dealing with membership and communication of multiple events.

every process keeps its interest in propagating an event until everybody is informed with high probability. This means a processes has to propagate all events which it has noticed and which have travelled less than $r$ rounds in the system. Although a process communicates only to one neighbour in a round, many events will increase the message sizes and the dissemination scheme will not scale to many events in the system.

In the study of buffer management we will consider two different methods of event dissemination:

1. Approaches like pbcast [Birman et al. 1999] and lpbcast [Eugster et al. 2001a] suggest that a process propagates only new events, otherwise it ignores them. The reduced number of rounds in which a single process communicates about the same event is partially compensated by sending every message to multiple neighbours. If the fan-out is chosen such that $K \geq 2$, the propagation of a single event is expected to terminate after $O(\log n)$ steps. Unless $K = \Omega(\log n)$ the dissemination scheme does not provide the guarantee as discussed above, however it succeeds to inform a high percentage of processes.

2. A way to increase reliability is to propagate events whenever they are received by a process. An analysis of this model, based on a balls-and-bins approach [Koldehofe 2002], shows that such a dissemination system can inform all group members about an event with high probability if $K = O(\ln n / \ln \ln n)$ and $r = O(\log n)$. In order to deal with the problem of many events being collected in a round, a process will propagate a maximum number of events with each gossip message. If the number of events is mod-

erate and events will not conflict with others during $r$ rounds, the scheme provides high reliability.

### 4.4.3 Termination of the dissemination

In this section, it is shown why dissemination systems should provide an explicit termination for disseminating an event by adding a *tag* to the gossip message. A tag provides the dissemination system with information to decide when to terminate the dissemination of an event. A simple mechanism is to count the number of hops: a process will not propagate information based on a message that exceeded a predefined number of hops.

Protocols without tag values and under ideal conditions, i.e. with unlimited buffer size, can provide termination. For example, lpbcast only propagates events which are not found in the history buffer. This way, every process sends at most once a gossip message about the same event to their neighbours.

In Figure 4.2 the behaviour of lpbcast under three different buffer sizes is illustrated: safe with safety-parameter $0.05$ (c.f. Figure 4.2(a)), critical (c.f. Figure 4.2(b), 4.2(c)), and almost safe (c.f. Figure 4.2(e), 4.2(d)). Figure 4.2(c) corresponds to a more detailed representation of Figure 4.2(b), and Figure 4.2(e) to a more detailed representation of Figure 4.2(d), respectively. The simulation used a system with $100$ processes and a fan-out of $5$. The probability that a process admitted a new event to the system during a round was $0.01$. The results are based on the evaluation of approximately $250000$ events in the system.

From Figure 4.2(a) one can observe that in $99.7\%$ of all cases $m$ can be bounded by $8$. Applying the analysis from Section 4.3, a buffer size of $16$ should suffice to guarantee at least $95\%$ of events to be safe. In the simulation the number of safe events was significantly higher with $99.98\%$.

On the contrary, if the buffer size is chosen to be only $8$ (critical), the number of multiple deliveries and the time for events to terminate increases significantly (c.f. Figure 4.2(b), 4.2(c)). The amount of events encountering multiple deliveries is around $57\%$. Also the number of multiple deliveries for each event is remarkably high, given by $3130$ multiple deliveries in the average. Further, it should be noted that $32\%$ of all events terminate after the second round. These events do not suffer from multiple deliveries, but they are delivered only by a low number of processes, decreasing the reliability. Although some events terminate early, the termination times for events are distributed over a large time interval (up to $1900$ rounds). The time interval shown in Figure 4.2(b) covers $72\%$ of all events, while the time interval shown in Figure 4.2(c) covers $25\%$ of all events.

Finally, in Figure 4.2(d) and Figure 4.2(e), the behaviour of the system is illustrated if the buffer size was chosen almost safe initially, i.e. a size equal to $14$. The

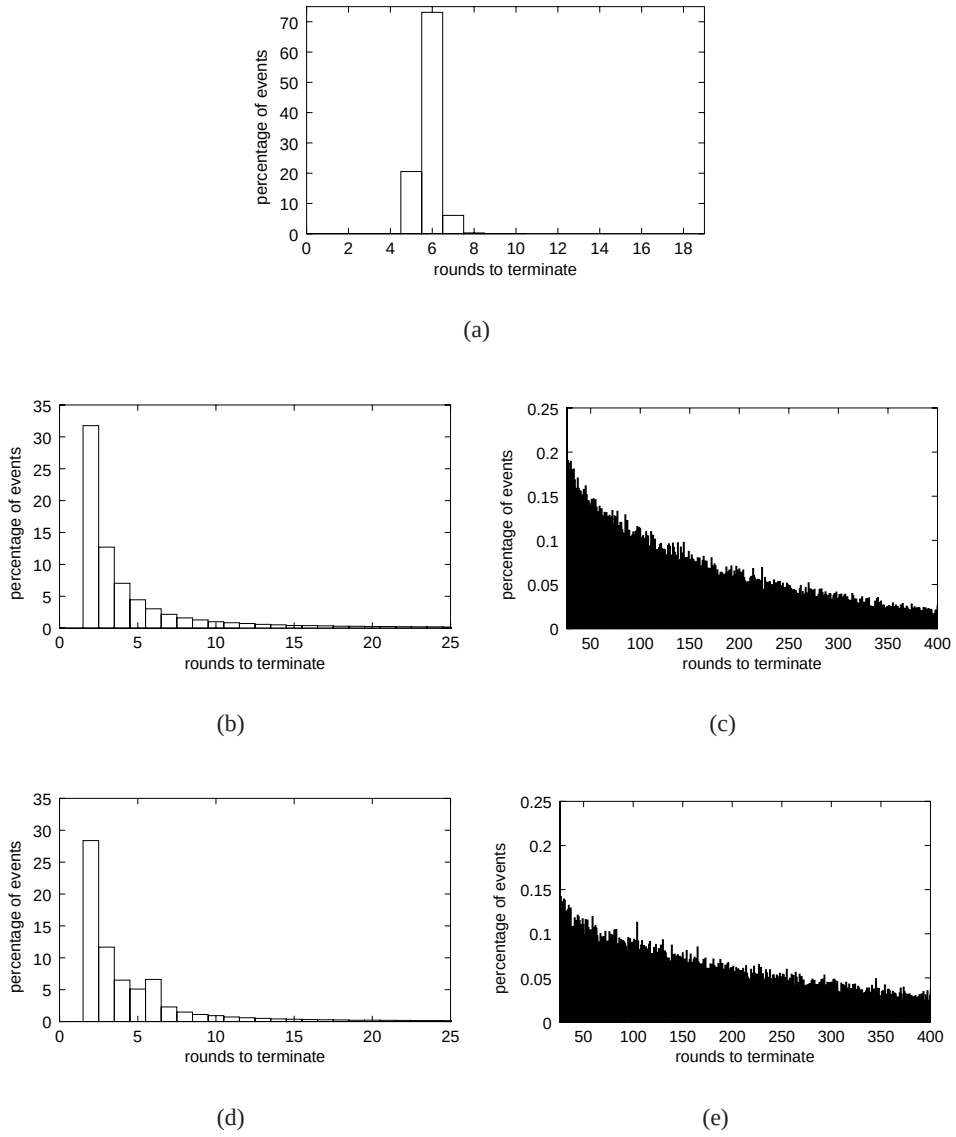



Figure 4.2: This data shows the distribution determined by the time interval events were delivered to the application in a simulation of lpbcast. Three different settings are shown, where 4.2(c) corresponds to a more detailed view of 4.2(b), and 4.2(d) to a more detailed view of 4.2(e) respectively. The group consisted of $100$ processes and each simulation disseminated approximately $250000$ events. The buffer sizes varied from $16$ in 4.2(a), $8$ in 4.2(b), 4.2(c) and $14$ in 4.2(d), 4.2(e). The view of each process was limited to $50$ processes.

behaviour reveals the problem we obtain from not choosing any termination tags. Although the number of multiple deliveries is not high initially, the occurrence of multiple deliveries changes the response time $m$ of the system, and the increase in $m$ also increases the vulnerability of the system to multiple deliveries. The system converges to an unstable state where the termination times are similarly distributed as if the buffer size was chosen critical initially.

**Conclusion.** The avoidance of multiple deliveries requires that gossip messages carry information which determine when to terminate the dissemination. This can be done by either adapting the size of the buffer according to the response of the system, or by terminating the message by using a computation based on tag values.

## 4.5 Management of buffer space and selection of tag values

In this section we discuss approaches to manage the history buffer of the dissemination systems in Section 4.4.2 with the goal to increase reliability and reduce multiple deliveries. The approaches provide a method for choosing tag values for events in order to guarantee the termination of dissemination and to avoid the problems discussed in Section 4.4.3.

In order to facilitate the description of the approaches we denote an event by $e$ and the respective tag value by $t(e)$. Further, $r$ denotes the number of rounds the dissemination system needs to propagate an event in order to be reliable, $\sigma$ an offset increased by one in every round, and $\Phi(e)$ the potential of an event.

**FIFO buffering with limited number of hops.** A common way to manage buffer space is to use a FIFO queue. New events (or its identifiers) will be added to the tail of the queue while old events will be removed from the head such that the length of the queue always respects the specified length of the buffer.

The termination of the dissemination is specified by assigning to each event a tag value. If $e$ is a new event, then $t(e) = 0$. Whenever $e$ is propagated with a gossip message, $t(e)$ is increased by one. If $t(e) > r$, $e$ will not be propagated by the dissemination system.

A criticism towards the FIFO strategy is that fresh events may be deleted earlier from the buffer at one process because these events have been received in different order than older events. If a fresh event is deleted in an early stage, it will take significantly more time to reach everybody in the group. At the same time in terms of the multiple delivery problem, an event which is disseminated longer than other events is more likely to cause multiple deliveries.

**Priority-queue based approaches.** In the following we present a priority-queue based approach. Within the queue events are associated with a *priority* which is determined by the *potential* of an event to infect other processes. New events are inserted with high priority while the priority of old events decreases over time. If the insertion of new events causes the buffer size to exceed its limit, the events with lowest priority are removed.

The use of priorities can help to prevent fresh events from being removed. We examine two different ways of assigning a potential to events.

1. *Estimated time to terminate (ETT).* This approach is based on the estimation of rounds an event needs to reach all participants and counting the number of hops an event has performed. In the round an event $e$ is created we set $t(e) = 0$ and calculate its potential $\Phi(e) = \sigma + r$. In the case a process receives $e$ for the first time it calculates the potential $\Phi(e) = \sigma + r - t(e)$. When propagating an event the processes increases $t(e)$ by one.

   We ensure that the values for $\sigma$ are finite by considering the case when the value of $\sigma$ exceeds a predefined maximum value. In this case the potential for each event $e$ of the buffer is recalculated by $\Phi(e) = \Phi(e) - \sigma$ and $\sigma$ is reinitialised to zero.

2. *Estimated potential approach (EP).* This approach is based on the fact that for a constant $c > 1$, placing $cn \log n$ balls uniformly at random into $n$ bins is sufficient for every bin to receive at least one ball with high probability. A new event initially starts with $t = [2n/K \log n] - 1$ balls. The potential in the history buffer is calculated by $\Phi(e) = [\log t(e)] + 1 + \sigma$ assuming that for the fan-out $K \geq 2$ holds (in this case the number of rounds to terminate the epidemic of an event is logarithmic in the initial tag value). Let $t_1(e), \ldots, t_l(e)$ denote the tag values a process received for an event in a round. When propagating $e$ it calculates $t(e) = [1/K \sum_{i=1}^{j} t_i] - 1$. If $e$ was received for the first time the potential in the history buffer is calculated in the same way, i.e. $\Phi(e) = [\log(t)] + 1 + \sigma$. The problem to ensure finite values for $\sigma$ can be solved in the same way as discussed for the ETT-approach.

The local cost of the priority-queue based approaches is determined by inserting every new event into the queue and removing it from the queue at some time. If no multiple deliveries occur, FIFO and Priority-queue based approaches use the same amount of insertions and deletions. The local cost for inserting an event is higher for the priority based queue, but still small to allow an efficient implementation. Further, the priority-queue based approaches have at certain times to recycle the offset values and therefore to decrease the priority of every element in the buffer.

In the next section we combine the presented schemes to manage buffer space with the formerly introduced dissemination systems and the general framework for dissemination of multiple events (see Section 4.4). We investigate whether the slightly more complex and costly way of implementing a buffer locally helps to reduce the number of multiple deliveries and increases reliability.

## 4.6 Evaluation of buffer management based on FIFO, ETT and EP

The focus of the presented evaluation is to investigate reliability together with multiple deliveries of the schemes for buffer management of Section 4.5 when the buffer size is chosen critical. We combine these approaches with the schemes for dissemination systems of Section 4.4.2 by introducing four protocols. The protocols use a common framework to propagate multiple events at the same time and organise the membership of the group which allows processes to join and leave the group dynamically (see also Section 4.4.1, Figure 4.1).

### 4.6.1 Protocol description

All protocols require that events are associated with a tag value. Their reliability depends on the estimated number of rounds the protocol allows events to stay in the system. If the number of rounds denoted by $r$ is chosen too small, the number of informed processes will decrease. If $r$ is chosen too large, the system might suffer from multiple deliveries.

**FIFOgossip.** The FIFOgossip protocol combines the FIFO buffering strategy with the dissemination strategy of lpbcast [Eugster et al. 2001a], i.e. it only propagates events which cannot be found in the history buffer. Termination is guaranteed by aborting the dissemination of events whose tag value is greater than $r$. Since the dissemination is expected to terminate in $O(\log n)$ rounds by itself the dissemination should be aborted latest after $r = O(\log n)$, assuming the buffer size is safe and $K \geq 2$.

**ETTgossip.** The ETTgossip protocol uses the same dissemination strategy and the same termination strategy as FIFOgossip. However, the buffer management uses the ETT approach instead.

**ETTBgossip.** The ETTBgossip protocol applies the dissemination strategy based on the balls and bins analysis [Koldehofe 2002], i.e. whenever an event is received

it will be propagated unless its tag value exceeds $r$. The buffer management uses the ETT approach. If many events are propagated with the protocol, a process may receive in a round more events than it can disseminate with one gossip message. In this case, a process gives preference to events with smallest tag values.

**EPgossip.** The EPgossip protocol estimates the potential of events by the number of balls associated with an event. The dissemination is similar to the ETTBgossip approach. However, the dissemination terminates when the protocol receives messages with tag values which are smaller or equal to zero. In case the protocol receives too many events in a round, it gives preference to events whose tag value is largest. The initial choice of the number of balls, when creating a new event, determines the number of rounds the event stays in the dissemination system.

### 4.6.2 Evaluation

The evaluation compares the stated protocols with respect to reliability and multiple deliveries. For reliability we examine two different measures:

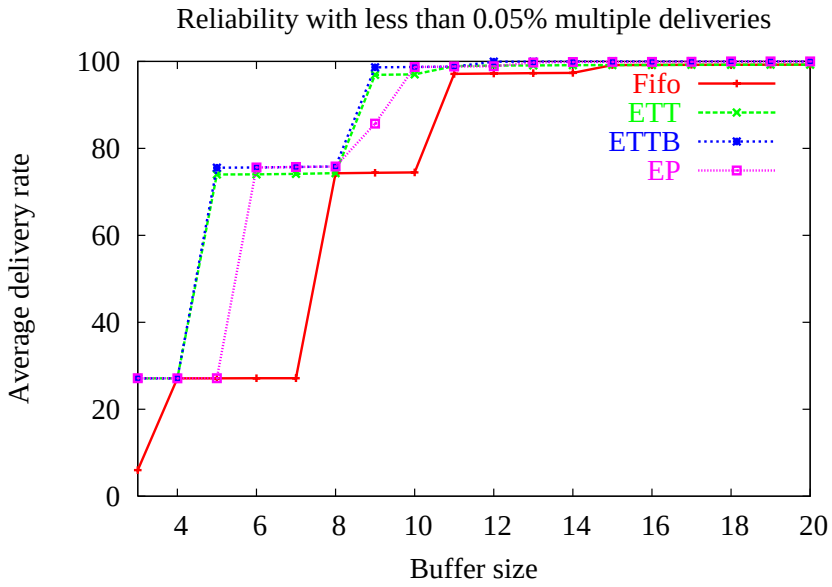
1. counting the average number of processes which are informed about an event,

2. counting the number of events which succeeded to inform all processes.

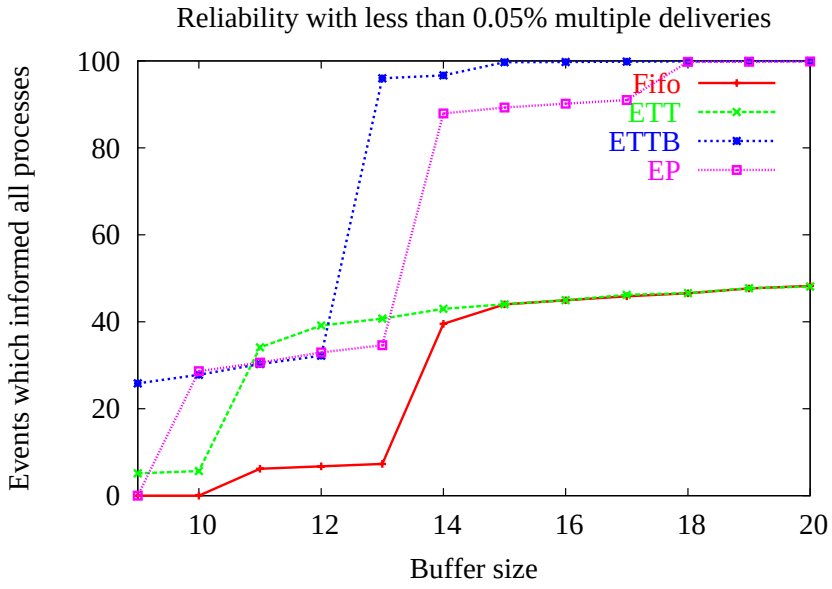
We combine reliability with multiple deliveries by requiring for all events transmitted with the protocol that the number of events which encounter multiple deliveries is small. In our setting less than $0.05\%$ of all events were allowed to encounter multiple deliveries. The comparison of the approaches is based on adjusting the tag value of the respective protocol to achieve best reliability, and respecting the limit for multiple deliveries at the same time.

Figure 4.3 illustrates the differences of the presented approaches for varying buffer sizes regarding the introduced reliability measure. The data is based on a simulation with a system consisting of $100$ processes and using a fan-out of $5$. The probability that a process admitted a new event to the system during a round was $0.01$. Each execution involved approximately $100000$ events.

The data set shows that even for a critical buffer size it is possible to inform a high percentage of processes in contrast to the approach using no tag values of Section 4.4. In the same setting as used for this evaluation, lpbcast encounters many multiple deliveries if the buffer size is chosen smaller than $16$. On the contrary, FIFOgossip can inform, by respecting at the same time the constraint on the occurrence of multiple deliveries, a high percentage of processes (larger than $75\%$) for buffer sizes greater than or equal to $8$. The large differences in reliability between two consecutive buffer sizes, e.g. the difference in reliability for FIFOgossip for




(a)




(b)

Figure 4.3: Reliability of the different approaches with respect to different buffer sizes by requiring that less than 0.05% of events encounter multiple deliveries. For part 4.3(a) reliability was measured by counting the average number of processes informed by the events, while part 4.3(b) measured the amount of events which were received by all processes.

a buffer size of 7 and a buffer size of 8, happen because the increased buffer size allows a longer dissemination of an event without violating the multiple delivery requirement.

The experimental data taken for ETTgossip shows the improvement achieved by managing buffer space with a priority-queue. ETTgossip succeeds to inform significantly more processes than FIFOgossip for buffer sizes smaller than 16. It also succeeds to inform a high percentage of processes for buffer sizes which are greater than or equal to 5. For large buffer sizes FIFOgossip and ETTgossip show the same behaviour.

The best overall reliability, as also expected from the analytical results presented in [Koldehofe 2002], is provided by EPgossip and ETTBgossip, whose dissemination scheme allows a process to propagate an event multiple times. EPgossip succeeds for most instances, and ETTBgossip for all instances to inform a higher percentage of processes than ETTgossip and FIFOgossip. The difference between the two dissemination systems is best illustrated in Figure 4.3(b), showing the percentage of events which were propagated to all processes of the system. EPgossip and ETTBgossip succeed with increasing buffer size for almost all events to inform all processes while ETTgossip and FIFOgossip succeed for less than half of all events, even if the buffer size grows very large. The dissemination system used in FIFOgossip and ETTgossip is only suited to inform a high percentage of processes. On the contrary, for ETTBgossip and EPgossip every event has a high probability to inform every group member.

Compared to ETTBgossip, the reliability for EPgossip is lower when the buffer size is small. In EPgossip a process may be informed by multiple processes in the same round which causes a process to increase the tag value of the respective event. This way the dissemination of an event takes slightly longer time and increases the vulnerability of the protocol to multiple deliveries.

## 4.7 Conclusions

This work has proposed a model suitable to analyse the buffer size in lightweight probabilistic dissemination protocols to avoid multiple deliveries to the application. A bound is proposed that gives a guarantee for a given buffer size that an event will be delivered at a destination multiple times. It is also shown that events should carry information which allows to terminate the dissemination in time since the size of groups can vary over time. The evaluation of dissemination systems shows that approaches may encounter a high amount of multiple deliveries and events may stay very long in the system if the termination of the dissemination relies on the history buffer only. This holds even if the buffer size is chosen almost safe initially. More-

over, buffer management based on priority-queues, in which events are associated with their potential to inform other processes, improve reliability and resistance to multiple deliveries compared to approaches which rely on FIFO buffering. The choice of the dissemination system also influences the reliability. Among the introduced approaches ETTBgossip, based on an analytical evaluation of a balls and bins model [Koldehofe 2002], has shown the highest overall reliability, especially if it is required that almost every event should reach all destinations.

The current analysis provides a general bound for multiple deliveries with respect to a given buffer size. A way to improve the understanding of the relation between reliability and multiple deliveries is to integrate the proposed analysis into the epidemic model [Bailey 1975]. The analysis could support gossiping protocols to adapt the buffer sizes of processes and the dissemination system to improve reliability and avoid multiple deliveries at the same time.

## Acknowledgements

Thanks a lot to Patrick Thomas Eugster for discussing the relation between reliability and buffer sizes in lpbcast, and this way motivating the study of the multiple delivery problem. Special thanks belong to Marina Papatriantafilou and Philippas Tsigas for helpful comments, discussions and suggestions that contributed to this work a lot.

# Five

# Lightweight Causal Cluster Consistency

Anders Gidenstam Boris Koldehofe

Marina Papatriantafilou Philippas Tsigas

**Abstract**

Within an effort for providing a layered architecture of services for middleware supporting multi-peer collaborative applications, this paper proposes a type of consistency management called *causal cluster consistency* which is 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 educational, 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. In a cluster consistency protocol a privileged dynamic set of processes, called coordinators, may concurrently propose updates to a subset of objects which form a cluster. The updates are applied in some order of interest by the coordinators of the cluster. Moreover, any interested process can receive update messages referring to replicated objects, with an option for the updates to be delivered unordered or in the same order as to the coordinators.

This work also describes a protocol implementing causal cluster consistency, which provides a fault tolerant and dynamic membership algorithm to manage the cluster members. The membership algorithm also coordinates the dynamic assignment of process identifiers to vector clock entries. Hence,

this protocol provides optimistic causal order in combination with any group communication protocol. We evaluate the performance of causal cluster consistency running on top of decentralised probabilistic protocols on support for group communication. These protocols scale well, impose an even load on the system, and provide high-probability reliability guarantees for events to be delivered to every process in the group.

**Keywords:** *large-scale group communication, consistency, collaborative environments, middleware, peer-to-peer communication*

## 5.1 Introduction

Many applications like collaborative environments allow a possibly large set of concurrently joining and leaving processes to share and interact on a set of common replicated objects. Processes deal with state changes of objects by sending update messages. Providing the infrastructure and middleware to support such systems places demands for multi-peer communication, with reliability, time, consistency and scalability guarantees, even in the presence of failures and variable connectivity of the peers in the system. Towards designing a set of service layers and lightweight distributed protocols to provide appropriate dissemination and coordination/consistency services for multi-peer collaborative systems to use, we introduce a consistency model which builds on scalable information dissemination schemes that provide probabilistic reliability guarantees. The required methods for communication and consistency must deal with users (processes) who may join and leave the system in several ways and also form groups which vary in size and behaviour; the interests and needs of these users may also vary with time.

Lightweight non-hierarchical solutions, which have good scalability potential at the cost of probabilistic guarantees on reliability have not been in the focus of earlier research in distributed computing, where the emphasis has been in proving feasible, robust solutions, rather than considering the aforementioned variations in needs and behaviour.

In this paper we look at a delivery service for updates which provides delivery in *optimistic causal order*. This guarantees that an event will only be delivered if it does not causally precede an already delivered event. Events which are about to 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 detect which events that are missing many algorithms rely on the use of vector clocks. These, however, grow linearly with the group size which implies a limitation on the size of process groups with respect to scalability.

This work addresses this issue by proposing a consistency management denoted by *causal cluster consistency*, providing optimistic causal delivery of update mes-

sages to a large set of processes. Causal Cluster Consistency takes into account that for many applications the number of processes which is 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 which are entitled to perform updates at the same time is restricted to $n$, which corresponds to the maximum size of the vector clocks used. However, the set of processes entitled to perform updates may change dynamically.

We present a dynamic and fault tolerant cluster membership algorithm which can be used to implement causal cluster consistency on top of any group communication protocol, which does not need to offer strict reliability guarantees. We have implemented the causal cluster consistency algorithm and we present an experimental evaluation of its potential with respect to 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.

The proposed cluster membership algorithm allows several extensions which go beyond causal cluster consistency. It is also possible to use the stated algorithm to distribute ownership of replicated objects and achieve types of consistencies which guarantee that all updates with respect to an object are observed in the same order by all group members.

**Structure of the paper.** In Section 5.2 we present background information and related work on causal ordering and lightweight probabilistic group communication. In Section 5.3 notation and definitions are given. Section 5.4 introduces and evaluates a layered architecture for achieving causal delivery. The dynamic cluster management is described and analysed in Section 5.5.

## 5.2 Background

Many systems like collaborative environments (e.g. [Miller and Thorpe 1995; Greenhalgh and Benford 1997; Carlsson and Hagsand 1993]) allow multiple processes to perform updates on shared replicated objects. In order to perform well for many processes, such systems rely on a middleware which provides scalable group communication, supports maintenance of membership information as well as fast dissemination of updates (*events*) in the system. Applications building on such systems would benefit from an event delivery service that satisfies the causal order relation, i.e. satisfies the "happened before" relation as described in [Lamport 1978]. A lot of work has been carried out achieving reliable causal delivery in the

occurrence of faults [Birman and Joseph 1987b; Birman et al. 1991; Raynal et al. 1991; Kshemkalyani and Singhal 1998]. However, the schemes to recover lost messages can lead to long latencies for events, while often short delivery latencies are needed. In particular the latency in large groups can become large since, in the worst case, a causal reliable delivery service needs to add timestamp information to every event, whose size grows quadratically with the size of the group.

Relaxed requirements, such as optimistic causal ordering as in [Baldoni et al. 1998; Rodrigues et al. 2000] can be suitable for systems where events are associated with deadlines. While the causal order semantics require that an event is only delivered after all causally preceeding events have been delivered, optimistic causal order only ensures that no events are delivered which causally precede an already delivered event. However, optimistic causal delivery algorithms aim at minimising the number of lost messages.

Recent approaches for information dissemination use lightweight probabilistic group communication protocols [Birman et al. 1999; Eugster et al. 2001a; Ganesh et al. 2001; Koldehofe 2003; Pereira et al. 2003; Baehni et al. 2004]. These protocols allow groups to scale to many processes by providing reliability expressed with high probability. In [Pereira et al. 2003] 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.

Vector clocks [Mattern 1989] allow processes to determine the precise causal relation between pairs of events in the system. Further, one can detect missing events and their origin. Since the size of the timestamps grow linearly with the number of processes, to ensure scalability, one may need to introduce some bound on the growing parameter.

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. [Stoica et al. 2001; Alima et al. 2003; Ratnasamy et al. 2001; Rowstron and Druschel 2001; Zhao et al. 2004]), as well as a recent position paper for atomic data access on CAN-based data management [Lynch et al. 2002].

## 5.3 Notation and problem statement

Let $G = \{p_1, p_2, \ldots\}$ denote a group of processes with ongoing joining and leaving of processes 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 [Eugster et al. 2001a; Ganesh et al. 2001; Koldehofe 2003] would be suitable. We refer to such protocols as *PrCast*. PrCast is assumed to provide following properties:

- An event is delivered to all destinations with high probability.
- 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 [Ahamad et al. 1995]. 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 $t_{e_2} < t_{e_1}$.

Throughout the paper it is assumed that each process $p$ maintains for each cluster $C$ a logical vector clock denoted by $clock_p^C$. A 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 5.4 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.

## 5.4 Layered architecture for optimistic causal delivery

This section proposes and studies a layered protocol for achieving optimistic causal delivery which can be combined with the dynamic cluster management algorithm presented in Section 5.5. In this section we assume that coordinators of a cluster are statically assigned to vector entries. Further, all coordinators of a cluster are assumed to know each other. Section 5.4.1 introduces the two layers of the protocol while Section 5.4.2 presents an empirical study which evaluates i) the effect of clustering with respect to scalability, ii) the gain in term of message loss and the overhead induced by providing optimistic causal order delivery.

### 5.4.1 Protocol description

The first of the two layers uses *PrCast* in order to multicast events inside the cluster. The second layer implements the optimistic causal delivery service. The protocol (cf. pseudocode description of Algorithm 1), is an adaptation of the causal consistency protocol by Ahamad et. al. [Ahamad et al. 1995] and is described here: Each process in a cluster interested in observing events in causal delivery (which is always true for a coordinator), maintains a queue of events denoted by $H_p$. For any arriving event $e$ one can determine from $T_p^C$ and the corresponding timestamp $t_e$ whether there exist any events which (i) causally preceed $e$, (ii) have not been delivered, and (iii) are still deliverable according to the optimistic causal order property. If there exist any such events, $e$ will be enqueued to $H_p$ for a time interval until $e$ may become obsolete. The time before $e$ becomes obsolete, depends on the delay of $e$, i.e the amount of time passed since the start of the dissemination, and is assumed to be larger than the duration of a PrCast and the time it takes sending a recovery message and receiving an acknowledgement. If an event in $H_p$ has a delay larger than the maximum length of a PrCast (which is estimated by the number of hops that an event needs before it has reached all destinations with w.h.p. using PrCast), $p$ tries to recover all optimistic deliverable events using reliable point to point communication, i.e. for each event which still needs to be recovered it contacts the source of the PrCast. Before $e \in H_p$ becomes obsolete, $p$ delivers $e$, all causally preceeding events in $H_p$, and causally preceeding recovered events by respecting their causal relation.

When a process $p$ delivers an event $e$ referring to cluster $C$, the logical vector clock $\mathit{clock}_p^C$ is updated by setting $T_p^C = t_e$. Process $p$ also checks whether any events in $H_p$ or recovered events can be dequeued and delivered.

When a coordinator $p$ in $\mathit{Core}_C$ which owns the $j$th vector entry, multicasts an event it will update $\mathit{clock}_p^C$ and increment $\mathit{seq}_p^C$ by one. Then, $p$ creates a vector timestamp $t$ with $t[i] = T_C^p[i]$ for $i \neq j$ and $t[j] = \mathit{seq}_p^C$. Since PrCast delivers

Algorithm 1: Two-Layer protocol

**VAR**
  $pred_e$: set of causally preceeding and deliverable events which were not requested to be recovered
  $R$: set of deliverable recovered events
**On** $p$ creates $e$
  Create timestamp $t_e$
  Prcast$\langle e, t_e \rangle$
  Buffer $e$ until no more recoveries expected
**On** $p$ receives $\langle e, t_e \rangle$
  **if** $e$ can be delivered **then**
    deliver($e$)
    deliver($e' \in H_p$) for $e'$ that can be delivered
  **else**
    delay($e$, time_to_terminate)
  **end if**
**On** timeout($e$, time_to_terminate)
  $\forall e' pred_e$ send $\langle$*RECOVER*, source($e'$), eid$\rangle$
  delay($e$, time_to_recover)
**On** timeout($e$, time_to_recover)
  deliver($e' \in pred_e \cap (H_p \cup R)$) with $e'$ can be delivered
  deliver($e$)
  deliver($e' \in H_p$) with $e'$ can be delivered
**On** $p$ receives $\langle$*RECOVER*, source($e'$), eid$\rangle$
  **if** $p$ has $e$ with identifier eid in its buffer **then**
    respond( $\langle$*ACKRECOVER*, $e, e_t\rangle$ )
  **end if**
**On** $p$ receives $\langle$*ACKRECOVER*, $e, e_t\rangle$
  **if** $e$ can be delivered **then**
    deliver($e$)
    deliver($e' \in R \cup H_p$) with $e'$ can be delivered
  **else**
    $R = R \cup \{e\}$
  **end if**

events with high probability $p$, there may be processes which will request to recover $e$ from the sender. For this purpose $p$ maintains a recovery buffer in which it stores $e$ until no more recovery messages are expected (this is for example the case if $\forall i\ t_e[i] < T_p^C[i]$).

**Properties of the protocol.** The PrCast protocol provides a delivery service which guarantees that an event will reach all its destinations with high probability, i.e. PrCast can achieve high message stability. However, if recovery of events needs to be performed, the number of processes which did not receive the event is expected to be low. This means that a process is only expected to receive a low number of recovery message when it multicasts an event. If processes do not encounter process failures and link failures, reliable point to point communication succeeds in recovering all causally preceding missing events, and thus provide causal order without any message loss. The following lemma is straightforward, following the analysis in [Ahamad et al. 1995].

**Lemma 5.4.1** *An execution using the two-layer protocol guarantees causal delivery of all events disseminated to a cluster if neither processes nor links fail.*

**Increasing throughput and fault tolerance.** The throughput and fault tolerance of the scheme can be increased by adding more redundancy. In this case processes are required to keep a history of some of the observed updates for a bounded time interval of length $I$. A process sending a *RECOVER* message needs only to contact a fixed number of processes in order to receive the respective update . Further, the recovery of failing processes could make use of such redundancy. Analysis similar to the one in [Koldehofe 2003] can give a buffer size and bound the length of the time interval $I$ for which PrCast can guarantee the availability with high probability.

### 5.4.2 Experimental evaluation

The experiments described in this section study the effect of clustering with respect to scalability. Moreover, we evaluate the effect of using the causality layer by comparing the gain in message loss and overhead induced by the causality layer. Hereby we consider a message lost if it was not received or could not been delivered without violating the optimistic causal order.

The evaluation of the two-layer protocol is based on data received from a network experiment on up to 125 computers at Chalmers University of Technology. The computers were running the Linux or Solaris 9 operating systems. The hardware varied among the machines (Sun Ultra10 and Sun Blade workstations, PC's with Pentium4 processors, multiprocessor PC's, all with memory ranging from

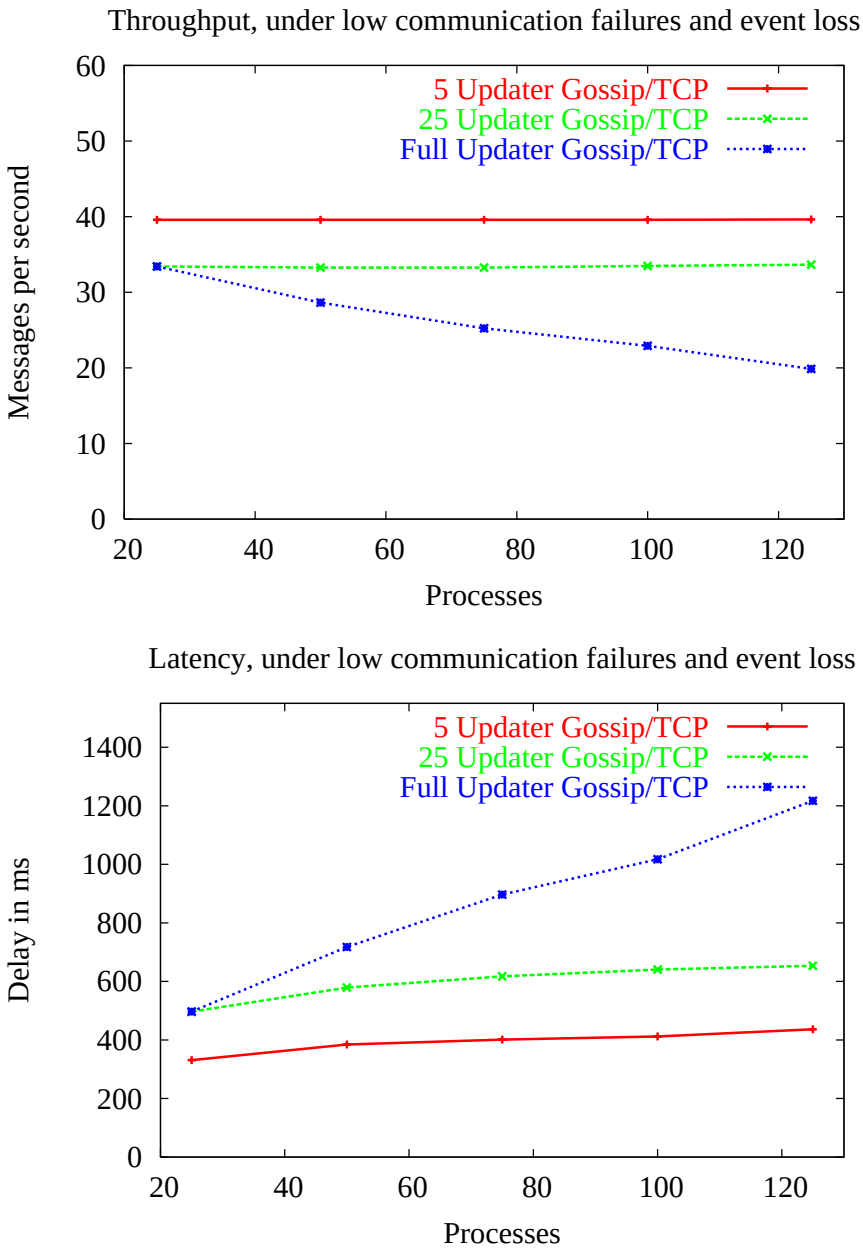



Figure 5.1: Throughput and latency with increasing number of group members

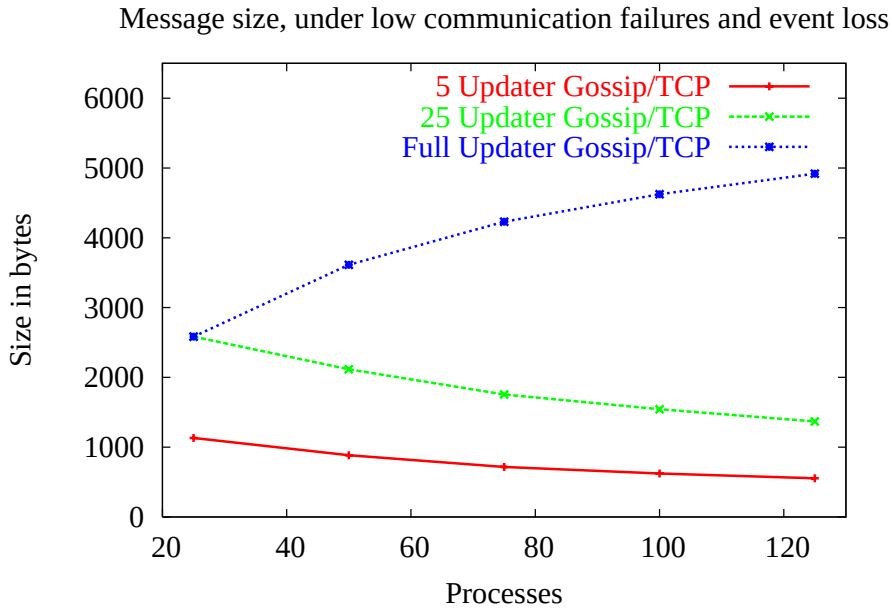



Figure 5.2: Message Size with increasing number of group members

128MB up to 4048MB). The computers were distributed over a few different subnetworks of the university network. Depending on the network load a IP-ping message of 4KB size had an average round-trip-time between 1ms and 5ms. As we did not have exclusive access to the computers and the network, the experiment had to coexist with other users concurrently running applications which potentially might have made intensive use of the network.

We have implemented the layered architecture in an object oriented, modular manner in C++. The implementation of the causality layer follows the description in Section 5.4.1 and can be used with several group communication objects within our framework. The PrCast implementation we use follows the outline of the ETTB-dissemination algorithm described in [Koldehofe 2003] together with the membership algorithm of lpbcast [Eugster et al. 2001a]. The message transport can use either TCP or UDP. For connection oriented communication a timeout can be specified to ensure that a communication round has approximately the same duration for all processes.

Our first experiment evaluates how the number of group coordinators effect throughput, latency and message sizes. The results are based on a test application which can act either as a coordinator or as an ordinary group member. The test application runs in rounds of the same duration as the PrCast protocol does. When

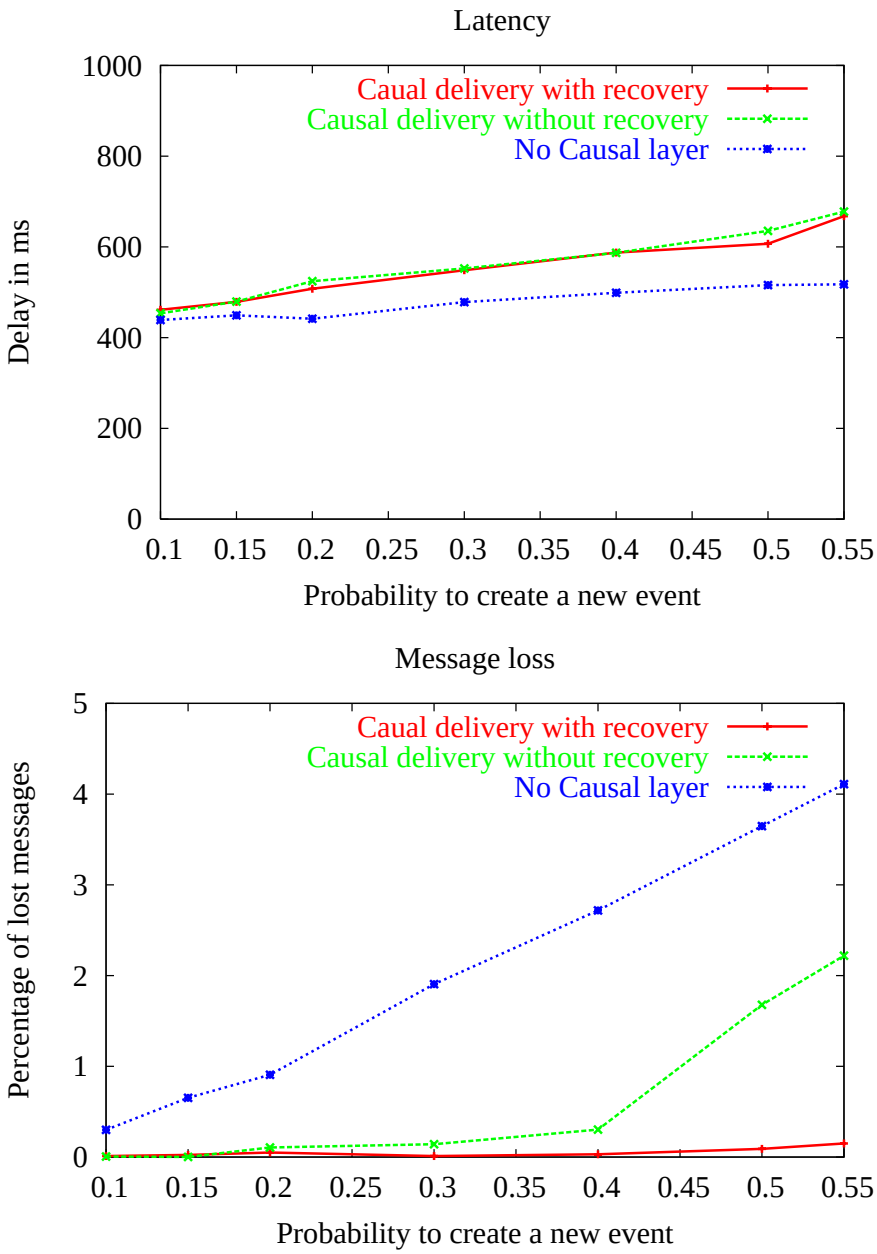



Figure 5.3: Overhead and gain in message loss by using the causality layer

the process acts as an updater it produces a new event with probability $p$. In our experiments the product of updaters and $p$ was kept constant (at $6$). The configuration parameters of PrCast were set to satisfy the goal of each event reaching up to $250$ processes with high probability. The fanout was $4$, while the termination time for an event was $5$ hops. PrCast was allowed to keep track of all members to avoid side effects of the membership scheme in this experimental study. The maximum number of events which can be transported in one gossip message was limited to $20$ events, while the number of group members transported was limited to $10$. The size of the history buffer was set to $40$ events, which is according to [Koldehofe 2003] high enough to prevent with high probability the same event from being delivered to the causality layer multiple times. The gossip messages was sent using connection oriented communication. This allowed us to tune the duration of each gossip round so that all experiments had approximately the same rate of failures for communication links. We did not experience a significant improvement by using UDP as the transport layer protocol. In each round a process sends its gossip messages concurrently by using multithreading, so the round time could be adapted to be 10ms longer than the expected maximum message delay.

Figure 5.1 and Figure 5.2 compare three instances of the two-layer protocol. In the full-updater experiment all processes acted as coordinators, while in the $5$-updater and the $25$-updater experiments the number of coordinators was restricted to $5$ and $25$ respectively. The protocols were tuned by adjusting the connection timeout such that the average message loss rate did not exceed $0.2\%$. The data shows the impact of the size of the vector clock on the overall message size. 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 for large group sizes, 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. This behaviour directly influences the observed results with respect to latency and throughput. For growing group size the protocols using a constant set of coordinators experience only a logarithmic increase in message delay and throughput remains constant. The message delay for the full updater protocol increases linearly while throughput decreases.

The second experiment studies the effects induced by the causality layer and the recovery scheme used in the two-layer protocol. Figure 5.3 compares the gossip protocol, the two-layer protocol with and without recovery. The protocol parameters were the same as for our first experiment, however the connection timeout was fixed using 190ms. Instead we evaluated the system for varying values of $p$. The range of values for $p$ shown in Figure 5.3 corresponds to a behaviour of the gossip protocol where a multicast is expected to reach a large set of processes. By using

larger values for $p$ message buffer will overflow, causing the gossip protocol and the causality layer protocols experience a high message loss.

The measurements in Figure 5.3 show that the causality layer reduces the amount of lost messages, especially if the number of events disseminated in the system is high. By using the recovery scheme almost all events could be delivered in optimistic causal order. With increasing event probability latency grows only slowly. The causality layer adds a small overhead by delaying events in preference to respect the optimistic causal order. The recovery scheme does not add any overhead with respect to latency, however it significantly reduces the number of lost messages. For some instances it was even possible to observe a slightly faster message delivery when using the recovery scheme. One possible reason is that recovery of multiple events at a time help to dequeue events faster from $H_p$ and thus allow a faster event delivery.

## 5.5 Dynamic cluster management

Recall that in the previous section the set of coordinators $Core_C$ was predefined and static. In dynamic environments the interest of processes may change over time. Therefor it is crucial that processes are able to join and leave the cluster. Compared to ordinary group communication membership management we will distinguish between two different ways of joining and leaving a cluster.

**Ordinary joining/leaving 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 in optimistic causal order, but will not be able to send events to the cluster.

**Coordinator is joining or leaving.** 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 coordinate a unique identifier corresponding to the vector clock entry of $clock_p^C$. When $p$ performs a *cleave* operation it release its coordinated vector clock entry and cannot send any more events to the cluster after that. The vector clock entry 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 vector clock entry within the cluster at the same time. The Vector clock entry 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 *join* 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.

In Section 5.5.1 we show how to achieve decentralised cluster management and outline a method where every coordinator of the cluster manages a subset of vector clock entries. We present two protocols: the protocol of Section 5.5.2 which works in the absence of failures and illustrates the basic idea, while Section 5.5.3 describes and proves a fault-tolerant membership protocol.

### 5.5.1 Decentralised 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 of the cluster the coordinator of a subset of the vector clock entries $\{0 \dots n-1\}$. We will ensure that there are never two processes that simultaneously own and coordinate the same vector clock entry. Once a process is detected as faulty and there exists sufficiently many non-faulty processes, another process will eventually become coordinator for the set of vector clock entries previously managed by the faulty process.

Let $i$ be process $p$'s own vector clock entry. A process $q$ which owns vector clock entry $j$ is defined to be the successor of $p$ if $\exists k$ s.t. $j = i - k \bmod n$ and $\forall l$ with $1 \leq l < k$ the $l$th vector clock entry is not owned by any process. Accordingly, $q$ is called the predecessor of $p$ if $\exists k$ s.t. $j = i + k \bmod n$ and $\forall l$ with $1 \leq l < k$ the $l$th entry is not owned by any process. Further, a process is called the $d$th closest successor (predecessor) of $p$, if the process is reachable in $d$ steps from $p$ by following the chain of successors (predecessors) starting at $p$.

We define the set of vector clock entries which is coordinated by a process in terms of successor and predecessor. Let $p$ and $q$ denote two processes owning vector clock entries $i$ and $j$ respectively and let $q$ be the successor of $p$. Further, let $S_p$ denote the set of vector clock entries coordinated by $p$. We define

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

Figure 5.4 gives an example of how processes maintain and coordinate vector clock entries, e.g. $p_2$ owns entry $4$ and coordinates the vector clock entries $\{2, 3\}$.

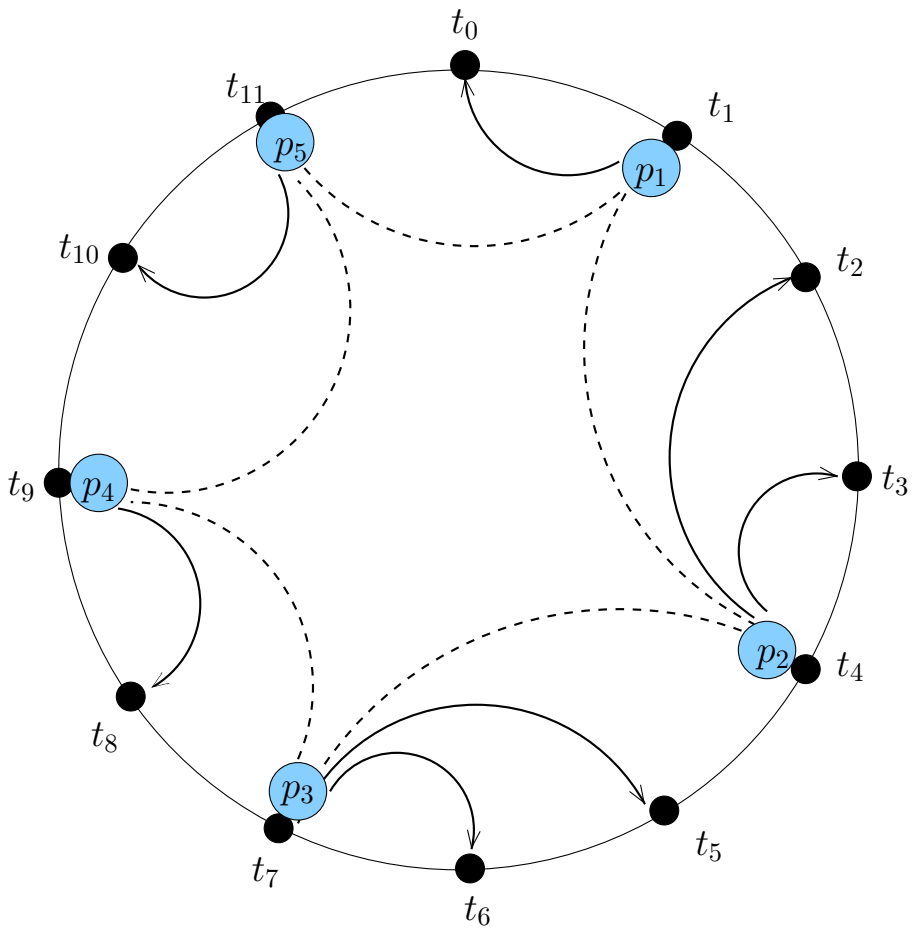



Figure 5.4: Illustration on how processes maintain and coordinate vector clock entries. An arrow from process $p_i$ to a vector element indicates that $p_i$ is the respective coordinator.

**Lemma 5.5.1** *Let $C$ denote a non-empty cluster ensuring that no two processes in the cluster coordinate the same clock entry, then*

1. *for any pair of processes $p, q \in \text{Core}_C$ with $p \neq q$ $S_p \cap S_q = \emptyset$,*
2. *every entry of the cluster's vector clock is either coordinated or owned.*

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

### 5.5.2 A protocol working in the absence of failures

Let us consider the case where all processes are non-faulty and there exist no link failures. A process $p$ performing a *cjoin* operation contacts an arbitrary coordinator of *Core*$_C$. This coordinator will decide on a suitable coordinator $q$ with successor $r$, to which it forwards the *cjoin* operation. Determining $q$ can be achieved by selecting the coordinator of an available uniformly at random chosen vector entry. When $q$ receives the forwarded *cjoin* request of $p$, $q$ will first serve all previous *cjoin* and *cleave* operations it received by other processes. $q$ will reject *cjoin* of $p$ if either $q$ itself has started a *cleave* operation before receiving *cjoin* of $p$ or $q$ coordinates only the vector entry it owns. If $q$ decides to serve the *cjoin* request of $p$, it assigns an entry $i \in S_q$ to $p$ (possibly reflecting the random choice when determining $q$ as a suitable coordinator). $q$ selects $p$ as its new successor, while $p$ selects $q$ as its predecessor and $r$ as its new successor. $p$ also informs $r$ about the state change. Process $r$ will acknowledge if it did not perform a *cleave* operation. Processes $p$ and $q$ may not perform a *cleave* operation or serve *cjoin* operations until receiving an acknowledgement or a *cleave* request by $r$. Process $p$ also PrCasts that it became a coordinator in *Core*$_C$ and that it owns entry $i$. 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 vector clock entry.

Any process $p$ with successor $r$ performing a *cleave* operation will contact its predecessor $q$ to leave. $q$ will reject a *cleave* if it announced itself a *cleave* operation before. Otherwise, $q$ processes first all previous *cjoin* and *cleave* operations. If by then $q$ knows $p$ as its successor it will acknowledge $p$ leaving. Further, it will inform $r$ about the state change. $q$ only proceeds serving *cjoin* after it receives an acknowledgement of $r$. $p$ PrCasts to leave $i$ and leaves the cluster after receiving acknowledgements from $q$ and $r$ .

In order to verify correctness of the protocol as stated in Theorem 5.5.1, recall that according to Lemma 5.5.1 correctly preserving the relation among successors and predecessors, given by Lemma 5.5.2, suffices to guarantee unique assignment of processes to vector clock entries.

**Lemma 5.5.2** *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 successors and predecessors,*

2. *an interleaving* cleave *operation of $r$ will successfully be managed at $p$ and therefore preserve correctly the predecessor successor relation of* Core$_C$.

**Theorem 5.5.1** *Let $\Sigma := \sigma_1, \ldots, \sigma_m$ denote a sequence of potentially interleaved operations on a cluster $C$ where $\sigma_i$ corresponds to a* cleave *and* cjoin *operation. If $\Sigma$ maintains* $\text{Core}_C$ *to include at least one process the algorithm guarantees for any $p, q \in \text{Core}_C$*

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

### 5.5.3 Dealing with link and process failures

In the following we present an algorithm which extends the previous framework of Section 5.5.2 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 $\delta$ 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 $q$ and $M$ is not received in time $\delta$ then both processes will act as if $M$ was never send. Moreover, we say 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 first messages from all $m$ sources.

The proposed 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. 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 $\mathit{Core}_C$ which does not perform a *cleave* operation remains in $\mathit{Core}_C$ 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. Failing processes will eventually be excluded from $\mathit{Core}_C$ and processes which perform *cjoin* subsequently may reuse the respective vector clock entries.

The first invariant is achieved by the processes in $\mathit{Core}_C$ sending *ALIVE* messages to their $2k + 1$ successors in each round. A process which receives less than $k + 1$ *ALIVE* messages during a round considers itself to have failed and immediately leaves $\mathit{Core}_C$.

Algorithm 2: Decentralised and fault tolerant cluster management: Data Structure and Initialisation

**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 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 |
| $P_{\text{exclude}}$: | probability to start exclusion algorithm after weakly detecting a faulty successor |

**Message types:**

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

**Init$_p$:**

Send $\langle \text{CJOIN}, p \rangle$ to a known coordinator in *Core*$_C$.

**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 $\langle \text{ALIVE}, p \rangle$ to $2k+1$ closest successors in Cview$_p$.

Algorithm 3: Decentralised and fault tolerant cluster management: Main Loop

**Main loop of the coordinator algorithm**

**Do** in every round (duration longer than PrCast)
  **if** $|\text{ALIVE}_p \cap L_p| < 2k+1$ **then**
    ThinkIamDisconnected $= true$
    exit loop
  **end if**
  Send $\langle \text{ALIVE}, p \rangle$ 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 $\langle \text{UPDATE}, R \rangle$ to $q$
    **if** STATUS is OK **then**
      $R_p = R$
    **else**
      Run exclusion algorithm with probability $P_{\text{exclude}}$
    **end if**
  **else**
    $\text{TempRounds}_p = \text{TempRounds}_p - 1$
  **end if**

**Handling of *UPDATE* messages**

**On** receiving $\langle \text{UPDATE}, R \rangle$
  $L_p = R$

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 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 vector clock entry $i$ .

In each round $p$ proceeds if it has received during a round at least $k+1$ *ALIVE* messages from processes in $L_p$, otherwise $p$ considers itself to have failed. 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 $L_p = L'_p$.

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 $2\delta$ 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 $p$
  **end if**

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 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 vector clock entries 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 preceeding $q$ 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 it considers itself to have 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 a *cjoin* request. All processes in $E_{pq}$ which neither have failed nor considered themselves 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 vector clock entry since in every round the closest alive process in $E_{pq}$ is guaranteed to consider itself to have failed at the end of the round.

Processes which are requested to acknowledge an exclusion interval $E_{pq}$ only acknowledge if their vector clock entry 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 dist$(p,q)$ rounds.

### 5.5.4 Correctness and analysis of the membership protocol

In order to prove correctness of the membership algorithm of Section 5.5.3, 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 5.5.3 we first consider the behaviour of the algorithm when no failures occur.

**Lemma 5.5.3** *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 5.5.2.*

**Proof.** Both algorithms show only different behaviour if $p$ executing Algorithm 3

weakly detects its immediate successor $r$ to be faulty. Since neither processes, nor links do fail $p$ must have detected $r$ as faulty because $r$ has considered itself to be faulty. 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 5.5.4 states that during a round the closest successor in $E_{pq}$ will fail.

**Lemma 5.5.4** *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*

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

**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 effect $L_p$ once $p$ has joined the cluster. Hence, all vertices in $L_r - E_{pq} - R_p$ are not alive after $p$ determined its set $R_p$.

*Case $r$ did not receive the respective token:* There must be an ancestor which re-

ceived 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 all processes in $L_r - E_{pq} - R_p$ are not alive, once $p$ has updated $R_p$. □

Lemma 5.5.4 immediately implies Corollary 5.5.1 which states how long a process $p$ needs to be temporary coordinator until at least $i$ alive processes in $E_{pq}$ have failed.

**Corollary 5.5.1** *The $i$th successor of $p$ in $E_{pq}$ will fail latest $i$ rounds after $p$ was acknowledged.*

**Proof.** The immediate successor of $p$ clearly fails because all *ALIVE* messages $r$ can expect according to Lemma 5.5.4 are 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 $i$th successor. □

**Theorem 5.5.2** *Algorithm 3 guarantees that two processes never have common vector entries they either own or coordinate.*

**Proof.** Lemma 5.5.3 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 vector entries. 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 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 5.5.1 guarantees that $r$ must have considered itself to have failed, contradicting that both $r$, and $s$ were active. □

## 5.6 Discussion and future work

We have proposed lightweight causal cluster consistency, a hierarchical layer-based middleware structure for multi-peer collaborative applications. Causal cluster consistency provides a dynamic interest management for processes on replicated objects. Processes can observe updates which correspond to their interest in optimistic

causal order. Moreover, a dynamic set of processes may concurrently propose updates to the state of the replicated objects. We have shown an implementation based on scalable group communication protocols. The protocol uses a fault tolerant and decentralised membership algorithm which has been proven to correctly provide optimistic causal ordering and succeeds in excluding faulty processes.

The proposed cluster membership algorithm allows several extensions which go beyond causal cluster consistency. It is also possible to use the stated algorithm to distribute ownership of replicated objects and achieve types of consistency which guarantee that all updates with respect to an object are observed in the same order by all group members.

Further work includes complementing this service architecture with other consistency models such as total order delivery with respect to objects.

# Six

# A Role-Based Distributed Hash Table

Sébastien Baehni Rachid Guerraoui Sidath B. Handurukande
Oana Jurca Boris Koldehofe

**Abstract**

Research on probabilistic reliable peer-to-peer communication has shown how to provide lightweight scalable and decentralised membership which is highly resilient to failures. For topic-based publish/subscribe several solutions building on structured and unstructured peer-to-peer networks have been proposed. A problem in existing solutions is to provide a fair balance of the load during the dissemination of events which considers the interest of processes. In this work we address this issue by proposing a role-based DHT which supports dynamic subscriptions and unsubscriptions to topics in a space efficient manner. A role-based DHT allows one to separate subscription management and dissemination, and it supports a dissemination algorithm in achieving a fair balance in forwarding events. Moreover, the role-based subscription management can eliminate the use of parasite messages, i.e. messages propagated by a process which does not share an interest in the content of the message. We present an algorithm implementing a role-based DHT, which can be easily combined with existing DHT algorithms, and present an evaluation in combination with two fundamental dissemination schemes: application-level multicast trees and gossiping. The results show that a role-based DHT can support a well balanced application-level multicast trees as well as support gossip-based dissemination algorithms in efficiently managing data structures.

**Keywords:** *publish-subscribe, structured and unstructured peer-to-peer dissemination, group communication*

## 6.1 Introduction

Large-scale event dissemination supporting many-to-many communication is a fundamental service needed in many distributed applications to support coordination among processes. With increasing number of processes the amount of concurrent events increases. A needed feature in providing scalability and reducing the number of events delivered per process is to support *selective event dissemination* considering the interest of processes. Processes in the network are divided into two possibly overlapping camps: *publishers* and *subscribers*. The publish/subscribe paradigm allows a process to express interest by performing a subscription. After a successful subscription, a process will be asynchronously notified about all published events corresponding to its interest. Publish/subscribe has been implemented using various ideas structuring the dissemination process including different paradigms like topic-based, content-based and type-based publish/subscribe.

For recent decentralised peer-to-peer applications it is important to distribute the load of the dissemination of events well among all peers in the system. For peer-to-peer based publish/subscribe systems a measure of *fairness* takes into account the load of the work in disseminating events dependent on the expressed interest. Ideally, processes which are rarely notified about events should be allowed to perform less work, while a peer which receives many notification should perform more work for receiving the service supported by many other peers. Since in general it is difficult to predict when publishers create events there is not a direct relation between the interest expressed by a subscription and the amount of events received. Therefore a fair dissemination of an event should avoid involving processes which are not interested in the event. Messages carrying such events are also called *parasite* messages.

Recent work on building scalable publish/subscribe systems uses lightweight decentralised peer-to-peer multicast dissemination to implement topic-based publish/subscribe. Processes express their interest by subscribing to a set of topics. Two common ways of implementing topic-based publish/subscribe are using application-level multicast trees in combination with structured peer-to-peer dissemination [Rowstron et al. 2001; Zhuang et al. 2001] and gossiping in unstructured peer-to-peer systems [Baehni et al. 2004]. In difference to unstructured peer-to-peer systems, structured peer-to-peer systems offer a lookup service which allows one to route efficiently for a given key to its closest location. By using for example uniform hashing it is ensured that each location is maintained by a unique

process. Structured peer-to-peer systems are also called a *distributed hash table* (DHT).

Application-level multicast trees in structured peer-to-peer systems allow an efficient handling of subscriptions and unsubscriptions to topics. Performing a lookup to the name of the topic determines the root of the application-level multicast tree. A drawback, however, is that the process performing as a root is not necessarily interested in the topic, but carries the highest load. Currently proposed solutions for implementing application-level multicast trees lead either to badly balanced trees or involve frequently high level processes close to the root in the application-level multicast tree to enforce a good balance.

On the other hand gossiping in unstructured peer-to-peer systems support fair event dissemination by associating each topic with a gossip group. However, subscriptions to topics are more difficult to achieve, and require to implement an additional structure that allows a process knowing about one process of the peer-to-peer system to subscribe to any topic maintained by the peer-to-peer system. Moreover, associating with each topic a group requires to allocate more space for maintaining for each group a view of group members. Although topic hierarchies can reduce the number of topics a process needs to subscribe, it is desirable to reduce the overall cost of space maintained at a peer depending on the number of topics a process has subscribed.

While dissemination and subscription management commonly are treated in a single structure, this work proposes a *role-based* DHT for subscription management which can support fair and space efficient event dissemination independent of the used dissemination scheme. While the subscription management is responsible in achieving fairness and space efficiency, the dissemination determines the reliability provided for event delivery. We present an algorithm for a role-based DHT which can be implemented in combination with existing DHT algorithms in the literature. The impact of the role-based DHT algorithm is evaluated with application-level multicast trees and gossip-based dissemination. The results show that in combination with a role-based DHT it is easy to achieve well balanced application-level multicast trees by loading all processes evenly. For gossip-based dissemination the role-based DHT is shown to be useful in maintaining random dynamic views for processes.

## 6.2 Background

The publish/subscribe paradigm allows processes to express their interest in events by performing an operation subscribe. After subscription, a process will receive events which correspond to the expressed interest. Publish/subscribe systems may

differ in the way they express interest such as topic-based, content-based, and type-based publish/subscribe schemes (cf. [Eugster et al. 2003]). Large-scale publish/subscribe systems like [Baehni et al. 2004; Rowstron et al. 2001; Zhuang et al. 2001] usually use topic-based publish/subscribe systems. In topic-based publish/subscribe each process subscribes to a list of topics. Any published event is associated with a topic and needs to be delivered to all processes interested in the respective topic. Often publish/subscribe systems consider hierarchies of topics and subtopics, i.e. a process subscribing to a topic will receive all events corresponding to the topic and all subtopics.

The implementation of scalable decentralised topic-based publish/subscribe systems commonly uses group communication to handle interest management in form of group membership and uses the underlying multicast primitive to disseminate events to all group members. While in early research the focus was on reliable fault-tolerant approaches, recent peer-to-peer based solutions trade the strong reliability guarantee with reliability expressed with probabilistic guarantee in order to scale to a large number of processes and provide a highly dynamic membership scheme.

One can distinguish among peer-to-peer solutions for large-scale event dissemination between structured and unstructured peer-to-peer-systems. Structured peer-to-peer systems like [Stoica et al. 2001; Ratnasamy et al. 2001; Rowstron and Druschel 2001; Alima et al. 2003] handle dynamic membership and provide a lookup service which efficiently routes for a given key to its closest location. Such a structure is also referred to as a *distributed hash table* (DHT). Unstructured peer-to-peer solutions providing probabilistic reliability use the gossiping paradigm (cf. [Demers et al. 1987; Birman et al. 1999; Eugster et al. 2001a; Ganesh et al. 2001; Koldehofe 2003; Pereira et al. 2003]). In distributed computing, gossiping was introduced in the context of data replication, whereas with pbcast [Birman et al. 1999] gossiping became an attractive communication paradigm for large-scale event dissemination. In gossiping protocols, a process is assumed to maintain a set of processes forming its view. When disseminating an event processes which have received this event forward the event to a subset of their neighbours. Similar to an infectious disease under certain conditions one can guarantee with high probability that every process of the system receives the respective event.

In publish/subscribe, structured and unstructured peer-to-peer solutions have been proposed. In structured networks, approaches like [Rowstron et al. 2001] form an application-level multicast tree. With help of the peer-to-peer system the application-level multicast tree can be maintained in as self-stabilising way. However, there is a difference in the involvement of processes depending on whether they are leaf or a forwarder in the tree. Moreover, the root of the tree is not necessarily interested in the topic. When building topic hierarchies the root of multicast

tree has to perform the largest amount of work. Distributing the load among the peers fairly is an issue in SplitStream [Castro et al. 2003] and Bullet [Kostić et al. 2003] where the data is divided into smaller items and distributed using several streams. For SplitStream a peer may depending on its interest, subscribe to a number of streams corresponding to its inbound requirement. Efficient ways of splitting data streams by providing fairness and a low amount of overall work has been analysed in [Bickson et al. 2004].

Splitting data in smaller streams is an attractive property in the context of multimedia streams. In the context of publish/subscribe systems, the events are often too small, so that splitting data may be used only to provide redundancy.

Data aware multicast [Baehni et al. 2004] is based on the gossiping paradigm and involves processes corresponding to the topics they have subscribed. Moreover, it considers the use of topic hierarchies. In the dissemination of events only processes are involved which have subscribed to the topic.

Application-level multicast trees involve significantly fewer messages compared to gossiping. However, both schemes also address different reliability properties. For gossiping the issue is to inform all processes with high probability in the presence of failures, while for application-level multicast trees a large set of processes may be temporarily disconnected.

## 6.3 Notation and problem description

A topic-based publish/subscribe system consists of a dynamic set of topics denoted by $T = \{t_1, \ldots, t_m\}$ and a dynamic group of a possibly large number of processes denoted by $G = \{p_1, \ldots, p_n\}$. Topic names as well as processes identifier are assumed to be associated with unique names. For process $p$ we write $p_{id}$ when referring to its identifier. The topics of the publish/subscribe system may form a hierarchical containment relation. A *topic hierarchy* is a partial order relation $H(T) \subset T \times T$ where $(t_i, t_j) \in T$ if all published events corresponding to $t_j$ correspond also to $t_i$. The topic hierarchy can be represented by a directed acyclic graph $D(T, E(T))$ where $(t_i, t_j) \in E(T)$ if and only if $(t_i, t_j) \in H(T)$ and $\forall (t_l, t_j) \in H(T) \Rightarrow (t_l, t_i) \in H(T)$.

Let $p$ be a process in $G$, then $T_p$ denotes the set of topics, $p$ is interested in. In a publish/subscribe system $p$ can perform the following operations:

- *publish*$(t_i, e)$: all processes $q \in G$ with $\exists t_j \in T_q$ s.t. $(t_j, t_i) \in H(T)$ will be notified about the event $e$.

- *subscribe*$(t_1, \ldots, t_l)$: after having performed this operation, $p$ will be notified about events corresponding to topics $t_1, \ldots, t_l$.

- *unsubscribe*$(t_1, \ldots, t_l)$: after having performed this operation, $p$ will not receive any events corresponding to topics $t_1, \ldots, t_l$.

The publish operation requires from a publish/subscribe system to offer an event dissemination algorithm. In addition to providing delivery guarantees to the subscribers, an important issue in a decentralised system is to distribute the load in forwarding events fairly among all subscribers of the publish/subscribe system. In a publish/subscribe system processes should perform work according their interest, i.e. the more services a process consumes the more it should be involved in providing services to other processes. We can measure *fairness* in a publish/subscribe system by considering for each process $p$ in $G$ the ratio denoted by $\phi_p$ over the number of notifications and the number of forwarded events. Let $\mu = \sum_{p \in G} \phi_p / |G|$, then one dissemination scheme is fairer than another if $\Phi := \sum_{p \in G} |\phi_p - \mu|$ becomes smaller.

Another important measure is the amount of *space* allocated at each process for forwarding events. For every topic, a process $p$ performs work in disseminating events, $p$ needs to maintain an additional structure, for instance a routing table, which helps to determine the next locations for forwarding an event.

The role-based DHT, introduced in Section 6.4, provides a subscriptions management scheme which can be used to adjust fairness and space efficiency in combination with decentralised dissemination algorithms. A DHT allows one to map processes to locations and provides a lookup service which returns for a key the closest process maintaining the location. The number of locations in a DHT we denote by $N$. We assume that a DHT uses uniform hashing in order to determine the location of processes. Let $\Sigma, I$ be alphabets and let $\mathit{hash} \in \Sigma^* \rightarrow I^l$ be a function known to all processes mapping a key of arbitrary length (e.g. a group or process identifier) to a domain of fixed length in a way that for two different keys a collision is unlikely to happen and the values appear uniformly distributed among the domain. Let $\mathit{hash}_1(x) = m_1 \in I^{l_1}$ and $\mathit{hash}_2(y) = m_2 \in I^{l_2}$. We define $\mathit{hash}_1(x) \circ \mathit{hash}_2(y)$ to be the concatenation of the strings $m_1$ and $m_2 \in I^{l_1+l_2}$.

In a role-based DHT, a process can use several roles in order to support services regarding a role. In a topic-based publish/subscribe system a role corresponds to a topic. A process can be forwarder or child with respect to a role. For each role a process acts as a forwarder it needs to perform work during the dissemination of events and maintain a location inside the role based DHT. The dissemination with respect to a role only involves those processes which have joined the DHT using the same role.

## 6.4 Topic-aware subscription management

In this section we introduce a structure, we call a *role-based DHT*, which can be used for management of dynamic subscriptions and unsubscriptions in a publish/subscribe system in order to support fairness and space efficient management of resources. A role-based DHT provides

- *interest management*: a process can join the DHT using multiple roles determining its interest,
- *interest-based lookup*: a process can perform a lookup of a location regarding a role, and if the location is maintained by a process using the same role, then the lookup will involve only processes which have joined the DHT using the same role,
- *role-based random lookup*: a process can perform a random lookup with respect to a role which returns a process chosen uniformly at random among all processes which joined the DHT using the same role,
- *load balance*: when performing a join operation a process can join either as a forwarder or as a child. A process which joins as a forwarder needs to maintain space for a role-based lookup table. A child only maintains the information on the parent and receives from its parent all events.

A role-based DHT can be achieved in combination with various routing schemes for DHTs mentioned in the literature, however we need to extend *join* and *lookup* operations to take into account the role aspect. Let $r$ be a role then $\textit{join}_p(r, v)$ allows process $p$ to join the DHT using role $r$ where $v$ decides whether $p$ acts as forwarder or child. The operation $\textit{lookup}_p(r, \textit{value})$ performs a lookup using role $r$. If *value* is maintained by a process using role $r$ the the lookup will not involve any processes sharing role $r$.

In a topic-based publish/subscribe system a role corresponds to a topic. The role-based lookup provides us with a notion of *topic-awareness* which guarantees that a process interested in a topic can route efficiently to any other process interested in the same topic by using no parasite messages.

In the following, we will explain the ideas behind implementing the role-based lookup operations and propose an algorithm which can be used to implement topic-based publish/subscribe considering fairness and space requirements.

### 6.4.1 A role-based lookup supporting topic-awareness

A lookup service requires to map processes uniquely to locations, for instance by using a consistent hash function. Let $p$ denote a process and $p_{id}$ its identifier then *hash*$(p_{id})$ would determine the location inside the DHT. A generalisation which allows processes to determine a location with respect to a role is to use *hash*$(p_{id}, r)$, where $r$ denotes the role. This way a process can join a DHT multiple times using different roles and use different virtual addresses. Any lookup of *hash*$(p_{id}, r)$ is guaranteed to end in $p$. If $q$ forwards a lookup operation to $p$, $q$ will add to its request the virtual location *hash*$(p_{id}, r)$. $p$ will serve the lookup by using a routing table corresponding to $r$. For maintenance of its roles $p$ uses a set $L_p$, called the *location view* of $p$. $L_p$ is defined by

$$L_p := \{\textit{hash}(p_{id}, r) \mid p \text{ joined the DHT using role } r\}.$$

For each $v \in L_p$, p maintains a routing table $R_p(v)$. The implementation of $R_p(v)$ can be done using existing DHT protocols of the literature. We assume $R_p(v)$ provides a set of neighbours $Q_p(v)$ which have the closest common prefix for values $x > v$ as well as for $x < v$. After performing a lookup of $w$ using $R_p(v)$, $R_p(v)$ will point to a forwarder with value $u$ whose common prefix is larger or equal to the common prefix of $w$ and $v$.

A role-based lookup can be used to achieve a subscription management algorithm for topic-based publish/subscribe which provides topic-awareness and maintains only a small location view. As a first step we propose a hash function which can be used to implement topic-awareness. From Lemma 6.4.1 we can obtain *hash*$(p_{id}, t)$ such that for any pair of processes $q$ and $r$ with *hash*$(q_{id}, t) \in L_q$ and *hash*$(r_{id}, t) \in L_r$, $q$ and $r$ can route to each other without using any parasite messages.

**Lemma 6.4.1** *Assume that for any pair of topics* $t, t' \in T$ hash *is collision free, i.e.* $\text{hash}(t) \neq \text{hash}(t')$. *Let* $p$ *denote a process in* $G$ *interested in* $t \in T$. *If* $\text{hash}(p_{id}, t) := \text{hash}(t) \circ \text{hash}(p_{id})$ *and* $\text{hash}(p_{id}, t) \in L_p$ *then* $p$ *can lookup any other process* $q$ *with* $\text{hash}(q_{id}, t) \in L_q$ *without using any parasite messages.*

**Corollary 6.4.1** *A DHT which guarantees that* $\forall p \in G, \forall t \in T_p$ $\text{hash}(p_{id}, t) \in L_p$ *provides topic-awareness.*

Corollary 6.4.1 follows immediately from Lemma 6.4.1 and gives a first implementation to achieve topic-awareness: for every subscription to topic $t$, $p$ keeps an entry in its location view, i.e. *hash*$(p_{id}, t)$ exists in $L_p$.

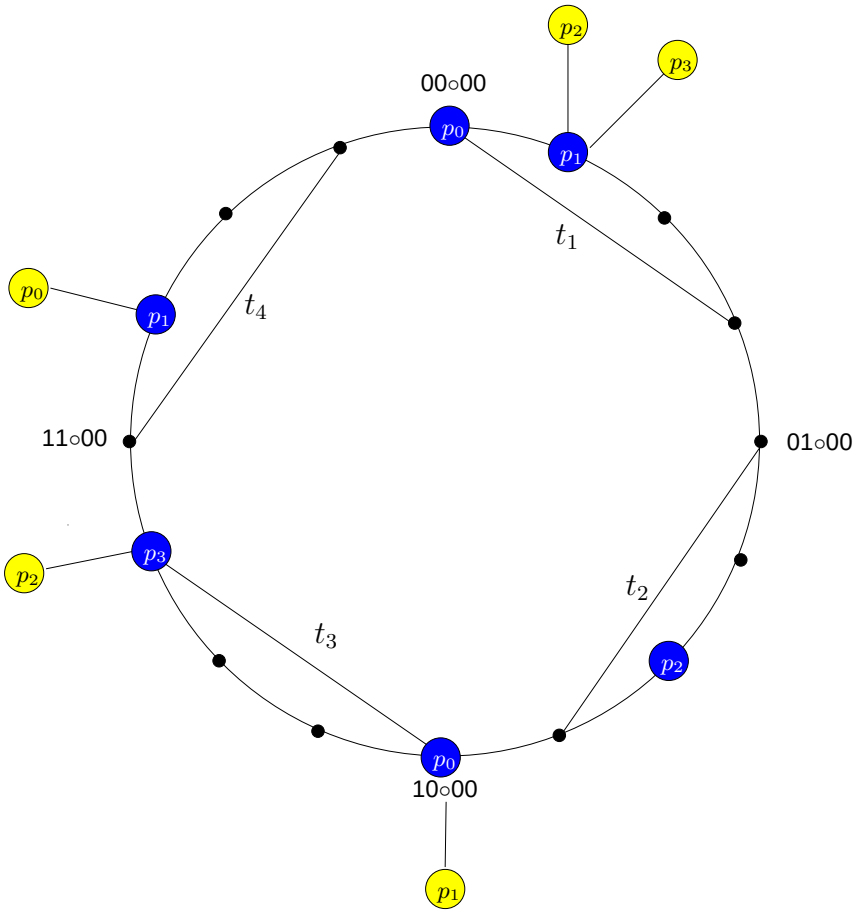



Figure 6.1: Forwarders and children in a role-based DHT allowing a maximum of four topics and four processes.

We show in the following how to reduce the average size of the location view, i.e. a process uses in its location view only a subset $S_p \subset T_p$ such that $L_p = \{\mathit{hash}(p_{id}, t) \mid t \in S_p\}$. We achieve this by making a distinction between forwarders and children (cf. Figure 6.1). In this case case the average size of a location view is expected to be reduced proportional to the number of children accepted by a forwarder.

A process is called a *forwarder* with respect to a topic $t \in T_p$ if $\mathit{hash}(p_{id}, t) \in L_p$, otherwise it is said to be a *child*. For each value $v = \mathit{hash}(p_{id}, t) \in L_p$ a forwarder $p$ maintains a set of at most $k$ children $C_p(v)$ interested in $t$. For any child in $C_p(v)$, $p$ is also called the *parent*.

If $q$ is a child with $q \in C_p(\textit{hash}(p_{id}, t))$, then $p$ is the closest forwarder to $\textit{hash}(q_{id}, t)$. This means any routing request to a child will reach its parent, which then can forward the routing request. A child $q$ maintains a set of forwarders denoted by $F_q(v)$ which also includes its parent. $F_q(v)$ is typically initialised by its parent. Process $q$ can use $F_q(v)$ to route to any process interested.

### 6.4.2 An algorithm for a role-based DHT

In the following, we present an topic-aware algorithm which manages subscriptions and unsubscriptions and balances the cost in forwarding events, i.e. the algorithm maintains the forwarder/child relation between processes interested in the same topic. We show how processes, based on subscriptions and unsubscriptions, manage their location view and introduce the following operations needed when performing a subscription or unsubscription to a topic:

- *join*$(p, q, t)$: Assumes $q$ is the closest forwarder to $v_p = \textit{hash}(p_{id}, t)$. After performing this operation, $p$ will be a child of $q$, i.e. $p \in C_q(v_q := \textit{hash}(q_{id}, t)$ and $F_p(v_p)$ contains $q$ and neighbours of $q$ in $R_q(v_q)$.

- *leave*$(p, t)$: Assumes $p$ is a forwarder of $\textit{hash}(p_{id}, t)$ and there exists at least one neighbour interested in $t$. After performing this operation, $p$ is neither forwarder nor child with respect of $t$. The children of $p$ become children of the closest forwarder.

- *split*$(p, q, r, t)$: Assumes $q$ or $r$ is a forwarder to $\textit{hash}(p_{id}, t)$ with $q$ being the closest forwarder to $\textit{hash}(p_{id}, t)$ with $\textit{hash}(q_{id}, t) < \textit{hash}(p_{id}, t)$ and $r$ being the closest forwarder with $\textit{hash}(r_{id}, t) > \textit{hash}(p_{id}, t)$. After performing this operation $p$, $q$ and $r$ are forwarders, and the children of $q$ and $r$ become children at their closest forwarders.

- *split*$(p, q, t)$: Assumes $q$ is the closest forwarder to $\textit{hash}(p_{id}, t)$.
  If $\textit{hash}(q_{id}, t) < \textit{hash}(p_{id})$ then $q$ finds $r$ which is closest forwarder to $\textit{hash}(p_{id}, t) > \textit{hash}(p_{id}, t)$ and performs *split*$(p, q, r, t)$. Otherwise $q$ finds $r$ which is the closest forwarder to $\textit{hash}(p_{id}, t)$ with $\textit{hash}(p_{id}, t) < \textit{hash}(p_{id}, t)$ and performs *split*$(p, r, q, t)$.

- *replace*$(p, q, t)$: Assumes $q$ is a forwarder to $\textit{hash}(p_{id}, t)$ with its closest neighbours $x, y$. After performing this operation, $q$ is neither forwarder nor child with respect of $t$, while $p$ joins as a new forwarder. All children of $q, x, y$ become children at their respective closest forwarder.

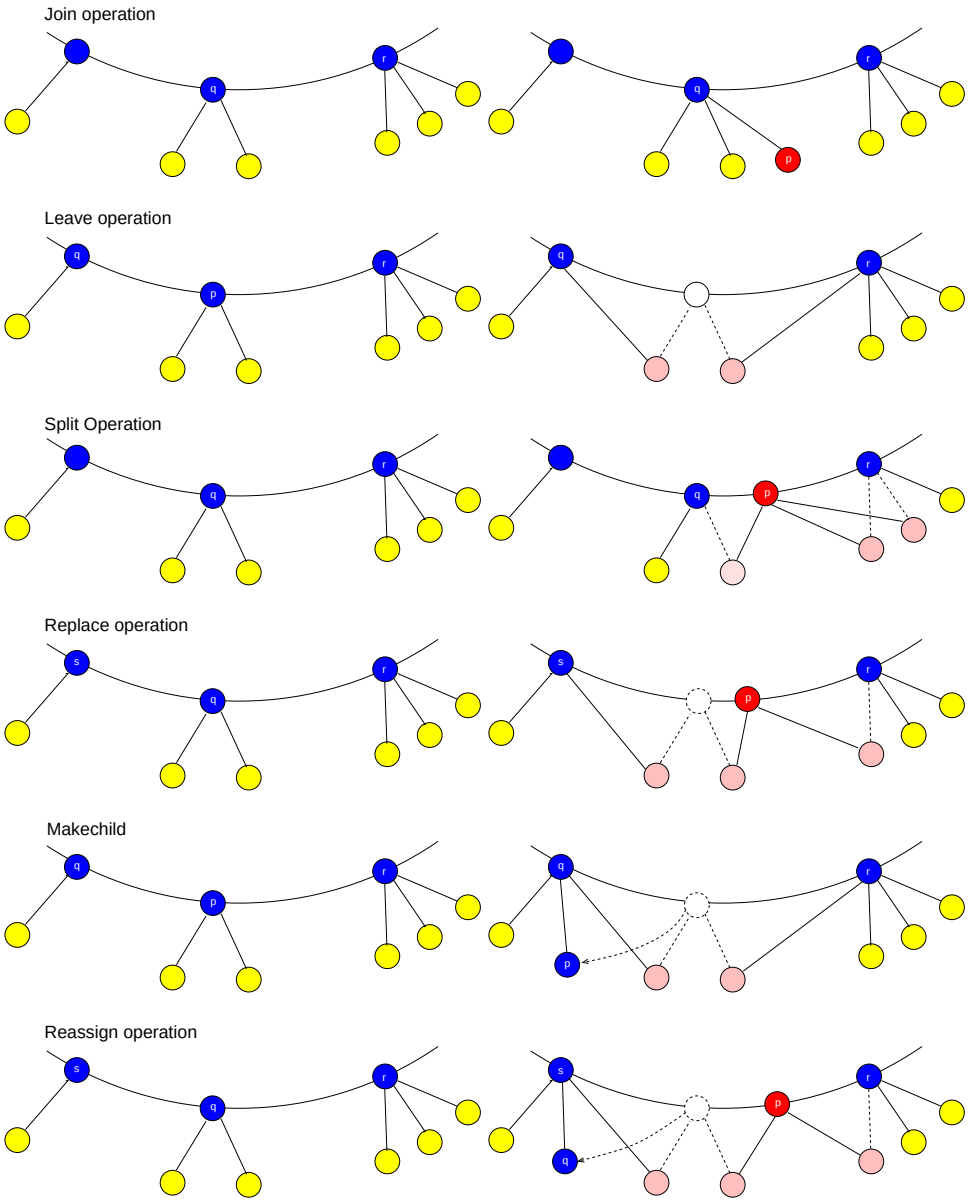



Figure 6.2: Algorithm's operations when performing subscriptions and unsubscriptions

- *makechild*$(p, q, t)$: Assumes $p$ and $q$ are forwarders for $t$ where $q$ is the closest forwarder to $p$. After performing this operation $p$ becomes a child of $q$ and all children of $p$ are reassigned to their respective closest forwarders.
- *reassign*$(p, q, t)$: Assumes $q$ is the closest forwarder to *hash*$(p_{id}, t)$. After performing this operation, $p$ will be a forwarder with respect to topic $t$ by performing first *split*$(p, q, t)$. Thereafter also $q$ becomes a child *makechild*$(q, p, t)$.

**Subscriptions.** Let $p$ be a process subscribing to topics $t_1, \ldots t_l$. Initially routes to $v_{p_i} :=$ *hash*$(p_{id}, t_i)$. Let $q_1, \ldots, q_l$ be processes where process $q_i$ maintains $v_{q_i} :=$ *hash*$(p_{id}, t_i)$. For any topic $t_i$ which is maintained by $q_i$, but $q_i$ is not interested in topic $t_i$, $p$ becomes a forwarder by adding *hash*$(p_{id}, t_i)$ to $L_p$. If $L_p$ is still empty, $p$ chooses $q_i$ with *hash*$(p_{id}, t_i)$ of $q_i$ with maximum set $C_{q_i}(v_{q_i})$ and initiates operation *split*$(p, q_i, t_i)$. After $p$ operates as forwarder for at least one topic it may initiate operation *replace*$(p, q_i, t)$ for forwarders $q_i$ with

$$\frac{|L_{q_i}| - 1}{|T_{q_i}|} > \frac{|L_p| + 1}{|T_p|}.$$

Besides considering the size of the location view one may consider other metrics such as the forwarded and received traffic of a process. Finally, $p$ initiates for all $i$ with $v_{p_i} \notin L_p$ operation *join*$(p, q_i, t_i)$.

**Unsubscriptions.** If process $p$ unsubscribes from topics $t_1, \ldots t_l$, it contacts all processes $q_i$ which function as a forwarder for $p$ to remove $p$ from its set of known children. For all topics $t_i$ for which $p$ functions as a forwarder and maintains any children, $p$ checks whether there exists another forwarder interested in the same topic. In this case $p$ initiates operation *leave*$(p, t_i)$. Otherwise $p$ selects a suitable process $q_i$ in $C_p($*hash*$(p, t_i))$ and initiates operation *replace*$(q_i, p, t_i)$. If $p$ is forwarder for topic $t_i$, but does not maintain any children for $t_i$, $p$ simply removes *hash*$(p, t_i)$ from its view.

**Maintaining at most $k$ children.** Subscription and unsubscription may overload certain processes, i.e. forwarder $p$ of $t$ may become responsible for more than $k$ children. In this case $p$ selects a suitable child, say $q$, and initiates operation *split*$(q, p, t)$.

### 6.4.3 Correctness in the absence of failures

A correct execution of a topic-ware subscription management scheme needs to ensure that i) a process will be forwarder or child for topic $t$ if and only if it is interested in topic $t$, ii) for every child $p$ there exists a parent and the parent is closest

to *hash*$(p_{id}, t)$ in spite of interleaving subscriptions and unsubscriptions. At first let us consider the case where operations do not interleave, i.e. every process is involved in at most one operation at a time.

**Lemma 6.4.2** *If all operations during unsubscription and subscription perform correctly and do not interleave, the subscription and unsubscription algorithm guarantees correctness.*

**Proof.** Since initially a process is interested in no topics, we need to show that after $p$ performs subscriptions or unsubscriptions, $T_p$ changes accordingly and for any process $q$, $T_q$ remains unaffected by the operations of $p$. Moreover, when inserting or removing a forwarder, all children need to be relocated to the closest parent.
When performing subscribe, $p$ becomes the forwarder for all topics where there is no other process. In this case there are no possible conflicts. If a subscribing process performs a *split*$(p, q, t)$ operation, children of $q$ become either children of $q$ or of $p$. Otherwise, $p$ joins as a child to the closest forwarder and no other processes are affected.
When unsubscribing, we distinguish between the case where $p$ unsubscribes as a child or as a forwarder. For topics for which $p$ is a child an unsubscription by contacting the parent does not affect any other processes. If $p$ is a forwarder and has no children $p$ can leave the DHT with respect to the topic safely without affecting any children. Otherwise, *replace* and *leave* ensure that all children of $p$ and of $p$'s closest neighbours are relocated to the closest forwarders.
Hence, if operations do not interfere, neither subscriptions and unsubscriptions affect any other processes and correctly maintain the forwarder/child relationship. □

In order to deal with interleaving operations we introduce the possibility to abort operations which then need to be retried by the initiator. Every operation is of the form *op*$(p, q, t)$ where $p$ initiates the operation whereas $q$ can abort the operation. $q$ serves an operation once all previous operations initiated or requested have been completed. However, there are two exceptions: first when $q$ detects that the assumption for an operation is not satisfied any longer then it will abort the operation immediately; second whenever it receives a join operation then it will either accepts or aborts it immediately.

**Lemma 6.4.3** *Every join operation succeeds after a finite number of retries.*

**Proof.** *join*$(p, q, t)$ will only be aborted after $q$ has left the DHT, i.e. by performing operations leave or replace. In this case the number of forwarders which can be found in a DHT is strictly decreasing. After rerouting a finite number of steps, $p$

will either be routed to a forwarder $q'$ accepting join or to a forwarder which is not interested in the topic. In this case $p$ performs *join*$(p, \perp, t)$ and will be a forwarder with respect to the topic. □

**Lemma 6.4.4** *After initiation, a leave operation will succeed in a finite number of steps.*

**Proof.** When performing leave, a process $p$ first leaves the DHT. Afterwards there will be no further operations which can interleave with $p$. $p$ requests from all children to perform a join operation. Note that these children are not involved in any other operation and thus perform join immediately. Since according to Lemma 6.4.3 join operations are guaranteed to finish in a finite number of steps, $p$ will receive from all children acknowledgements after a finite time. □

**Lemma 6.4.5** *Any sequence of interleaving operations maintains subscriptions and unsubscriptions correctly.*

Similar to the proofs of Lemma 6.4.4 and Lemma 6.4.3 we can show for the remaining illustrated operations that they can succeed in finite number of steps once initiated. Since each process serves only one operation at a time the execution of interleaving operations corresponds to sequential execution of non-interleaving operations.

### 6.4.4 Providing failure resilience

In the situation of link or process failures it becomes costly to establish correctness as illustrated in the case of no failures. Instead we assume that in the non faulty case the execution behaves equivalent as described in the previous section. However, when failures occur we allow communication to timeout. In this case a process can still locally complete the operation. As a consequence, some children may not be connected to a forwarder, or they are connected to a forwarder which is not closest to them. Therfore, every child performs a little protocol periodically which self-stabilises to a correct state.

Algorithm 5 illustrates how children become reassigned to their closest parent. Every child $p$ routes to *hash*$(p_{id}, t)$ and its parent by contacting a node in $F_p($*hash*$(p_{id}, t))$ which forwards the routing request. Let $q$ be the process closest to *hash*$(p_{id}, t)$ receiving the routing request of $p$. If $p$ is not a child of $q$, $q$ adds $p$ to its set of children. In any case, $q$ acknowledges to be the parent and forwards

Algorithm 5: Self-Stabilising Protocol finding for a child the closest parent

**VAR**

| | |
|---|---|
| $C_p$: | the parent known $p$ |
| $F_p$: | Alternative forwarders known to $p$ |
| $R_p$: | Routing table of $p$ |
| $Q_p$: | Neighbour set of the routing table of $p$ |
| *parent*$_p$: | Parent of $p$ |
| *recupdate*: | Truth value to check whether $p$ received an update |
| $l$: | maximum number of alternative forwarders |

**Repeat periodically**
  **if** *recupdate* $= true$ **then**
    Choose $q \in F_p(\mathit{hash}(p_{id}, t))$
    Send to $q$ $\langle \mathit{ROUTEPARENT}, p_{id}, t \rangle$
  **else**
    Using $R_p(\mathit{hash}(p_{id}), t')$ find $q = \mathit{lookup}(\mathit{hash}(p_{id}, t))$
    Send to $q$ $\langle \mathit{ISPARENT}, p_{id}, t \rangle$
  **end if**

**On** $q$ receives $\langle \mathit{ROUTEPARENT}, p_{id}, t \rangle$
  Using $R_q(\mathit{hash}(q_{id}), t)$ find $r = \mathit{lookup}(\mathit{hash}(p_{id}, t))$
  Send to $r$ $\langle \mathit{ISPARENT}, p_{id}, t \rangle$

**On** $q$ receives $\langle \mathit{ISPARENT}, p_{id}, t \rangle$
  $C_q(\mathit{hash}(q_{id}, t)) = C_q(\mathit{hash}(q_{id}, t)) \cup p_{id}$
  **for** $i = 1$ to $l$ **do**
    Choose $r_{id} \in Q_q(\mathit{hash}(q_{id}, t))$ uniformly at random
    $F'_p(\mathit{hash}(q_{id}, t)) = F'_p(\mathit{hash}(q_{id}, t)) \cup r_{id}$
  **end for**
  Send to $p$ $\langle \mathit{ISCHILD}, q_{id}, t, F'_p(\mathit{hash}(q_{id}, t)) \rangle$

**On** $p$ receives $\langle \mathit{ISCHILD}, q_{id}, t, F'_p(\mathit{hash}(q_{id}, t)) \rangle$
  recupdate = true
  *parent*$_p(\mathit{hash}(p_{id}, t)) = q_{id}$
  $F_p(\mathit{hash}(p_{id}, t)) = F'_p(\mathit{hash}(q_{id}, t)) \cup q_{id}$

a recent view of its neighbours to $p$. If $p$ does not receive an acknowledgement, it can retry using an arbitrary role for routing.

**Lemma 6.4.6** *If no failures occur after a finite number of steps any child will be located to the closest forwarder.*

### 6.4.5 Topic hierarchies

In order to establish topic hierarchies, we assume that a process which subscribes to a topic $t_i \in T$ knows about all topics $t_j$ with $(t_i, t_j) \in H(T)$. Using the property of role-based DHT which provides processes with the possibility to perform a random lookup, a process can find a process chosen uniformly at random which is interested in a lower level topic. Lemma 6.4.7 shows that the algorithm introduced Section 6.4.1 provides this property.

**Lemma 6.4.7** *Let $G_t = \{p_1, \ldots, p_l\}$ denote a non-empty set of processes that subscribed to topic t. The role-based DHT algorithm of Section 6.4.1 allows one to route to a process in $G_t$ chosen uniformly at random.*

**Proof.** A process selects a random value *rand* and requests to route to *hash*$(rand, t)$. Since $G_t$ is non empty, the topic-aware algorithm guarantees that there exists at least one process being forwarder of $t$. The routing of the DHT guarantees to find the forwarder closest to *hash*$(rand, t)$. Since each forwarder maintains in its view the closest processes in $G_t$, a route will locate the process in $G_t$ closest to *hash*$(rand, t)$. By the uniformness of *hash* this corresponds to selecting uniformly at random a process in $G_t$. □

A forwarder of topic $t_i$ will forward all corresponding events to a process interested in $t_j$ with $(t_i, t_j) \in E(T)$ by performing a random lookup as illustrated in Lemma 6.4.7.

## 6.5 Dissemination using topic-awareness

Section 6.4 has shown how to achieve topic-awareness in a DHT. This section focuses on dissemination schemes which remove parasite messages and aim to provide fair load sharing. We focus on two ways of achieving dissemination, gossiping and application-level multicast. Gossiping is commonly used by unstructured peer-to-peer systems, however we show that one can maintain at low cost a suitable membership to implement gossiping by exploiting the uniform structure of DHTs. Application-level multicast is the common way to achieve dissemination and can easily be constructed using the routing table of the DHT. Differences between

gossip-based and application-level multicast dissemination are based on reliability and message load given by the schemes. Gossip-based dissemination aims at reliability expressed with high probability in the occurrence of failures. Application-level multicast in structured peer-to-peer systems, ensures self-maintenance of the multicast tree, i.e. a failure of a process will cause restructuring of the application-level multicast. Reliability achieved by gossip-based dissemination use high redundancy. Therefore, the message complexity, i.e. $O(n \log n)$ push-based unstructured protocols compared to $\Theta(n \log \log n)$ using combined push and pull is higher than by using application-level multicast, which can be achieved by sending $\Theta(n)$ messages.

### 6.5.1 Gossip-based dissemination

Gossip-based protocols maintain a view of communication partners which may change over time. We consider a framework of gossiping in the spirit of pbcast. In every round a process communicates with $k$ partners ($k$ denotes the fanout) from its view, chosen uniformly at random, and informs them about recently informed events. Reliability depends on the way how the dissemination scheme terminates (cf. [Koldehofe 2003]), the fanout, and the size of the local view. Ideally the membership scheme allows processes to choose communication partners uniformly at random among all group members. Lpbcast addresses this issue by maintaining partial views and exchanging randomly selected communication partners, such that a random choice of a process from the local view, appears as choosing the destination uniformly at random among all processes. In Scamp, randomness is used to initialise the view of a process. The view size needs to be in the $O(\log n + C)$, and processes have to gossip in every round with approximately $O(\log n)$ communication partners in order to achieve reliability with high probability.

In the following, we show that DHTs can be used to efficiently maintain partial views before proposing how to combine gossiping in a fair manner with structured DHT of section 6.4. Algorithm 6 outlines a generic lightweight membership algorithm. A process maintains a view containing a maximum of $l$ other group members. Depending on the fanout and the frequency the membership algorithm refreshes the view of a process. Refreshing the view happens by each process sending with probability $f$ its own address to $K$ processes of the system. The destinations of the processes are determined by routing to $K$ values chosen uniformly at random within the domain of the DHT's hash function.

Algorithm 6: Generic Lightweight Membership Protocol

**VAR**
$V_p$: known group members of $p$
$K$: fanout
$f$: frequency the view gets updated
$l$: maximum size of a view

**Initialisation**
**for** $i = 1$ to $l$ **do**
$r =$ random value $\in hash(\Sigma^*)$
route $\langle$*REQUEST*, $p_{id}\rangle$ to $r$.
**end for**

**Do** in every round with probability $f$
**for** $i = 1$ to $K$ **do**
$r =$ random value $\in hash(\Sigma^*)$
route $\langle$*REQUEST*, $p_{id}\rangle$ to $r$.
**end for**

**On** $p$ receives $\langle$*MYADDRESS*, $q_{id}\rangle$
$V_p = V_p \cup q_{id}$
**if** $|V_p| > l$ **then**
remove oldest identifier s.t. $|V_p| = l$
**end if**

**On** $p$ receives $\langle$*REQUEST*, $q_{id}\rangle$
Send to $q$ $\langle$*MYADDRESS*, $p_{id}\rangle$

**Lemma 6.5.1** *Assume hash provides a uniform mapping of process identifiers to their locations in a DHT. An algorithm using the membership of Algorithm 1 for $l = K$ and $f = 1$ corresponds to selecting neighbours uniformly at random from a full view. The membership algorithm provides an overhead of $O(K \log n)$ additional routing messages performed by each process.*

**Remark 6.5.1** *pbcast implemented in an unstructured peer-to-peer network requires to maintain a full view. Lemma 6.5.1 shows that in a structured peer-to-peer network one can reduce the view size to $k$ by using an overhead of $O(k \log n)$ additional routing messages performed by each process. Algorithm 6 can also initialise the membership provided used in the dissemination scheme of Scamp. However, here we do not experience any overhead since the local view remains static.*

**Remark 6.5.2** *Algorithm 6 can also implement lightweight gossiping by exploiting proximity of the membership information.*

**Topic-aware gossiping.** In order to combine the role-based DHT providing topic-awareness of Section 6.4 with Algorithm 6, every process $p$ needs to maintain for every topic $t \in T_p$ corresponding data structures e.g. gossip view, update frequency, etc. According to Lemma 6.4.7, routing to $r = \mathit{hash}(\mathit{rand}, t)$ finds group members chosen uniformly at random among the processes which subscribed to topic $t$. Hence, we can generalise Algorithm 6 to maintain for each topic a gossip view. Dissemination of events with respect to topic $t$ happens by using the corresponding view to $t$.

### 6.5.2 Application-level multicast forest

Using a role-based DHT allows a space efficient construction of a forest of application-level multicast trees which can be used to achieve a fair and efficient distribution of the work without using any parasite messages. Achieving a fair data distribution is coupled to all processes, interested in the same topic, performing equal amount of work when forwarding a message. We ensure this by constructing a forest in which every forwarder/process is a root of an application-level multicast tree. By using the random routing technique, introduced in Section 6.4.5, the load of performing a multicast becomes evenly balanced, and according to Lemma 6.4.7 only processes interested in the topic become involved. In the following, we outline the construction and maintenance of the forest. In our description for simplicity we consider only forwarders. However, note that it is straight-forward to generalise this scheme to include all processes interested in the topic.

Let us consider a topic $t \in T$. Each forwarder of $t$ is required to maintain $O(\log N_t)$ links. Similar to [Bickson et al. 2004] we distinguish between level $0, \ldots, \lceil \log N_t \rceil - 1$ links. During the dissemination of an event the root of the tree will use level $0$ links to forward the event. Accordingly, a process will forward an event, once it has received the event via a level $i$ link, to its level $i+1$ links.

Let $p$ be a forwarder and let $m = |\mathit{hash}(\Sigma^*)|$. We propose a construction of an application-level multicast where a level $i$ link corresponds to the forwarders reachable when routing to

$$\mathit{hash}(t) \circ (\mathit{hash}(p_{id}) + m/2^i \bmod m)$$

and

$$\mathit{hash}(t) \circ (\mathit{hash}(p_{id}) - m/2^i \bmod m).$$

In this construction the highest level links are the closest neighbours in $Q_p$. Hence, a process can determine the number of levels needed locally by using its neighbourhood set. The maintenance of level links can happen in two ways: i) bounded to incoming links, i.e. from time to time a process reinitialises its outgoing level $i$

links, or ii) bounded to outgoing links, i.e. processes route to their level $i$ incoming links (using the same function).

Maintenance bounded to outgoing links is of advantage for quickly adapting to changes in the structure. For example, in a construction which is bounded to incoming links a new process subscribing may need to wait some time before receiving disseminated events, while a construction which is bounded to outgoing links ensures that processes receive events immediately after a successful update of all incoming links.

In order to deal with failing processes the root of a multicast tree needs to send heartbeat messages periodically. Processes which did not receive any heartbeat messages via a level $i$ link perform a lookup of the respective incoming link.

Further redundancy can be achieved by disseminating via multiple application-level multicast tress. This ensures that in spite of a fixed number of failures a disseminated event reaches all interested and failure-free processes. Redundancy can also be used to save on heartbeat messages. When disseminating an event the identifiers of all root processes of used application-level multicast trees are attached to the event. A process which received the event only from a subset of these processes can determine which multicast tree did not perform correctly and fix the respective level $i$ link.

### 6.5.3 Dissemination using topic hierarchies

Both dissemination schemes of Section 6.5.1 and Section 6.5.2 can be used to consider topic hierarchies. The gossiping schemes of Section 6.5.1 can be combined with data-aware multicast [Baehni et al. 2004]. By modification of Algorithm 6 we can maintain a set of random vertices of the closest higher-level topic for which there is at least one interested process. When gossiping, a part of a process's gossip message will go to higher-level processes.

When using application-level multicast the root of the tree is also responsible to forward the published event to a process of the next level. Again one can modify Algorithm 6 for processes to maintain a set of higher level processes which can be used to initiate a redundant application-level multicast of the next higher level.

## 6.6 Evaluation and analysis

We evaluate the dissemination schemes of Section 6.5 with respect to fair sharing, space complexity, time complexity, and reliability. We distinguish between four protocols:

- *FullGossip*: The dissemination scheme uses gossiping and makes no distinction between forwarders and children.
- *ForwardGossip*: Only forwarders are used in a gossip view. A forwarder receiving an event for the first time forwards the event its set of children.
- *FullTreeCast*: Every process is part of an application-level multicast for all topics subscribed.
- *ForwardTreeCast*: Only forwarders are used to built a forest of application-level multicast trees. A forwarder also propagates an event to its children when receiving the event for the first time.

**Fair sharing.** In a gossip-based protocol a process communicates with destinations chosen uniformly at random from its view. In a dynamic view initialised by Algorithm 6 every process/forwarder has the same probability of receiving an event and being involved in gossiping about an event in the next round. Hence, every process/forwarder is expected to perform the same amount of work with respect to a topic. For the *FullGossip* protocol this implies that every process performs fairly. In the *ForwardGossip* protocol fairness depends on the distribution of forwarders and children. A process being forwarder of a hot topic has potentially to do more work than a forwarder of a low-interest topic. If the distribution of traffic with respect to topics is known in advance we can use this information during subscription to achieve fairness (cf. Section 6.4.1). Otherwise the use of heuristics can give a fair distribution of the load among the processes.

The construction of application-level multicast trees in Section 6.5.2 ensures that every process interested in a topic $t$ is root of exactly one application-level multicast tree. Choosing the root for disseminating uniformly at random guarantees that every process is expected to perform $2^i/N_t$ messages via a level $i$ link. e.g. in half of the cases a process is expected to propagate messages while in the other half a process is a leave node in an application-level multicast tree. Similar to gossip-based protocols *FullGossip* provides full fairness, while *ForwardGossip* requires also a fair assignment of forwarders and children.

**Space complexity.** In combination with the role-based DHT the main space cost is the maintenance of routing tables. For each value $v = \mathit{hash}(p, t) \in L_p$ a process keeps a routing table of size of $O(\log N)$. Note that the size of a view or the number of links is bounded by $O(\log N_t)$. Hence, the full view dissemination schemes needs $O(|T_p| \log N)$ while forwarder-based dissemination schemes require space $O(|L_p| \log N)$. Assuming a fair distribution of children and forwarders, the average space by a process depends on the overall number of topics maintained in

a publish/subscribe system and how densely each topic is populated. Each active topic needs at least one forwarder. Let $k$ be the maximum number of children a forwarder accepts before performing a split operation. If for every $t \in T$ there exists at least $k$ processes sharing an interest then $|L_p|$ can be bounded by $1 + |T_p|/(k+1)$, i.e. achieving a space reduction assumes the topic space is densely populated.

A more significant space reduction can be achieved for densely populated topics if one associates with children a tree-like structure. Each vertex of the tree corresponds to a DHT using $O(\log N_t)$ vertices. Hence, the depth of the tree is in the order of $O(\log N_t / \log\log N_t)$ and each child's routing table is bounded to $O(\log\log N_t)$. For determining the correct position we take a hash function which maps $p_{id}$ similar to CAN [Ratnasamy et al. 2001] to a Cartesian product of $O(\log N_t / \log\log N_t)$ dimensions and additional a level which equals $i$ with probability $(\log N_t)^i / N_t$. In this case the value of each dimension determines the path to follow while the level determines at which level of tree to join a DHT. Within a DHT a process always routes to the process closest to $\mathit{hash}(p_{id})$. By using this scheme in a densely-populated topic space, a process needs only to be forwarder of $O(\log |T_p|)$ topics and maintain a routing table of size $O(\log N)$ whereas for the remaining topics a process maintains a routing table of size $O(\log\log N)$, resulting in an overall space complexity of $O(\log |T_p| \log N + |T_p| \log\log N)$. Although the scheme is more space efficient when topics are densely populated and one could use a similar protocol like Algorithm 5 to self-stabilise children to be closest to their forwarder, the maintenance cost and overhead are significantly higher.

**Time complexity.** For a topic $t$ gossip-based dissemination and application-level multicast terminate in $O(\log N_t)$ rounds. When using topic hierarchies, in each round a topic will reduce the distance to processes which have subscribed to higher level topics. In this case the worst-case time complexity is $O(\log N_t \log |T|)$. Distinguishing between forwarders and children adds only a constant cost.

**Reliability.** Application-level multicasts protocol can tolerate up to $f$ failures when using $f + 1$ different root vertices. When distinguishing between forwarders and children then a failure of a forwarder affects at most $k$ children. Hence, $f$ failures will affect $kf$ processes not receiving the message.

Gossip-based dissemination informs w.h.p., i.e. for a constant $c$ with probability $O(1 - n^{-c})$ all processes assuming for a constant $1 > \epsilon > 0$, there exists at least $(1 - \epsilon)$ non-faulty processes. In the case when distinguishing between forwarders and children every failing process will affect $k$ children.

## 6.7 Conclusion

This work has proposed a role-based subscription management algorithm suitable for large-scale topic-based publish/subscribe systems. In combination with fundamental dissemination schemes, topic-awareness ensures a fair and efficient distribution of events by maintaining also the benefits of recent structured peer-to-peer membership schemes in providing a failure resilient maintenance of membership. Topic-awareness allows one to deploy the technique of random lookups which find for a given topic a process chosen uniformly at random which has subscribed to the topic. This technique gives benefits when establishing topic hierarchies, but also supports failure redundant event dissemination.

The results of this work suggest that the distinction between structured and unstructured peer-to-peer dissemination systems is not a matter of which dissemination system is used, but rather the structure of the peer-to-peer system matters in how to bootstrap data structures used for disseminating events.

In the context of gossiping, this work has shown that one can use a structured peer-to-peer system in order to establish in a space efficient way a dynamic set of randomly chosen communication partners. The cost is $O(\log N)$ routing messages per round compared to $O(1)$ when maintaining a static membership. This corresponds to the higher space complexity of dynamic views over static views in unstructured networks.

# Part II

# Collaborative Learning of Distributed Algorithms

# Seven

# Collaborative Learning Using Simulation and Visualisation

This part considers the use of interaction and collaboration in order to support students in learning distributed algorithms. The goal is to provide students with tools that help in following the execution of an algorithm, support the discovery of critical instances of the algorithm, as well as to allow students to interact and cooperate with each other on the same problem.

*Distributed algorithms* run concurrently on many interconnected processing elements called processors or processes. Such algorithms need to work correctly independent from the speed of the communication links and the structure of the network. Understanding distributed algorithms, including their performance analysis and their correctness, is important in courses related to distributed systems, operating systems and computer networks. While for many algorithms the execution follows a sequential thread of control, for distributed algorithms one needs to consider complex interaction among processes. Tracing the execution of distributed algorithms gives a huge amount of data describing local state and interaction among processes. Moreover, the variation in speed of communication links may end in a different flow of control, even for the same input.

In this thesis, simulation and visualisation are examined as a method in providing interactivity and collaboration to users. Visualisation can help to obtain an overview of the global state and important events in a distributed system's execution. Combining visualisation and simulation can also help to identify critical properties of algorithms. Such critical properties may occur only sporadically in a real system's execution and may be difficult to find using real traces. However, identifying critical properties is important in order to achieve an understanding of the correctness of algorithms.

**Structure of this chapter.** Besides providing background information on collaborative and interactive learning in Section 7.1, this chapter identifies several levels of interactivity which can be supported when providing general computer-based simulation and visualisation components (cf. Section 7.2). The different levels of interactivity are related to educational scenarios as well as algorithmic solutions discussed in Part I. Section 7.3 discusses the design of simulation components to allow real-time interactions while the execution of a distributed algorithm evolves. Moreover, collaborative situations are considered in which users control the behaviour of an algorithm by introducing a decentralised simulation framework. Section 7.4 gives an overview of the contributions of the papers in the subsequent chapters.

## 7.1 Background: collaborative and interactive learning

Collaborative and interactive learning has been suggested to be used in several educational levels as a way of improving the understanding of a concept. While in collaborative learning the focus is on supporting social interactions between the participants in a learning situation, interactive learning is about interactions with *learning objects*. A learning object as introduced in web-based learning is a reusable instructional component (cf. [Wiley 2000]), for instance a simulation/visualisation environment.

Educational research has conducted work examining the effectiveness of collaborative learning (see [Dillenbourg et al. 1996] for an overview). Results presented in [Slavin 1995] suggest that in many scenarios collaboration among students can be more effective than students working alone. However, collaboration by itself is not a guarantee for more effective learning as often assumed. The effectiveness often depends on the interactions which are supported between collaborators. In practice the outcome may depend on many more factors, for instance the engagement of students and the introduction of the collaborative situation by the instructor.

Several learning theories such as the socio-constructivist approach, socio-cultural approach and shared recognition approach (cf. [Dillenbourg et al. 1996]) have been proposed in order to explain how collaboration can support learning and what conditions should be provided to make collaborative learning efficient. Typically, learning theories deal with the effect of social interactions in collaborative situations, but they focus on face-to-face learning in specific circumstances, for instance considering age differences among collaborators.

*Computer-supported collaborative learning (CSCL)* addresses the use of computers to ease collaboration between students who may be physically not present at

the same location. CSCL has been suggested to support concept learning, problem solving, and designing [Kumar 1996]. As pointed out in [Williams and Roberts 2002] CSCL can be helpful to support collaborative situations in large classes and may serve as a way to reduce discrimination of and competition between students. Since communication in CSCL typically uses the Internet, CSCL can be used in distance education. In distance education courses, CSCL can help to enrich the web based lecture material and provide services to the learners so they can exchange and discuss problems with the teacher and other course participants. CSCL in distance education may also be used to prepare students to work in international teams, as shown in a collaborative programming project by [Last et al. 2000].

Despite the previously mentioned positive effects, the possibility of social interactions is limited by the use of technology and therefore a critical point when introducing CSCL related technology into the classroom [Kreijns and Kirschner 2001]. The problems in supporting natural interactions and presence when using modern technology are also discussed in the context of virtual reality applications [Heldal 2004]. In fact, the combination of virtual reality with CSCL has lead to a variation called *computer-supported collaborative learning requiring immediate presence (CSCLIP)*. An overview of CSCLIP related systems is presented in [Lucca et al. 2003].

So far collaboration has been discussed in the context of inter-human interactions. In computer-supported interactive learning environments, as is the case for a simulation/visualisation environment, an important factor deals with interaction between the learning object and human beings. Depending on the level of interactivity one may say the learner collaborates with the learning object if the interactions help the learner to develop a new insight.

Different levels of interactivity with learning objects has been discussed in the literature. For example, a graphical user interface provides a user with a basic way of interaction such as to start, to run, or to stop a learning task. Beyond this it is desirable for a learning environment to provide interactions which allow users to influence the state of a learning object. In order to identify fundamental differences, an effort in categorising interactivity in computer-based environments has been made. [Schulmeister 2003] distinguishes among the following levels of interactivity:

1. *Viewing objects and receiving*
2. *Watching and receiving multiple representations*
3. *Varying the form of representations*
4. *Manipulating the content*

5. *Constructing the object or representation contents*

6. *Receiving intelligent feedback from the system through manipulative action*

Many algorithm visualisations are restricted to observing one or multiple representations of the same algorithm execution. Often the only form of interactions provided are changing the view of the visualisation, and the algorithm's input data. This limited form of interactivity may be an explanation for mixed results in examining the effectiveness of algorithm animations in computer science education [Hundhausen et al. 2002].

However, in many cases algorithms consider more aspects which can be used to provide better interactivity. Especially distributed algorithms cannot be explained by following a sequential thread of control. The behaviour of the execution depends on the underlying network and the parameters defining timing of links between processes. A simulation/visualisation environment, as discussed in Chapter 8, allows users to

- change the input of the algorithm,
- model own networks,
- change the properties of links in the system,
- modify the algorithm,
- define a failure model.

The simulated and visualised execution gives feedback of the behaviour of the algorithm under the user-defined properties. This can help to identify and understand properties of the formal algorithm description.

Nevertheless, simulation/visualisation environments also differ in the level of interactivity they can support. In the following we identify levels of interactivity in a simulation/visualisation environment and discuss the possible solutions of implementations based on the algorithmic ideas discussed in Part I.

## 7.2 Levels of interactivity in a simulation/visualisation environment

A design criteria for simulation/visualisation environments is to decouple simulation and visualisation in autonomous system components. Typical benefits to a learning environment are exchangeability and flexible use of components. One can easily replace the simulation by using real system traces or use a visualisation

to monitor the execution of a real system. Moreover, one can support the monitoring of the execution from multiple places at the same time, e.g. in the setting of a labroom where students can follow the execution of a discussed algorithm, however individually controlling the speed of the animation. The same is true for replacing/adding new visualisation components. One may provide alternative visualisations illustrating a concept from a different perspective or simply provide an improved representation. Especially, differences in the perceptions among students as well as teachers make it desirable to have multiple representations available.

However, having components separated gives challenges in order to provide interactivity during the execution of a simulation. For example, considering failures in a distributed system execution leads to questions as to how the system would behave if a specific link or process fails. Although in a simulation the simulator may be configured to let processes fail, a more intuitive way for a learner is to take the decision online in order to construct certain algorithm scenarios. We distinguish among the following levels of interactions in a simulation:

- *offline interactions.* All decisions influencing the execution of a simulation happen before starting the simulation. Once started the simulator has control on how the execution evolves. The user can only interact with the visualisation by determining the speed of the visualisation or deciding the view on the execution, but the user's interactions do not change the behaviour of the algorithm's execution.

- *real-time interactions.* Single or multiple users can influence the execution as the algorithm evolves. From an educational perspective it makes a difference whether multiple users interact at the same time or only a single user. Therefore, we distinguish between *single-user real-time interactions* and *multi-user real time interactions*. Each interaction of a user may change the behaviour of the algorithm's system execution. For example, a user can decide that a link or a process will fail. For multiple users the interactions may also be a meaningful way to collaborate among each other. However, the simulator is in charge of validating interactions and evaluating the outcome. If no interactions occur the simulator shows the behaviour as in the offline scenario.

- *collaborative interactions.* While offline and real-time interactions assume a central simulation component, collaborative interactions reflect a distributed form of simulation where the system is split in several components, each maintained by single simulation component. Collaborative interactions allow one user to decide the steps of a simulation component. The multiple simulation components need to collaborate in order for the system to per-

form correctly. Collaborative interactions can be combined with multi-user interactions. Each user may subscribe to a set of components and choose the level of detail of the whole simulation to be observed.

Single and multi-user interactions are helpful for interfering with the execution of a distributed algorithm as it evolves and also offer the possibility of a basic form of collaboration. Collaborative interactions are motivated by the results in [Ben-Ari 2001] presenting an interactive animation. In the animation a user needs to maintain the decision for a process of the distributed system while the simulator is responsible for maintaining the state of the remaining processes. This technique can also be generalised to support multi-user collaboration in a group of users by each user taking control over a process. By supporting interactions between users we can achieve collaboration where users share the goal of maintaining a well-functioning distributed system.

## 7.3 Design of educational simulation components on supporting interactivity

A simulator for distributed algorithms allows one to run the execution of an algorithm in a user-defined environment and produces according to the given configuration parameters an execution which appears to have been executed on a real system. Examples of typical configuration parameters are, among others, the topology of the network and the speed and design of communication channels. A basic simulation environment facilitates:

- the design of one's own algorithm by providing a high level programming language,
- the design of network topologies,
- the execution under different failure models,
- creation of traces of the system execution.

There are two common approaches to the design of a simulation environments: *Transport-centred simulation* supports simulation of protocols by providing a realistic model of the protocols transport (cf. [Bajaj et al. 1999]). The creation of algorithms is often close to network programming. The simulation allows one to draw conclusions from the execution of an algorithm such that a user can receive realistic feedback on the performance. *Event-driven simulation* as introduced in Chapter 8 does not assume a specific transport protocol. In the context of message

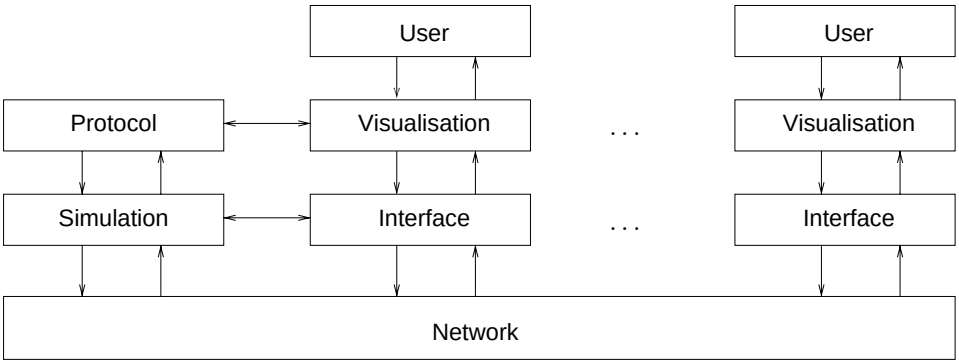



Figure 7.1: Multi user real-time interaction in an simulation/visualisation environment.

passing one distinguishes among send, receive and internal events. The occurrence of corresponding send and receive events depends on the topology description. This way protocols can often be implemented according to a typical textbook description.

In an educational scenario a transport-centred simulation is of interest if the purpose is to familiarise learners with the characteristic of particular transport protocol or with issues involved in network programming. If the focus is on understanding fundamental algorithmic ideas and learning to analyse algorithms based on time and space complexity an event-driven simulation may be the more suitable choice.

Most commonly the interactions supported by a simulation are offline interactions. A user defines properties of the topology before executing a protocol. In many cases it is desirable to receive feedback and change the behaviour of protocols, e.g. by failing links or processes as the execution evolves. This requires that a simulator supports feedback.

A *simulator supporting feedback* is a simulator with an input and an output channel. The simulation performs in rounds where each round consists of two phases. In the first phase, at the beginning of each round, the simulator reads interactions from its input channel. Depending on the underlying interaction model, the simulator first validates the interactions, and for all allowed interactions the simulation is updated. In the second phase, at the end of the round, the simulation continues with the execution of the algorithm by creating a sequence of events corresponding to the changes of successfully executed interactions and the next steps of the algorithm execution. The sequence of events is written to the output chanel of the simulator. In an event-based simulation an interaction typically is valid if it does not collide with any concurrent interactions and if the interaction is based on

the events created by the simulator in the previous round.

Simulation supporting feedback allows the implementation of single-user as well as multi-user real-time interactions as proposed in Section 7.2. The simulation/visualisation consists of one simulation process and an arbitrary number of user-steered *interfaces* (cf. Figure 7.1). An interface serves as a communication layer for a visualisation component. In terms of event-based dissemination interfaces and simulator form a group. The simulator multicasts all events written to an output chanel. Each event is associated with a sequence number and a round number. When performing an interaction the interface forwards the interaction using point-to-point communication to the simulator. Each interaction message also carries information on the round in which events have been observed. This allows the simulator to verify the validity of an interaction.

Having a single simulation component facilitates the implementation of interactions since all event ordering is coordinated by the simulator. The simulator determines the speed at which events are created. In order to perform valid interactions the interfaces must follow the speed of the simulator.

The design of collaborative interactions requires a different notion of simulation. So far only one process controlled the simulation while other processes performed real-time interactions via interfaces. In order to support collaborative interactions we propose to use a *distributed simulation* where the system is split into components. A *component* consists of a set of processes and each component is controlled by a different simulation process. Not necessarily all components use the same simulation. For instance, a simulation may be computer-controlled or user-controlled. Components are said to be adjacent if in the underlying network topology there is a link connecting a process of each component. If, in a simulation, a process sends a message to an adjacent component than the component of the sender is responsible for creating the send event while the component of the receiver is responsible for creating the corresponding receive event.

An interface may subscribe to several components and perform real-time interactions. The simulation process of each component is in charge of validating the interactions. If each component consists of only a single process, than the simulation corresponds to a real system execution supporting user interactions.

Collaborative interactions are suitable for a simulation of protocols using an asynchronous communication paradigm. While messages inside a component depend on the simulation parameters, a message sent between two adjacent components depends also on the communication delay between the corresponding simulation processes. However, for interfaces subscribing to multiple components it is important to respect the causal relations among components.

The simulation of protocols using a synchronous communication paradigm requires a synchronisation between all components to proceed at the same speed.

This restricts the overall number of components which can be supported.

The implementation of collaborative interactions using a large number of components and asynchronous communication can be implemented using the publish/subscribe paradigm. In a large system with continuous joining and leaving of peers, a component may consist of multiple replicated simulation processes where only the most significant simulation decides the next sequence of events. We can use, for instance, lightweight causal cluster management of Chapter 5 and topic awareness of Chapter 6 to determine the simulation process responsible for performing the next simulation step and obtain a causal order relation for interfaces which have subscribed to multiple components. Each simulation process, before publishing a sequence of events, performs a lookup of the hashed component name. The lookup returns the closest process to the hashed component name in the topic aware DHT. This process may proceed and publish its events. Note, however, that in the occurrence of failures there may temporarily be multiple processes which publish simulation events, and the state of interfaces may temporarily be different.

## 7.4 Contributions

In Chapter 8 and Chapter 9 we propose and examine two forms of visualisation/ simulation environments:

- LYDIAN, a computer-based visualisation/simulation environment,
- dramatisation using human actors.

Besides presenting the ideas behind the approaches, an evaluation of the two forms has been undertaken in two course studies.

LYDIAN is a simulation/visualisation environment, which has been developed as part of this thesis. It supports the functionality described for simulation/ visualisation using offline interactions. LYDIAN allows one to:

- simulate existing protocols,
- modify existing or create new protocols,
- create one's own network structures,
- view an animation following the execution of a simulated protocol,
- create new animations.

In particular we have examined general meaningful visual representations for concepts in distributed systems.

While LYDIAN is a general-purpose environment for distributed systems, we have also explored dramatisation for a specific distributed concept. By using human actors we could achieve multi-user real-time interactions while a simulation of a self-stabilising algorithm was proceeding. The idea of dramatisation has also been transfered to a computer-based environment using some of the techniques discussed in this chapter.

# Eight

# LYDIAN

Boris Koldehofe
Marina Papatriantafilou
and
Philippas Tsigas

**Abstract**

LYDIAN is an environment to support the teaching and learning of distributed algorithms. It provides a collection of distributed algorithms as well as continuous animations. Users can combine algorithms and animations with arbitrary network structures defining the interconnection and behaviour of the distributed algorithm. Further, it facilitates the creation of own algorithm descriptions as well as the creation of own network structures. This makes LYDIAN a flexible tool to be used with students of different skills and backgrounds.

This article gives an overview about various ideas and concepts behind LYDIAN (which have been discussed before separately in several publications [Papatriantafilou and Tsigas 1998; Koldehofe et al. 1998; Koldehofe et al. 1999; Koldehofe et al. 2000; Holdfeldt et al. 2002; Koldehofe et al. 2003]) by describing in detail the framework for an educational visualisation and simulation environment for learning/teaching distributed algorithms as well as discussing possible extensions which may improve possibilities for user interaction. Moreover, in our effort to understand better what visualisation and simulation environments such as LYDIAN need to provide we show results taken from a case study integrating LYDIAN in an undergradute distributed systems course.

## 8.1 Introduction

Distributed algorithms are algorithms that run concurrently on many interconnected processing elements called processors or processes. The algorithms are supposed to work correctly independent from the speed of the communication links and the structure of the network. Understanding such algorithms including their performance analysis and their correctness plays an important role in courses related to distributed systems, operating systems and computer networks. Mostly, students try to achieve an understanding of the algorithms control flow and its performance by following the explanations on the board and the pseudo-code description presented in a technical book or paper. This approach often suffers from the large amount of data describing local state and complex interaction between processes. Simulation and animation of distributed algorithms give students the possibility to experience how the distributed algorithms evolves over time and test the algorithm under different system behaviour. Compared to executing the algorithms on a real system, the animation and simulation allows the student to interact with the system state, pause the animation and execute an instance of an algorithm multiple times. A simulator also enables students to trace behaviour which under real circumstances rarely occurs, but is important to understand the correctness or asymptotic behaviour of the algorithm.

In this article we present LYDIAN an environment to support the learning of distributed algorithms. LYDIAN provides a database of distributed algorithms and respective continuous animations. Students can write their own algorithm implementation in a high level language and test it with any arbitrary interconnection of processes. The provided animation framework allows interactive demonstrations of distributed algorithms. The animations do not use a fixed interconnection of processes, but allow the students to create their own networks descriptions in a visual way and apply them to the respective algorithm and animation. This way teachers can use LYDIAN in various ways depending on the level and background of their students.

**Related Work.** Compared to advanced system simulation tools like [Khanvilkar and Shatz 2001] LYDIAN focuses on the educational aspects of algorithm visualisation which are mainly to support the student in reasoning on the analysis of the distributed algorithm. At the time we introduced the concepts behind LYDIAN [Papatriantafilou and Tsigas 1998] and our work on building an animation framework [Koldehofe et al. 1999],[Koldehofe 1999] for distributed algorithms there was only one known attempt towards a a set of animations of distributed protocols for educational purposes, ZADA [Mester et al. 1995], based on the animation package Zeus, a Modula-3 based system for specialised platforms. The

effort resulted in a small archive of protocols, for each of which the set of views is fixed and the implementation is the same program as the animation (this implies essentially fixed timing, workload, etc).

Of relevance was also the interesting work by Ben-Ari in [Ben-Ari 1997] and [Ben-Ari 2001]. There, the focus is on providing a framework for writing distributed algorithms (in a portable language) that allows students to interact with the states of a process and this way understand state changes and data structures of the algorithm. Subsequently, more tools with emphasis on different educational aspects evolved.

VADE [Moses et al. 1998] is a system that supports algorithms to be executed as Java processes on a server, and providing the client with a consistent view on algorithm events that happens on the server. The visualisation is based on WEB approach where users can view on a web page the visualisation of a selected algorithm by downloading the respective Java client. The animations supports multiple views, but makes no distinction between views for special educational purposes. The approach is mainly designed to make students view a prepared protocol, but not to implement protocols on their own. It seems that it was even thought to prevent an observer from viewing the code behind an algorithm. To the best of our knowledge, there is no recent development of this tool.

In the contrary ViSiDiA [ViSiDiA 2000] supports, like LYDIAN, an integrated approach of simulation and animation of algorithms and in this respect covers closest the aspects addressed by LYDIAN. The interconnection of processes is abstracted by a communication graph model, which can be created interactively by the user. The provided algorithms implemented in Java can be run on top of the selected network. Hereby, the code for processes is simulated with Java threads. Users can also create own protocols by using the provided library functions. However, the user has no influence on defining timing behaviour for communication links of the network. The animations show the graph model visualising events and states by displaying labels attached to links and processes. The visualisation mainly addresses to visualise the current states, but does not provide the user with information on other issues, such as causal relations and message complexity.

The work presented in [Schreiner 2002] provides a nice object oriented framework which allows a simple specification of protocols in Java. The specification protocols reflect the automaton model presented in textbooks on distributed algorithms such as [Lynch 1996]. The animation, because of its non-continuous nature, cannot give the user a picture about actions that happen concurrently and it does not address other aspects in educational visualisation. Moreover, the network is specified with the definition of a process, i.e. the code for each process identifies the respective neighbours of processes.

**Organisation of this article.** This article is organised to give first a general overview of LYDIANs components. In the following sections we present in detail the framework used for the provided animations of LYDIAN, where we give special attention to the educational aspects of the animations (see Section 8.3). Further, in Section 8.4 we introduce how LYDIAN supports the creation of own protocols with LYDIANs simulator. The concepts behind are illuminated on a simple example, presenting how to implement a broadcast algorithm with LYDIAN. Section 8.5 describes a case study intgrating LYDIAN in a basic distributed systems course. We evaluate a distributed system assignment in which students used LYDIAN to implement their algorithms. In our study neither the teachers nor the students had earlier class experience with LYDIAN. Section 8.6 we discuss an extension of the visualisation framework based on Virtual Reality technology to increase possibilties for user interactivity. Finally, in Section 8.7 we present our conclusions and future work.

## 8.2 An overview on LYDIAN

LYDIAN is intended to serve a wide range of educational purposes. For instance, one can use LYDIAN (i) to give a demonstration of prepared animations, (ii) to let students create their own network structures, which can be linked to the respective animations, or (iii) to let students create own protocols, which can be executed on LYDIANs simulator. The interaction with LYDIAN is based on a graphical user interface (GUI) written in TCL/TK [Ousterhout 1994] which allows to access LYDIANs archive of created created resources as well as to create own resources.

Since in the execution of a protocol there are many components involved, LYDIAN introduced the concept of *experiments* in which the user can describe the properties for relevant components. An experiment contains information about

- the *protocol* the user wants to execute,
- the underlying *network structure* describing how processes are interconnected and the characteristics of the timing behaviour,
- a *trace file* in which during the execution of the algorithm significant events are stored which can be traced by the user,
- and an *animation* which can give a graphical representation of the events in the trace file.

The experiment is abstracted by a single window containing all experiment specific information. The user interacts with the experiment by pressing buttons representing different actions or modification choices. There are two actions a user can

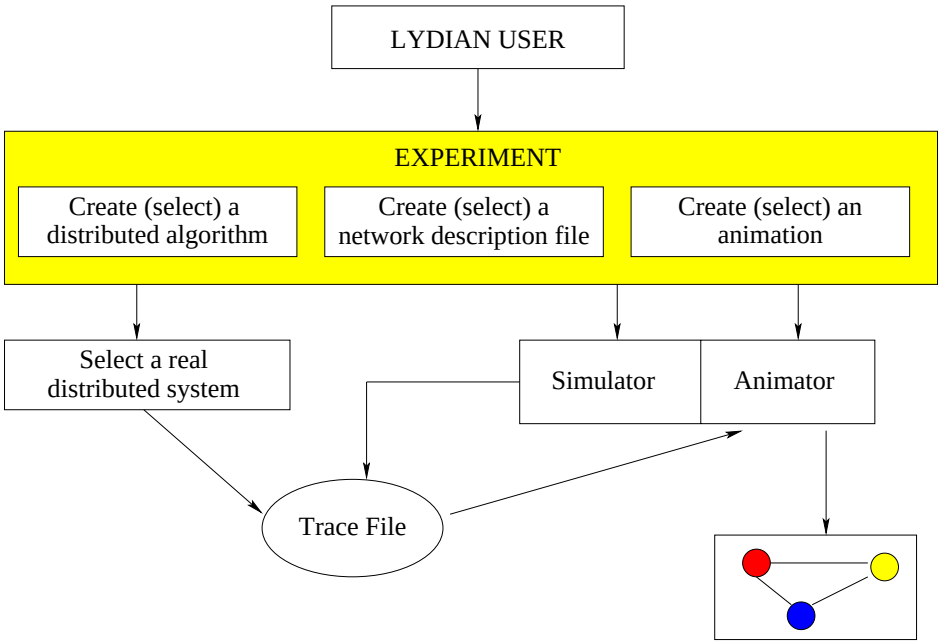



Figure 8.1: An overview on LYDIANs functionality from a users perspective.

perform on an experiment. It is possible to “run” the experiment, i.e. the respective protocol will be executed as specified in the experiment, or one can “animate” an experiment, i.e. an animation also with respect to the specification in the experiment will be shown.

When a user selects to “run” the experiment, the simulator of LYDIAN will be started and in the protocol defined events written into a trace file. The user can specify in its experiment further data evaluating the protocol. It is even possible to add own events by adding debug lines to the protocol. In order to view the data the user can choose between graphical or text output for the animation. The text output is useful if users added own events and wish to see all information created by the simulator. However, most users will prefer the graphical visualisation which provides an animation with respect to the components selected within the experiments. We will describe the framework used in LYDIAN to show animations in more detail in Section 8.3.

The GUI of LYDIAN is designed such that the user can view all relevant information in one window and can change components of a selected experiment on the fly. For instance, in order to change the network, a user simply selects another network description file in the experiment and can run the experiment and view the

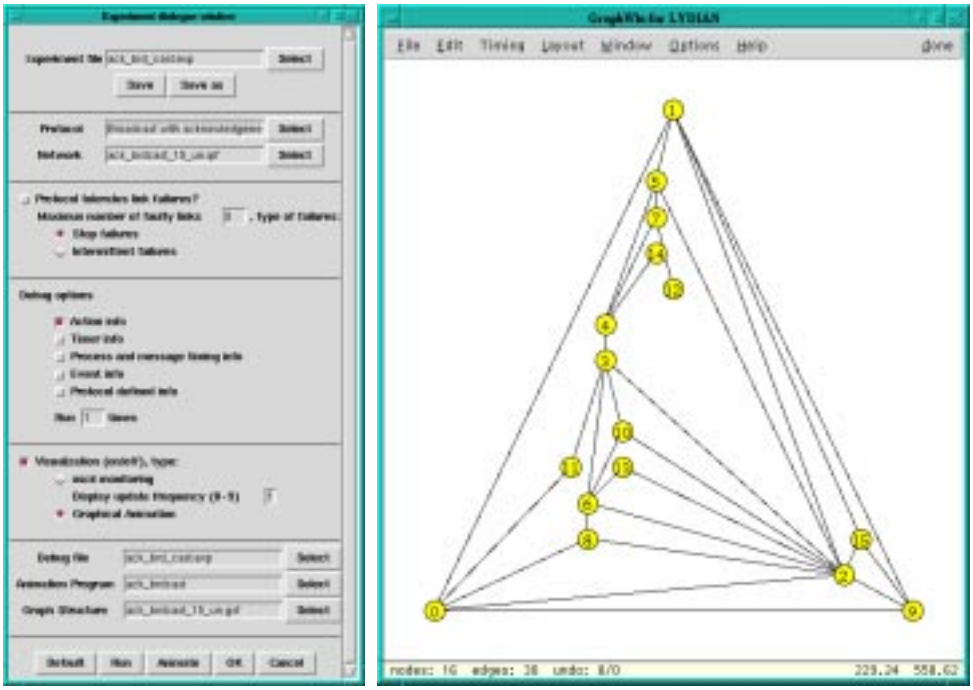


Figure 8.2: The experiment dialog and the graphwin drawing tool.

animation as before, but with respect to the new network structure.

It is important for students to experiment with different network structures. LYDIAN provides an easy visual way to create their own network descriptions. This component is based on LEDA [Mehlhorn and Näher 1999], a library for efficient data structures and algorithms, providing many algorithms to manipulate graphs and draw them efficiently. In LYDIAN the user simply draws a network based on a graph in which vertices represent processes and edges links between processes. Besides moving the vertices in order to achieve a pleasant layout, the user can apply a wide range of layout algorithms. For specifying the timing behaviour of the network, it is possible to choose among many different distributions valid for all processes, but also define a specific behaviour for a link or process. When saving the graph the files will be available for the simulator as well as for the animator. This way the user can see the same network in the animation as it was drawn in the graph editor.

## 8.3 The animation framework of LYDIAN

This section is on our work in building animations of distributed algorithms to demonstrate *(i)* the "key ideas" of the functionality of the algorithms, *(ii)* their behaviour under different timing and workload of the system, *(iii)* their communication and time complexities. The visualisation takes as input any possible execution trace of the respective algorithm, so that students (users) can view it in any possible execution that they can select. We propose the use of a set of views, which also take into account two inherent difficulties in understanding distributed algorithms executions. These difficulties stem from the absence of *global time* in the system, which implies

- that processes need to rely on their knowledge of *causal relations* among events in the system,
- and that in order to measure the length of an execution in time, we need to employ some mechanism related to the *dependencies* induced by each algorithm.

In the following we describe the set views that we provide for each animation and also motivate our decisions, by explaining the role each one plays in assisting the understanding of the algorithms. The code for all but one ("special") view is modularly used by all algorithms, as they are to assist in understanding issues which are common in all distributed algorithms. The idea behind the "special" view is to illustrate the *special concepts* for each algorithm (therefore the view needs to be different for each algorithm).

For our animation programs we use the Polka library [Stasko 1995], which is highly portable, friendly to use and has very good features for visualisation, including possibility for multiple views, speed tuning, step-by-step execution and callback events to assist interactive animation.

### Animation Views

It should be noted that all views evolve continuously as the execution of the algorithm evolves (continuous motion). The user can decide which views should be shown. The views can be selected by a menu window. Also a further control window enables the user in changing the speed or even halt animations in order to watch interesting parts or skip uninteresting parts of the algorithms execution. Moreover, the user has the possibility in zooming into interesting parts of the animations as he/she can move to any area of a view. This is important since by nature some animations will not be able to take place in a bounded window frame because the animator has not any previous knowledge of further executions of the

algorithm. With exception of the basic view, in which an individual animation for each algorithm was developed, the offered views were designed such that they are transferable for any distributed algorithm for a message passing system although they allow some specifications. Thus further development will have to concentrate only on the main ideas of algorithms.

The accompanying figures[1] illustrate a snapshot of the animation of an execution of the ECHO algorithm (broadcast with acknowledgements) [Tel 1994]. The problem and the algorithm are as follows: One process(or) needs to broadcast a message to all the others and to also know when all have received it. It can only communicate with its neighbours in the network, so it sends the message to them. Each process, upon receiving the broadcast message for the first time, propagates the message to its other neighbours and waits to receive acknowledgements from all of them. Once, a process received all acknowledgements, it starts sending its own acknowledgement to the one process from which it received the message for the first time. Any process receiving the broadcast message again acknowledges immediately to that sender and does not propagate it again.

**Basic View (c.f. Figure 8.3).** It illustrates the basic idea of the algorithm, hence Basic Views of different algorithms most likely look different. However, for many algorithms it is of interest to see the state of processes and messages which are sent along links. This can be achieved by showing the communication network, by colouring its nodes (processes) according to their state, and by showing moving arrows which are coloured according to the kind of message sent along an edge (link). In the particular algorithm the Basic View shows the communication network, the propagation of the broadcast and the acknowledgement messages (arrows in green and blue respectively) and colours (green or blue) the nodes (processes) that have received the broadcast message and/or the acknowledgements, accordingly (initially all nodes are yellow, except from the one that initiates the broadcast, which is always shown in red). As the algorithm execution evolves, *waiting chains* are formed among processes. Each process in the chain waits for an acknowledgement from its next one in the chain. These chains also determine the *time complexity* of the algorithm. The edges between two consecutive processes in the chains are marked in red. In this particular algorithm they also form a spanning tree of the network at the end of the algorithm.

**Communication View (c.f. Figure 8.4).** This view assists in measuring the communication complexity of the algorithm and is often helpful in finding relationships between communication complexity and the structure of the communication graph.

[1] they are in colour, hence the reader may find them more explanatory if the file is printed in colour

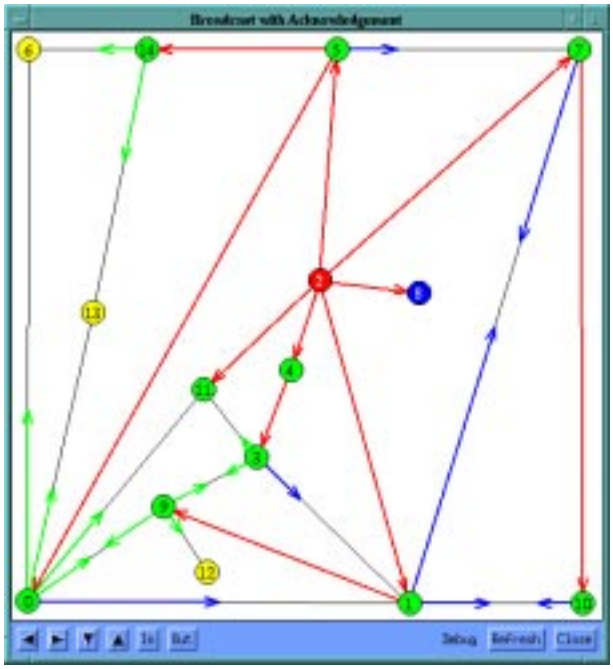



Figure 8.3: Basic view of the broadcast-with-acknowledgements algorithm animation.

It shows the contribution of each process(or) in the traffic (messages) induced by the algorithms execution and it also shows the average number of messages per process(or) during the execution. The number of messages are displayed in a bar chart where bars grow online with the number of messages sent by a process(or). In this example it is easy to observe that the amount of traffic induced by each process(or) is proportional to its degree in the communication graph (shown in figure 8.3).

For some algorithm it is also of interest to have a measure of the bit complexity of messages. The actual known maximum size of a message (represented in bits) is displayed below every processes bar. The size of a message is represented by a circle of which its area content is proportional to its message size. As the message size increases online the user is able to observe how fast message sizes are increasing.

In our example algorithm the bit complexity of a message was constant so that the bit complexity is not of any interest and thus not shown in Figure 8.4.

**Causality View (c.f. Figure 8.4).** It illustrates the causal relation between events in the system execution (arrows represent message transmission). It also shows how

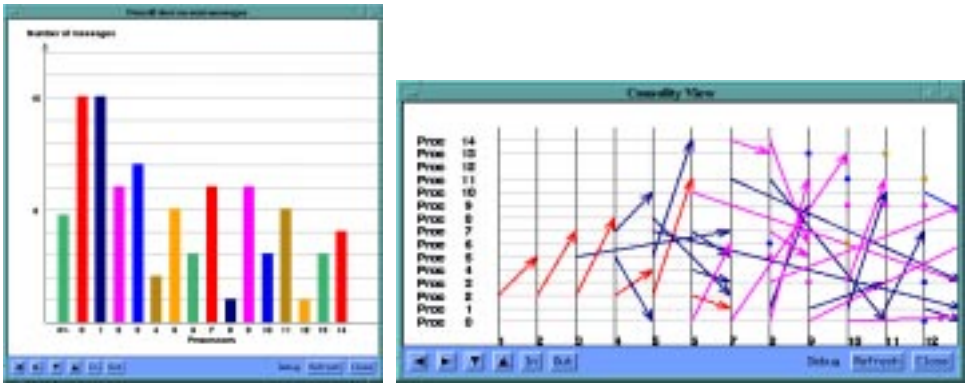


Figure 8.4: Views showing (a) the communication induced by each process(or) the average and (b) the causality and logical times (e.g. as would be seen by a monitoring process.

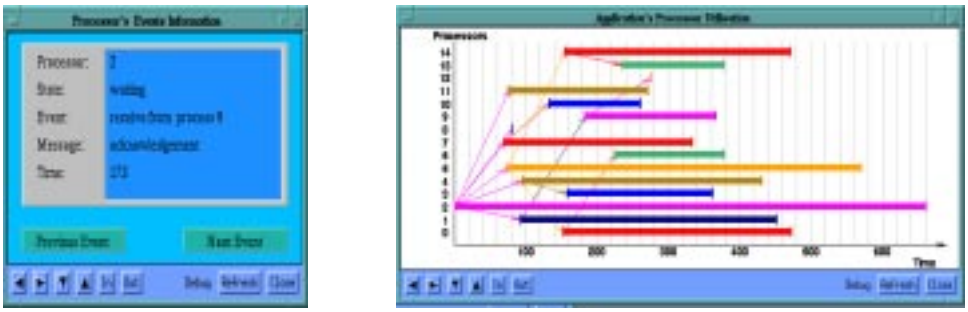


Figure 8.5: Process step view and process occupation view.

the processes logical clocks are incremented during the execution. Even though logical clocks are not used in all algorithms, the view is always available. Its purpose is to show how would a monitoring process view the execution, based on traces as would be given by each process separately. This is important, as the processes in a distributed system do not have global knowledge of time. Besides, as consecutive causally related events change colour, overlapping arrows with different colours visualise the degree of asynchrony in the execution. It should be noted that showing the maximum directed path in the resulting graph shows the length of the execution in units of message transmission times.

Naturally, only a part of the whole view can be shown in the window, but the user is able to go back and return (as well as to zoom in and out), as it is possible in all other views.

**Process Step View (c.f. Figure 8.5).** This gives the user the possibility to click on any node in the basic view window, to receive information about its status (state, last event processed, last message received/sent, etc.) at any point during the animation (or even after it has completed). It is also possible to view *interactively* the whole execution of the selected process. This can be done for any process in the system.

**Process Occupation View (c.f. Figure 8.5).** It shows in actual times, i.e. as given by the simulation trace, the period that each process is kept busy by the algorithm during the animated execution. If it is required by the algorithm it is also possible to distinguish how long a process was kept busy in a certain state. Therefore, a user may come to a better understanding of the algorithms time complexity by retracing, for instance with the Process Step View, why a specific process was kept busy for a long time. In this example, it can be easily observed that the initiator of the broadcast is the first to start and the last to finish. By receiving the acknowledgements from its neighbours, i.e. its children in the induced spanning tree, it knows that the broadcast message reached everybody, hence it terminates.

## 8.4 Writing own protocols

In courses assignments often require to write own protocols since the implementation helps the student to reflect in more detail on the main concepts behind an algorithm. However, writing own protocols on “real systems”, i.e. based on libraries as MPI [Gropp et al. 1996] can become a complex task requiring a lot of time for the students to understand the respective library. LYDIAN offers a simulator including a simple language and data structures that allow to implement quickly own protocols. Although the language is based on C syntax, it requires the user to know only about the most elementary commands, e.g. needed for loops and case analysis. The simulator helps the students to implement their ideas closely to the way they are represented in ordinary textbooks and avoids much of the overhead needed when learning a real system. The simulation also gives the possibility to test the program under user defined behaviour, needed to test cases which rarely occur, while in a real system a user may be unlikely to encounter the same situation. After the execution the user can trace significant information about the execution of the algorithm within the user experiment.

LYDIANs simulator called *DIAS* is based on the work for *DSS* [Spirakis et al. 1992]. The simulator is implemented by using the concept of Communicating Finite State Machines which allows to model the distributed system as a collection of processes communicating via communication links with messages. The simulation

is event driven, i.e. when an event takes place the process performs a computation depending on its state.

As a first crucial step a programmer has to consider the states, messages and events of the algorithm and understand how the algorithm is supposed to behave on the occurrence of an event. Therefore the programmer creates a table of transitions which associate with each pair, consisting out of a state and an event, the appropriate function call to be executed. The defined code associated with a function call is valid for all processes of the network observing the same event in the same state. This way it is not required to create special code for a process, and the programmer is supported to create code valid for any interconnection of processes.

The table of events is created interactively when a user chooses to create a new protocol within LYDIAN. Depending on the states and messages of the protocol, LYDIAN will ask for each possible transition the respective function call. Finally, the user is asked to specify a file which contains the function calls for the defined transitions. LYDIAN links all this information together and creates a protocol, which a user can select within an experiment and run it with different network structures.

In the following we will illustrate the concept behind the simulator of LYDIAN by outlining how to implement a simple broadcast algorithm. The main idea behind the algorithm is to send a piece of information from one source to all processes of the network. The process starting the algorithm sends a message containing this information to all its neighbours. A process receiving the information for the first time, sends this message to all its neighbours, while a process that received the information before ignores the message.

Recall that the first step to implement an algorithm is to identify the states and messages needed. In order to distinguish between the state where a process has not received a message yet and the state where a process can ignore the information, we introduce two states: *sleeping* and *received*. For propagation of the information only one message denoted by *broadcast* is needed.

From this information LYDIAN creates the following transitions, where on the right hand side of the arrow we have to specify corresponding actions inform of a function call:

```
sleeping  ×    INITPROTOCOL     →     start()
sleeping  ×  RECMES(BROADCAST)  →    forward()
received  ×        init         →    illegal()
received  ×  RECMES(BROADCAST   →     ignore()
```

Since *ignore()* and *illegal()* are just place holders for doing nothing or throwing an exception because the algorithm entered an illegal state, the only functions which remain to be implemented are *start()* and *forward()* (c.f. Figure 8.6).

The function *start()* is called when the protocol is initialised. We assume that

```
start()
{
    MESSAGE * mess;
    int i;

    debug(DEBUG,"START %d %d",me,get_time());
    for (i=0; i< PCB[me].adjacents; i++) {
        mess = create_message();
        mess→kind = BROADCAST;
        send_to(mess, i);
        debug(DEBUG,"SEND_BROADCAST %d %d %d",me,
              PCB[me].adjust[i].id, get_time());
    }
    new_state = RECEIVED;
}


forward()
{
    MESSAGE * mess;
    int i;

    debug(DEBUG, "REC_BROADCAST %d %d %d", me,
          CURMESS→from, get_time());
    for (i=0; i< PCB[me].adjacents; i++) {
        if (CURMESS→port ≠ i) {
            mess = create_message();
            mess→kind = BROADCAST;
            send_to(mess, i);
            debug(DEBUG, "SEND_BROADCAST %d %d %d", me,
                  PCB[me].adjust[i].id, get_time());
        }
    }
    new_state = RECEIVED;
}
```

Figure 8.6: The code implementing the simple broadcast algorithm.

the protocol is executed on a network such that only one single process is woken up by receiving the event *INITPROTOCOL*. To create such a network with the respective properties we can use LYDIANs graph drawing tool (see also section 8.2). The process receiving the event *INITPROTOCOL* is the initiator of the broadcast algorithm. It creates a new message and sends it to all adjacent vertices. Creat-

ing and sending a message is done by using the commands *create_message()* and *send_message()* from the simulators library. The function calls require to use the data structure *Message*, which allows us to define the type of the message, but also to send information with the message. In order to communicate with its neighbours a process must be able to know about the link information. Therefore, the simulator provides a data structure called *PCB* from which a process can extract information which must be known locally to a process, for instance the number of adjacent processes. Finally, the process changes its state by setting the variable *new_state* to the new valid state of the process.

The function *forward()* behaves similar to the function start(), however it occurs when a process receives a message. In our description the message will not be forwarded to the sender. The contents of the received message is available in the variable CURMESS.

For evaluation of the protocol in both functions we defined debug messages. After running the protocol the user can view within the trace file of the respective experience the global order of events as they were executed within the simulator.

The example demonstrates only the most elementary functionality of the simulator, sufficient for most introductory assignments though. However, the simulator of LYDIAN provides a wide range of functions as timeout events and the possibility to specify parameters with the protocol which also makes LYDIAN usable in graduate education and could even assist researchers to implement and test ideas of their own.

## 8.5 Course integration

In this section we present a case study of integrating a simulation-visualisation environment into a distributed system course. We present the evaluation of a distributed system assignment in which students used LYDIAN to implement their algorithms. In our study neither the teachers nor the students had earlier class experience with LYDIAN. The feedback received gives valuable information on what simulation-visualisation environments for distributed algorithms need to provide in order to be successfully used in class. We are not aware of any similar study in the area of distributed computing. However, the feedback we have received shows the significance of such evaluations to help users improve their performance and help them to acknowledge the wealth of tools they are provided.

The study is based on results taken from a compulsory basic undergraduate course in computer science and engineering -distributed systems-, at our university. The teachers taking part in this study have not used LYDIAN in class before, but were positive in using it from what they heard and read about it. The feedback re-

ceived shows that students succeeded well in the implementation of an algorithm. Many students experienced some behaviour of the algorithm they did not expect before, and this helped them in better understanding the algorithm. The feedback also shows that students should be asked to test their implementations by exploiting various parameters inherent in network topologies to improve their knowledge. Naturally, good simulation-visualisation environments should provide such possibilities. One should also remark that animation, as expected, is of good help. For students being able to successfully exploit features of these environments, places also high demands on the documentation of the respective tool.

### 8.5.1 Description of study

The evaluation is based on results taken from a basic undergraduate compulsory distributed system course at our university. We received answers from 50 students of the course. The questionnaire was anonymous. The main subject of most students was computer science and engineering, however there were also some students with other major subjects. Most students were in their final year of studies, but all of them had studied for at least two years in a program at our university. Hence, most of the students were experienced in programming. The percentage of female students participating in the study was around 10%. The age of students varied from 21 to 40, where most of the students were younger than 25.

For our study it is important to mention that neither the students nor the teachers had earlier class experience with LYDIAN. The teachers had to work out their own assignment, which would correspond to approximately one week of work for each student. The idea was to use LYDIAN for a programming assignment in which students had to implement some distributed algorithm based on elementary algorithms introduced in the course, i.e. the echo broadcast algorithm, logical clocks and voting. The students could choose to implement one of the following algorithms:

- leader election, based on an echo-broadcast approach,
- leader election, based on a voting approach,
- resource allocation, based on logical clocks.

The outline of the algorithms was given with the assignments, so the students essentially had to understand the algorithm and try to implement it. Parts of the algorithms, as the echo broadcast, were also available together with an animation, but needed to be changed to be usable within the algorithm. Most students decided to implement the algorithm based on the echo broadcast approach. This is probably because this concept seemed easier to realize. The algorithms were supposed to work on any arbitrary network structure.

For our study and evaluation our interest was focused on the following aspects:

- the students' performance in implementing an algorithm,
- how students test and reason about their implementation,
- whether/how much can LYDIAN help the students get an insight into distributed algorithms and their behaviour,
- whether students consider LYDIAN to be helpful,
- general feedback on the tools administration and maintenance.

### 8.5.2 Outcome and observations

The questions of this study and the answers received are summarised in Table 8.1 and Table 8.2. In Table 8.3, we examine the correlation between answers of students, in order to examine the relation between:

- the factors that helped the students most in getting a better insight into distributed algorithms,
- the help that the students got from using LYDIAN and their performance in carrying out the assignment,
- and the students performance in the assignment and their appreciation of LYDIAN.

Because of the anonymity of the answered questionnaire, we cannot associate the success of the students in their assignment with the answers that they gave to our questions. We simply trust their answers regarding their understanding of the assignment material. Below we discuss the results of our study.

- About 60% of the students were done with understanding how to use LYDIAN and with implementing and testing their solution in 1.5 working-days[2], while 80% of them were done in 2.5 to 3 working days. It should be mentioned here that the whole assignment was intended to take a maximum of 5 working days.
- Nearly half of the students tried the animation part —although it was not required in the assignment. As expected, animation stimulates the students' interest in studying.

[2] measured with 8 hours per working day

1. Approximately how long have you used LYDIAN?

| hours | 0-4 | 5-8 | 9-12 | 13-16 | 17-20 | 21-40 |
|---|---|---|---|---|---|---|
| students | 2 | 11 | 17 | 7 | 9 | 4 |

2. Which algorithm did you select to implement?

| Election with Echo | Election with Voting | Resource Allocation |
|---|---|---|
| 41 | 5 | 4 |

3. Approximately how long did it take you to understand LYDIAN's interface?

| hours | 0-4 | 5-8 | 9-12 | 13-16 | $\infty$ |
|---|---|---|---|---|---|
| students | 30 | 8 | 7 | 2 | 4 |

4. Approximately how long time did you spend on studying the algorithm that you had to implement?

| hours | 1 | 2 | 3 | 4 | 15 |
|---|---|---|---|---|---|
| students | 31 | 11 | 4 | 3 | 1 |

5. Approximately how long did it take you to implement and test the same algorithm in LYDIAN?

| hours | 0-4 | 5-8 | 9-12 | 13-16 | 17-20 | 21-30 |
|---|---|---|---|---|---|---|
| students | 13 | 17 | 9 | 5 | 2 | 4 |

6. Did you try to use the animation part of LYDIAN?

| yes | no |
|---|---|
| 24 | 26 |

Table 8.1: Questions and Answers

- Every third student experienced some behaviour/property of the algorithm they implemented, which they had not thought about before. Of those students the majority thought that this experience helped them to understand the algorithm better. The animation part of LYDIAN can be even more beneficial in this aspect, since the same execution can be seen multiple times, and difficult scenario can be "scrutinised" and digested better by the students' minds.

- Approximately 60% of the students tested their implementations on more than one network structures.

- Although some students experienced some difficulties with using specific parts of the tool –mainly where documentation was not sufficiently detailed–

7. When or after you implemented the distributed algorithm in LYDIAN, did you experience any behaviour of the algorithm that you did not think about before?

| yes | no |
|---|---|
| 17 | 33 |

8. If yes, did this help you understand better the algorithm or other material discussed in the course?

| yes | no |
|---|---|
| 12 | 5 |

9. How many different network topologies did you use when testing your implementation?

| number | 0-1 | 2-5 | 15 |
|---|---|---|---|
| students | 21 | 28 | 1 |

10. LYDIAN is useful for understanding algorithm in distributed computing.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 4 | 10 | 14 | 21 | 1 |

11. LYDIAN is easy to use.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 17 | 22 | 7 | 4 | 0 |

12. Which parts of LYDIAN do you think need to be improved?

| Documentation in general | Example implementations | Interface |
|---|---|---|
| 39 | 17 | 27 |

| Stability | Documentation on network creation | No comment |
|---|---|---|
| 14 | 21 | 5 |

13. What is your year of study?

| year | 3 | 4 | 5 | 6 | ? |
|---|---|---|---|---|---|
| students | 9 | 30 | 1 | 1 | 9 |

14. Age.

| age | 21-23 | 24-26 | older |
|---|---|---|---|
| students | 27 | 13 | 5 |

Table 8.2: Questons and Answers

Students who experienced some behaviour of their algorithm they did not expect before, gave the following answers to these questions:

1. Did you try to use the animation part of LYDIAN?

| yes | no |
|---|---|
| 10 | 7 |

2. Approximately how long did it take you to implement and test the same algorithm in LYDIAN?

| hours | 0-4 | 5-8 | 9-12 | 13-16 | 17-20 | 22 |
|---|---|---|---|---|---|---|
| students | 4 | 8 | 2 | 2 | 0 | 1 |

3. How many different network topologies did you use when testing your implementation?

| number | 0-1 | 2-5 | 15 |
|---|---|---|---|
| students | 7 | 10 | 1 |

4. LYDIAN is useful for understanding algorithm in distributed computing.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 1 | 1 | 3 | 12 | 0 |

The following groups of students thought as follows about LYDIAN being helpful:

1. Students tested their algorithm only with one network topology or were not aware of them.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 2 | 2 | 9 | 8 | 0 |

2. Students tested their algorithm with multiple network topologies.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 2 | 8 | 5 | 13 | 1 |

3. Students who tested the animation part of LYDIAN.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 0 | 7 | 4 | 13 | 0 |

4. Students who experienced behaviour they did not expect before.

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| 1 | 1 | 3 | 12 | 0 |

Table 8.3: Correlation between answers

the overall impression is that the class found the tool to be useful or relatively useful for understanding distributed algorithms.

- The majority of students who got better insight into their algorithm also tried the animation part. Maybe these students were more interested in the subject. However, the use of animations does not show any relation to the number of network topologies students used for testing purposes.
- Conforming to our expectation, the students who got more insight into the algorithm they implemented, tried more network structures in their testing. It seems worth the effort to try to stimulate students to test more and experiment more with their implementations. It is also good that LYDIAN's supported simulator provides this possibility.
- One main observation is that students who experienced unexpected behaviour of their algorithm mainly thought LYDIAN to be helpful.
- Students who used many network topologies did not think that LYDIAN is more helpful than those who did not use this feature. However the opinion is more biased, i.e. students who experienced with this feature have a stronger attitude whether they like or dislike LYDIAN in a course, while students who did not use it tended to be more neutral.
- A similar observation can be made for people who used the animation feature of LYDIAN.

An analysis of these results, from the perspective of seeing what such simulation-visualisation tools need to provide in order to be successfully used in class and how users (teachers, students) can be helped to improve the performance of the learning process, leads to the following lessons:

**Tools.** Since the "bottleneck" in understanding distributed algorithms is the actual concurrency and the parameters that can affect the step interleaving in each execution, it is very important for a tool which aims at facilitating the learning process in this area, to provide:

- means for the user to experiment by varying all parameters (this helps in revealing a scenario that the user had not thought about before),
- a good way to visualise concurrency,
- the possibility for a user to observe the same execution multiple times,
- a good documentation and good user guides.

**Users.** Since more experimentation is shown to be effective, it is important that instructors are explicit –e.g. as part of an assignment– in asking the students to use the visualisation/animation possibilities, as well as to experiment by changing the parameters of the system and by coming up with “special", unusual constellations.

### 8.5.3 Conclusion

Teaching is improved by pointing out unexpected properties/instances of the taught material. This is especially important in teaching distributed algorithms and systems, where there is a large number of parameters that affect the sequence of steps that a process will follow in each execution. LYDIAN was shown to be of good value in this respect, since it helped students to observe such instances, in an efficient manner.

Furthermore, teachers can help and get helped by using such tools in class to provide special case studies, to test cases, and to stimulate students to do own experimentation. Simulation and animation environments such as LYDIAN are shown to be useful in this respect, as well. Animation helps in understanding and also stimulates students.

LYDIAN helped in providing insight into the taught material, even though it had not been used in class by the teachers before and even though there were parts of the documentation which were not complete. The effort to improve on the supporting material –improved manuals, more examples– is expected to be appreciated and further increase the tools use-basis.

## 8.6 EnViDiA: a Virtual Reality extension

In this section we describe an extension of LYDIANs visualisation framework, which is intended to improve the interaction with the users. In difference to the approach described in Section 8.3 EnViDiA represents the communication structure in a 3D-model in which users are immersed. This way a natural interaction based on real world behaviour is possible. As it it is the case for the 2D-animations of LYDIAN, the algorithms are required to work correctly using any arbitrary interconnection of processes represented by a communication graph. However, in contrast to ordinary 2D-worlds, complex non-planar graph models can be nicely represented in three dimensions with the perspective adapting to the movements of the user. Further, within such a world the orientation is facilitated providing spatial sound. It assists the user becoming aware of the important system events.

Students working within such an environment can be more active since they walk or fly through the distributed system world in a game like scenario. EnViDiA has been developed by undergraduate students within the CAVE, an immerse VR

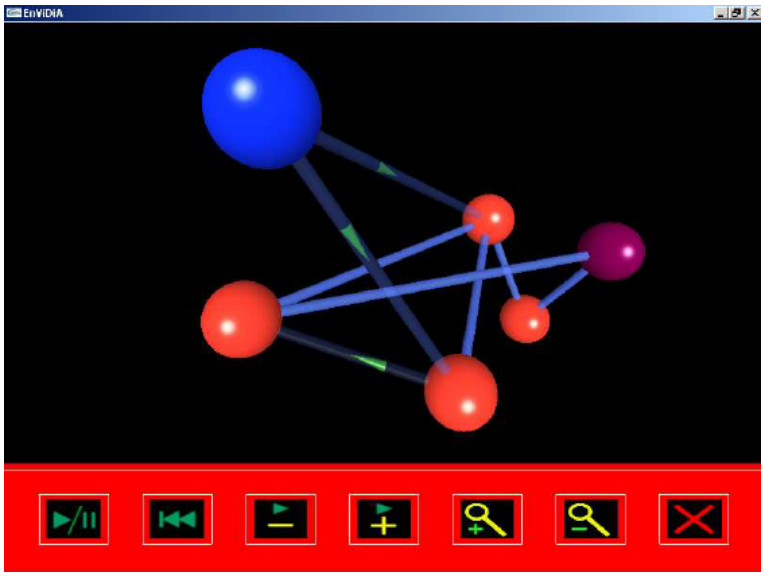



Figure 8.7: This screenshot shows the EnViDiA interface on a desktop computer executing an algorithm based on resource allocation.

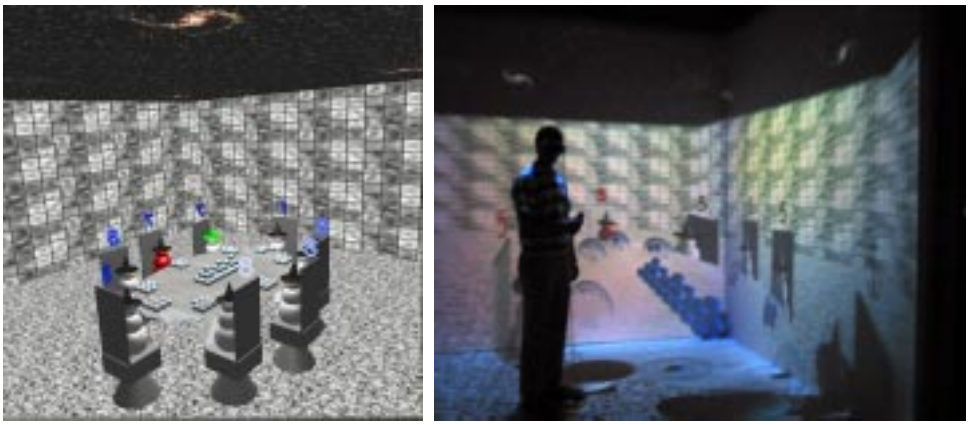


Figure 8.8: An animation based on a student project allowing multiple users to collaborate on the concept of self-stabilisation.

environment. The animation framework is based on the problems the students experienced themselves when studying the distributed algorithms for the first time. Although EnViDiA is intended to be used in immerse VR environments, the tool can also be used in a simpler version on ordinary desktop computers supporting 3D-graphics. At its current state EnViDiA supports three distributed algorithms namely simple broadcast, broadcast with acknowledgement and resource allocation based on the algorithm by Ricart and Agrawala. These algorithms are taught in a basic distributed system course at Chalmers University of Technology. Besides adding more algorithms and evaluating the tool at its current state, the main focus is on providing features to support multiple user collaboration, which are tested at the distributed concept of self-stabilisation [Koldehofe and Tsigas 2001]. The focus here is on removing the constraints of the traditional communication model in order to allow more user interaction with the distributed concept and among multiple users themselves (c.f. Figure 8.8).

## 8.7 Conclusion

Although simulation and animations cannot be a replacement for students to study carefully material presented in textbooks or classes, it can well assist the student in perceiving a better understanding on the functionality of the algorithm as well as to reflect on its performance behaviour and correctness. LYDIAN provides an extensible framework which includes a wide range of material sufficient to cover a big part taught in a distributed system course. It is freely available for Linux and Unix platforms, while a Win32 version is currently under development. LYDIAN has been successfully tested at various universities in courses on distributed systems for students of various backgrounds.

Besides improving the availability for LYDIAN to various platforms, our focus is to provide more teaching material based on the feedback we received from the students using LYDIAN. Further, we are working on several projects which on the one make it easier to develop own animations, but on the other hand also support collaborative work among students to learn distributed concepts.

# Nine

# Using Actors to Teach Self-Stabilisation

Boris Koldehofe
and
Philippas Tsigas

**Abstract**

This paper[1] describes an animation which uses dramatisation in order to teach a concept of distributed computing called self-stabilisation. Dramatisation is a technique which allows the audience to interact with the concept to be taught in class by representing the concept as a play. The animation of this paper uses human actors to play the role of processes in a distributed system. We present an evaluation of the method based on feedback we received from a graduate course on distributed systems. The feedback shows that the animation and the method have been well perceived by the students participating.

## 9.1 Introduction

We describe and evaluate an experiment where actors were used to simulate the behaviour of processes in a distributed system in order to explain the concept of self-stabilisation in a graduate course on distributed systems.

[1] A version of this paper is published in the proceedings of the 6th Annual SIGCSE/SIGCUE Conference of the ACM on Innovation and Technology in Computer Science Education (ITiCSE'2001).

A self-stabilising system is one that ensures that the systems behaviour eventually stabilises to a safe subset of states regardless of the initial state. Protocols satisfying this elegant property, which enables a system to recover from transient failures that can alter the state of the system, are often hard to understand, especially for students that have not studied distributed computing and systems before.

The experiment was part of an introductory course on distributed computing and systems for graduates in October 2000. The purpose of this interactive animation was to introduce to the students the basic concepts behind self-stabilisation (eligible states, transient faults, execution convergence) before their formal introduction.

All of the students had a degree either in mathematics or computing science and had taken a course on algorithms before. However, most of the students did not have a background in distributed systems or distributed algorithms. The latter was not only the motivation for preparing this method of presentation but also what made this a challenging effort.

The feedback from the class was that the concept and this teaching method were very well received. We could observe that their understanding evolved to the point that they were able to successfully come up with ideas for solutions and argue for/prove their correctness. As suggested in [Ben-Ari and Kolikant 1999], dramatisation of executions can help the students to understand new issues and complications. This work shows that this is true even for graduate level courses. In our experiment we could conclude that dramatisation can be almost as powerful as a programming exercise in the teaching process; sometimes even more efficient, especially when we need to teach new concepts to an audience with diverse educational backgrounds. In analysing the results of our method we make a combination of the qualitative and quantitative approaches [Kolikant et al. 2000].

## 9.2 Self-stabilisation

The self-stabilisation paradigm, first introduced by Dijkstra [Dijkstra 1974], defines a system as a self-stabilising one if it can recover following the occurrence of a fault that puts the system in an arbitrary state. The self-stabilising system will stabilise to a legal system state within finite amount of steps when faults stop. Hence, even though the system might be negatively affected by a failure, e.g. a power failure or a malicious process, once the failure ceases the system will start functioning again as desired after a finite number of system steps. This property is of big importance for systems like the Mars Polar Lander; for these systems it is desirable that they have the capability to fulfil their mission, in a timely manner, in the presence of failures, or accidents with no need for human interaction.

Formally, we define self-stabilisation, for a system $S$, as a property with respect to a predicate $P$ over its set of configurations. The predicate $P$ depends on the task that the system is executing. For instance, when the task is mutual exclusion, the predicate $P$ is: there is at most one processor in the critical section. A system is self-stabilising with respect to a predicate $P$ if starting in any configuration of $S$, the system is guaranteed to reach a configuration satisfying $P$ within a finite number of state transitions and $P$ is a stable property (closed) under the execution of $S$ [Dolev 2000; Schneider 1993].

Although self-stabilisation was introduced in 1973 its importance was not realised until 1983 when Lesley Lamport emphasised in [Kolikant et al. 2000] the importance of this work by Dijkstra. Since then self-stabilisation has evolved to one of the most active research fields in distributed computing and became an important concept to theoreticians and practitioners.

Traditionally, students dealing with distributed protocols and in particular self-stabilising protocols as introduced in this paper, have problems understanding the representation of the algorithms because of state explosion and the lack of a real-life metaphor.

## 9.3 Dijkstra's self-stabilising token-passing algorithm

The algorithm that was presented to the students, first presented in [Dijkstra 1974], assumes that processes are interconnected in a ring and each one of them can access only information that is local or shared with its direct neighbours (left, right). The algorithm has to ensure mutual exclusion among the processes, i.e. that only one process at a time can perform a special computation.

The solution uses the token passing method; there is one special entity, called token, that processes can circulate among themselves. A process has the privilege to perform this special computation whenever it holds the token. After finishing, it passes the token to one of its neighbours. There is a consistent direction of flow of the token (say, anticlockwise), to guarantee fairness among the way that the processes receive the token.

In an arbitrary state, the system may contain no tokens or more than one tokens. A self-stabilising solution guarantees convergence to the behaviour described in the previous paragraph, with only one token in the system that flows anticlockwise. The solution is illustrated in Figure 9.1.

The token is realised by the shared memory variables $x_1, \ldots, x_n$. The processes whose if-condition is true is the process holding the token. The algorithm assumes the existence of a leader and that processes have consistent sense of orientation that cannot be affected by failures. Inconsistency due to faults, i.e. more than

1. $P_1$ : **do** forever
2. **if** $x_1 = x_n$ **then** $x_1 := (x_1 + 1) \bmod (n + 1)$
3. $P_i (i \neq 1)$ : **do** forever
4. **if** $x_i \neq x_{i-1}$ **then** $x_i = x_{i-1}$

Figure 9.1: Solution to Dijkstra's self-stabilising token-passing algorithm

one processes are holding a token is resolved by all non-leader processes $P_i$ equalising the two values shared with their respective neighbours. Hence, the algorithm converges to a situation where $x_1 = x_2, \ldots, x_{n-1} = x_n$ meaning only $P_1$ holds the token, i.e. the system stabilises.

## 9.4 Dramatising Dijkstra's algorithm

The idea of the experiment was inspired by a childrens game where children seated in a cycle use apples to synchronise in order to ensure that only one child is speaking at a time. The experiment consisted out of several acts. Each act used the same representation of the distributed system: a table with three actors (processes), in which apples by each actress[2] could be placed between herself and the two neighbouring actresses into two small baskets (variables $x_i$, $x_{i-1}$).

### Act 1
### (Perfect system)

In this act the actresses aimed at: i) introducing the non-stabilising algorithm, ii) explaining the token-passing idea and iii) motivate the problem. Initially, one apple was placed between two actresses. The actress that found the apple in her left hand side basket took the apple and started speaking (i.e. the actress holds the token). While speaking the actress explained to the students why she was allowed to speak and what were the next steps of the protocol that she was going to follow. Having finished the actress put the apple to her right hand side, i.e. between herself and the anticlockwise-next actress, thus forwarding the token. The next actress was then enabled to act and repeated the same actions as the previous actress. The three actresses continued this act for some time.

[2]We assume in this paper all our actors to be female.

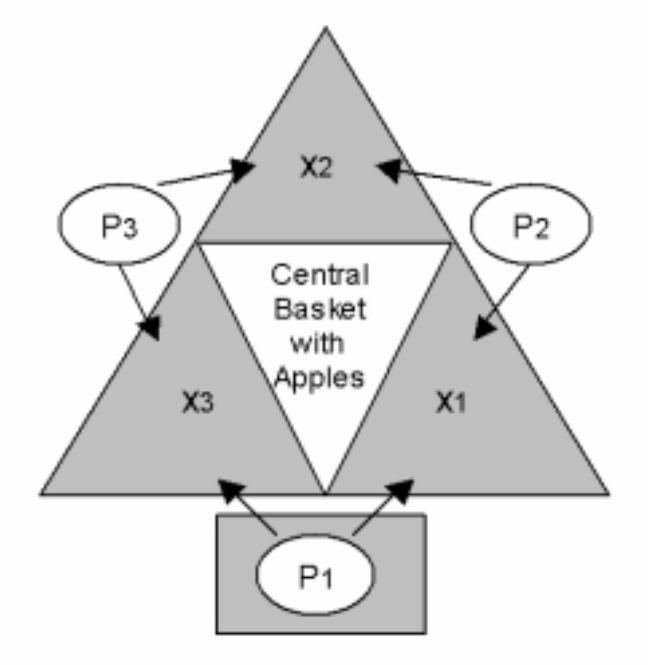



Figure 9.2: The table with actors $P_1$,$P_2$ and $P_3$ where $X_1$, $X_2$ and $X_3$ denote the number of apples. Actress $P_1$ is the leader.

**Act 2**
**(Introducing transient failures into the system)**

After a while the play moved on to the next act where one of the actresses - we call her the evil actress - started maliciously either adding new apples or removing apples thus bringing the system into arbitrary states. The purpose was to introduce the problems that arise when transient faults occur. Finally, the lecturer discussed with the students these problems that occurred because of the evil actress and the inability of the previous algorithm to cope with them.

**Act 3**
**(An attempt for stabilisation)**

In the third act the actresses proposed a solution that could potentially guarantee self-stabilisation (c.f. Figure 9.2). One (predefined) actress became the leader, a property which could not be affected by the evil actress. Initially, between each pair of actresses there were no apples while in the middle of the table there was a

central basket with many apples (the latter is a technicality, in order to give to the actresses an "unlimited" number of apples to apply their rules of the game, so that they do not have to worry for an additional constraint in attempt to come up with stabilising rules). Recall that each actress can only see the apples that are placed between herself and her immediate neighbours. The leader first checked the number of apples in each of her hand side baskets. If there were equally many apples in both baskets (e.g. no apples at all, as it was initially the case in our experiment) the leader could start speaking, i.e. she was holding the token (in our experiment every actress was explaining to the class what she was doing, the new rules in the system, etc). When the leader finished her story, she took an apple from the middle of the table and put it into the basket to her right, thus forwarding the token to the right. Each of the other actresses was allowed to speak only when the number of apples in her right side basket was not equal to the number of apples in her left side basket. The rest of the rules for them were, however, the same; after an actress finished speaking (explained her acting), she had to add an apple (taken from the pile in the middle) into the basket at her right hand side. The actresses continued for a while with this behaviour in order to give the audience the possibility to understand the algorithm so far.

Next, the evil actress started adding/removing apples to/from her left (evil actions), being able to speak at the same time while another actress was speaking. However, the system could converge to a stable state, i.e. within some amount of time after the evil actions stopped, only one actress at a time was able to speak.

### Act 4
### (Students find the “bug” and finalise the solution themselves)

In this act the students had the ability to interact with the system and introduce the faults themselves. After a short discussion, the lecturer invited the students to place apples in a way that the system would fail. However, the students could observe that the system managed to recover apart from one critical scenario: a student placed between the leader and her left neighbour apples such that the leader had more apples to her left and the neighbour had more to her right (c.f. Figure 9.3). For the system to stabilise, the left neighbour of the leader should equalise the number of apples on her left and right side. However, at the beginning we used a simplified, “buggy” version of the algorithm not working in this case. The students managed to encounter the problem and fixed the “bug” by correcting the rule. Any non-leader actress was now allowed to speak whenever the number of apples at her left side was not equal to the number of apples at her right side, and then the actress had to adjust the number of apples at her right side to be equal to the number of apples at her left side. Having the possibility to interact directly with the system,

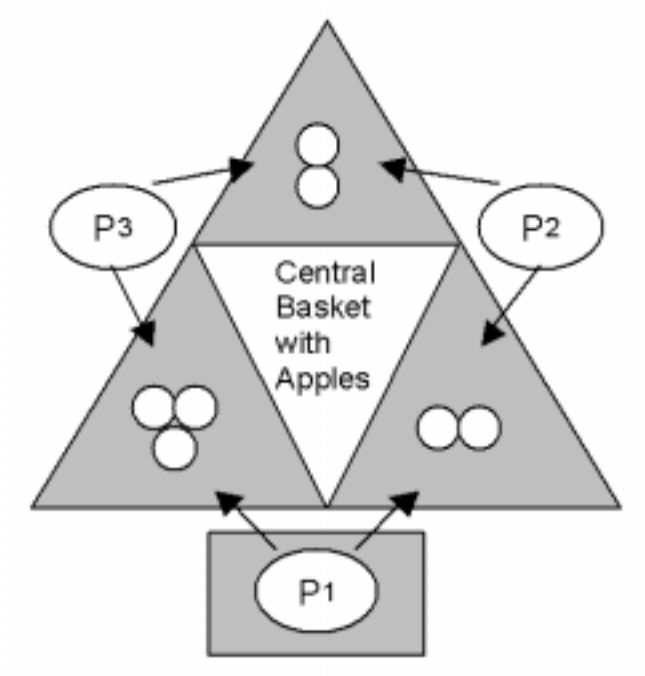



Figure 9.3: Deadlock situation due to wrong algorithm used by $P_3$.

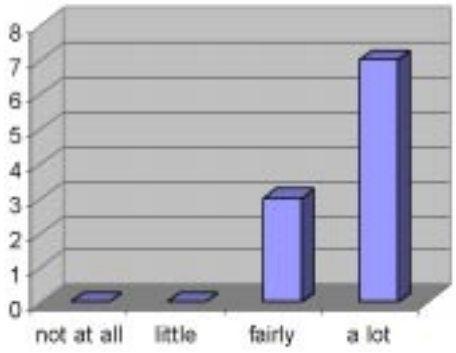
the students were also able to argue about the stabilising properties of the algorithm also in presence of concurrent actions.

## 9.5 Evaluation

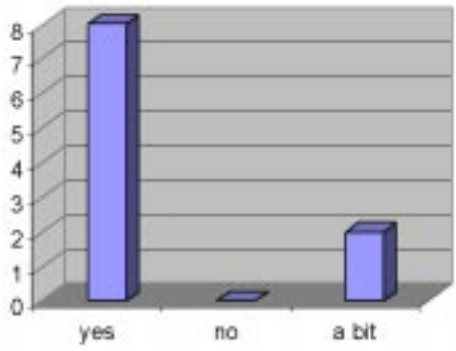
In analysing the results of our method we make a combination of the qualitative and quantitative approaches.

In the course we had 13 students participating 11 of them handed back the questionnaire that was given to them. All of the students had a degree either in mathematics or computing science and all of the students have had a course on algorithms before. However, the majority of the students did not study distributed systems or distributed algorithms before.

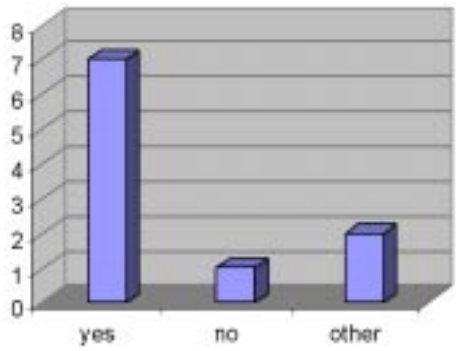
Our feedback from the class has been that the concept and the method were very well received. We could observe that the students' understanding evolved to the point that they were able to successfully come up with ideas for solutions and argue for/prove their correctness.



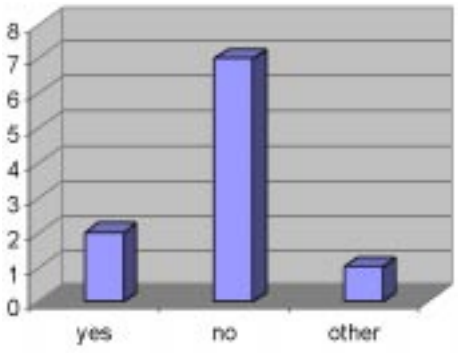

(a) How much did the animation shown in todays lecture help you in understanding the algorithm?




(b) During the animation you had the ability to interact with the animation. Did this help you to understand the algorithm better?




(c) Did it help you that the animation was run by human beings than computers?




(d) Did the fact that the animation was presented as a childrens game help?

Figure 9.4: Some of the questions and results from the questionnaire handed out to the students.

Knowing that many of the students were used to formal representations and had a strong background on formal methods, at the beginning, we were wondering whether our method could help or would be appreciated. Seven students answered that the dramatisation helped them a lot to understand the new concept, while only three answered that the dramatisation helped them fairly. None of them thought that it did not help at all or only a little.

As a next step, we were interested to know which methods used in the animation were the significant ones that helped the students understanding. Before executing the experiment, we were guessing that interactiveness would increase the effectiveness of the animation. The students' answers (eight said that the interactive part helped, while two said it helped only a little) seem to confirm this hypothesis. One student wrote that the bug, which is described in the previous section, leading to the correct and final version of the algorithm helped the most.

Another aspect that we were interested in evaluating, was, how much more efficient it was to have a dramatisation with human beings, compared to an interactive computer animation like the ones presented in [Koldehofe et al. 2000; Moses et al. 1998; ViSiDiA 2000]. To be able to answer this question we also asked whether the students have seen any computer animations before, which was true for seven and six of them were also familiar with educational computer animations. The answers (c.f. Figures 9.4), show that the majority was in favour of an animation with human beings. Some of the comments indicate that the spontaneous and not predefined reactions of human beings provide more information.

When designing the experiment, we thought that by representing the system and the protocol as a childrens game we could enhance the effectiveness of the animation. The students seem to disagree: seven students answered that the childrens game representation did not increase the effectiveness of the dramatisation, while only two answered that this representation actually helped them.

Since most of the students had a very positive attitude towards this dramatisation, it was not a surprise that all students answered that they believe that animations in general can be of big help in understanding the behaviour of algorithms or distributed algorithms.

## 9.6 Conclusion

Analysing further our experiment and the way it was received by the students, we can conclude that dramatisation can be almost as useful and powerful in the learning process as a programming exercise even in graduate courses; sometimes even more efficient, especially when we need to teach concepts to an audience with different backgrounds. New ideas are transmitted faster, while the students, by being active (and interactive) participants, have the possibility to point out the issues, which they find confusing, and to obtain more viable knowledge.

Although students seemed to favour a human animation we cannot say for sure that computer animations cannot be as powerful. We would like to continue this project by implementing the dramatisation in a virtual environment in which students can interact with virtual actors and apples.

## Acknowledgments

We would like to thank: i) H. Sundell and Yi Zhang for being such great actors, ii) M. Papatriantafilou for her big help during the writing phase of the paper, and iii) the graduate students that enthusiastically participated in the project.

# Part III

# Discussion

# Ten

# Conclusions and Future Work

The development of collaborative application requires a consideration of aspects addressed by different research directions. The focus of this work has been on supporting the development of collaborative applications by examining algorithmic and educational aspects of collaborative environments. The presented algorithmic solutions aim at providing a scalable, reliable and consistent communication media. The educational aspects have been addressed by exploring interactivity of collaborative environments to enhance the learning of distributed algorithms.

Peer-to-peer event dissemination and membership services offer interesting properties which support the design of scalable collaborative applications. Publish/subscribe and group communication, which have both been identified as fundamental communication paradigms for collaborative environments, can be implemented in unstructured and structured peer-to-peer systems. By relaxing the strong reliability guarantees, peer-to-peer systems provide failure resilience and good performance in spite of ongoing joining and leaving of peers. Moreover, peer-to-peer systems can be used to balance the load of disseminating events fairly among all peers.

In order to provide a notion of consistency to designers of collaborative applications, we have examined, analysed, and proposed resource-friendly peer-to-peer solutions which can provide a probabilistic guarantee for the delivery of events as well as support ordering of events. Important resources in peer-to-peer systems are determined by membership information and buffer space maintained by a process. We have shown that only a good configuration of these parameters can give a high delivery guarantee.

A critical point for supporting real-time interactions and ordering of events is the overhead in time to establish the order. We have proposed a model in which objects are clustered and the number of processes which are allowed to perform

concurrent updates per cluster can be controlled. Moreover, we have shown an algorithm for processes to access a cluster in a dynamic, decentralised, and fault-tolerant manner. In combination with causal ordering, the algorithm provides a low overhead in message ordering.

While fair load distribution in peer-to-peer systems often means that every peer performs the same amount of work, for publish/subscribe systems fair load distribution should ensure that each peer performs the amount of work proportional to its interest. The proposed topic-aware subscription management facilitates the balancing of work and can also be used as a bootstrapping procedure for membership maintenance of dissemination algorithms. As a consequence, we may not any longer distinguish between dissemination schemes for structured and unstructured systems, but mention whether the membership was bootstrapped by a structured or unstructured membership protocol.

The techniques introduced to control concurrent updates, to balance the load, and to bootstrap membership serve as an indication for future work directions. Besides supporting different ordering algorithms, cluster management can be explored in a more general setting in order to restrict concurrent access to shared resources. Restricting the access to resources may be combined with balancing techniques in supporting fair data dissemination. Random routing has been explored to balance the dissemination load and bootstrap the membership in publish/subscribe systems. It is likely that this technique can be used in a more general fashion to initialise data structures in structured peer-to-peer systems. Also for the configuration of gossiping protocols there remain a couple interesting problems such as understanding the relation between the size of the fanout, the size of the view, and the dynamics in updating the view to guarantee delivery with high probability.

For many of the presented solutions it is not clear how they perform in a mobile network. A common difficulty is typically the lack of point-to-point communication. Dynamic peer-to-peer solutions supporting locality may be interesting in supporting mobile middleware services. Topic awareness and cluster consistency management partially fulfil these requirements.

The presented algorithmic solutions for collaborative environments can support interactions in educational environments. In particular, we have studied levels of interactions in simulation/visualisation environments to support the learning of distributed systems. The visualisation and simulation framework of LYDIAN was designed to allow users (students/researchers) to test the same protocol under different networks and timing behaviour without having to change the protocol or the respective animation. The animations evolve continuously and provide the users with information about several aspects, which are important for the understanding of distributed algorithms. Some additional work was carried out on the representation of the communication graph. A 3D-environment such as Virtual Reality can

allow a better representation of non-planar network structures. Users being immersed in the animation can also improve the interaction with the concept.

As an alternative method to the traditional communication graph visualisation, dramatisation was shown to be helpful in improving the interaction between users and the concept. Dramatisation was especially successful in class using real actors, but also showed to be a good concept for creating collaborative animations in 3D-environments like the CAVE. In fact, applying the algorithmic ideas to support multiple levels of interactivity does not apply only to the study of virtual actors; it can also be integrated to support collaborative and real-time interactions in LYDIAN.

Besides focusing on the functionality, e.g. by increasing the level of interactivity to LYDIAN, future work directions should address reducing barriers for using systems like LYDIAN in class. Indeed, many educational tools do not suffer from their educational value, but from the effort needed to integrate them in courses. Following some of the suggestions proposed in [Naps et al. 2003] better feedback from users may be received and thus help to better satisfy the needs of the users. Some of the recent efforts for LYDIAN actually dealt with improving lecture material and facilitating the portability and installation of LYDIAN.

# Bibliography


Ahamad, M., Neiger, G., Kohli, P., Burns, J. E., and Hutto, P. W. 1995. Casual memory: Definitions, implementation and programming. *Distributed Computing 9*, 37–49.

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

Attiya, H. and Welch, J. L. 1994. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems 12*, 2 (May), 91–122.

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

Bailey, N. T. J. 1975. *The Mathematical Theory of Infectious Diseases and its Applications*, 2nd ed. Griffin.

Bajaj, S., Breslau, L., Estrin, D., Fall, K., Floyd, S., Haldar, P., Handley, M., Helmy, A., Heidemann, J., Huang, P., Kumar, S., McCanne, S., Rejaie, R., Sharma, P., Varadhan, K., Xu, Y., Yu, H., and Zappala, D. 1999. Improving simulation for network research. Tech. Rep. 99-702, University of Southern California.

Baldoni, R., Prakash, R., Raynal, M., and Singhal, M. 1998. Efficient $\Delta$-causal broadcasting. *International Journal of Computer Systems Science and Engineering 13*, 5, 263–269.

Ben-Ari, M. 1997. Distributed algorithms in Java. In *Proceedings of the 1st Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'97)*. ACM Press, 62–64.

Ben-Ari, M. 2001. Interactive execution of distributed algorithms. *ACM Journal of Educational Resources in Computing (JERIC) 1*, 2es, 2–8.

Ben-Ari, M. and Kolikant, Y. B. D. 1999. Thinking parallel: The process of learning concurrency. In *Proceedings of the 3rd Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'99)*. ACM Press, 13–16.

Bickson, D., Malkhi, D., and Rabinowitz, D. 2004. Efficient large scale content distribution. In *Proceedings of the 6th Workshop on Distributed Data and Structures (WDAS'2004)*.

Birman, K., Schiper, A., and Stephenson, P. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems 9*, 3 (Aug.), 272–314.

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

Birman, K. P. and Joseph, T. A. 1987a. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (11th SOSP87), Operating Systems Review*. 123–138.

Birman, K. P. and Joseph, T. A. 1987b. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems 5*, 1 (Feb.), 47–76.

Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E., and Randall, K. H. 1996. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*. 297–308.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices 30*, 8 (Aug.), 207–216.

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

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS) 19*, 3, 332–383.

Castro, M., Druschel, P., Kermarrec, K., Nandi, A., Rowstron, A., and Singh, A. 2003. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. Operating Systems Review, vol. 37, 5. ACM Press, New York, 298–313.

Cugola, G., Nitto, E. D., and Fuggetta, A. 2001. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering 27*, 9, 827–850.

Davis, R., Merriam, N., and Tracey, N. 2000. How embedded applications using an RTOS can stay within on-chip memory limits. In *Proceedings for the Work in Progress and Industrial Experience Sessions, 12th EuroMicro Conference on Real-Time Systems*. Stockholm.

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 1–12.

Dijkstra, E. W. 1974. Self-stabilizing systems in spite of distributed control. In *Communications of the ACM*. 11, vol. 17. 643–644.

Dillenbourg, P., Baker, M., Blaye, A., and O'Malley, C. 1996. The evolution of research on collaborative learning. *Learning in Humans and Machine: Towards an interdisciplinary learning science*, 189–211.

Dolev, S. 2000. *Self-stabilization*. MIT Press.

Dubois, M., Scheurich, C., and Briggs, F. 1986. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*. IEEE, 434–442.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. 2003. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR) 35*, 2, 114–131.

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

Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kermarrec, A.-M., and Kouznetsov, P. 2001b. Lightweight probabilistic broadcast. Tech. rep., IPSL, Lausanne. Jan.

Floyd, S., Jacobson, V., Liu, C.-G., McCanne, S., and Zhang, L. 1997. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ ACM Transactions on Networking 5*, 6 (Dec.), 784–803.

Frigo, M. and Luchangco, V. 1998. Computation-centric memory models. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*. SIGACT/SIGARCH, 240–249.

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

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. IEEE, 15–26.

Greenhalgh, C. and Benford, S. 1995. MASSIVE: A distributed virtual reality system incorporating spatial trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 27–35.

Greenhalgh, C. and Benford, S. 1997. 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*. LNCS, vol. 1242. Springer-Verlag, 113–128.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. A high-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing 22,* 6 (Sept.), 789–828.

Gummadi, K. P., Gummadi, R., Gribble, S. D., Ratnasamy, S., Shenker, S., and Stoica, I. 2003. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, 381–394.

Gupta, I., van Renesse, R., and Birman, K. P. 2001. Scalable fault-tolerant aggregation in large process groups. In *International Conference on Dependable Systems and Networks (DSN 2001)*. Gothenburg, Sweden, 433–442.

Hagsand, O. 1996. Interactive multiuser VEs in the DIVE system. *IEEE Multimedia 3,* 1, 30–39.

Heldal, I. 2004. Collaborative virtual environments: Towards an evaluation framework. Ph.D. thesis, Chalmers University of Technology.

Holdfeldt, P., Koldehofe, B., Lindskog, C., Olsson, T., Petersson, W., Svensson, J., and Valtersson, L. 2002. Envidia: An educational environment for visualisation of distributed algorithms in virtual environments. In *Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2002)*. ACM press, 226.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing 13,* 3, 259–290.

IEEE. 1993. *IEEE standard for information technology - protocols for distributed simulation applications: Entity information and interaction.* IEEE Computer Society. IEEE Standard 1278-1993.

Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. El Paso, Texas, 654–663.

Karp, R., Schindelhauer, C., Shenker, S., and Vöcking, B. 2000. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 565–574.

Kermarrec, A.-M., Massoulié, L., and Ganesh, A. J. 2003. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems 14,* 3 (Mar.), 248–258.

Khanvilkar, S. and Shatz, S. M. 2001. Tool integration for flexible simulation of distributed algorithms. *Software: Practice and Experience 31,* 14 (Nov.), 1363–1380.

Koldehofe, B. 1999. Animation and analysis of distributed algorithms. M.S. thesis, Universität des Saarlandes.

Koldehofe, B. 2002. Simple gossiping with balls and bins. In *Proceedings of the 6th International Conference on Principles of Distributed Systems (OPODIS'02)*. 109–118.

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

Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 1998. Building animations of distributed algorithms for educational purposes. In *Proceedings of the 3rd Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'98)*. 286.

Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 1999. Distributed algorithms visualisation for educational purposes. In *Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'99)*. ACM Press, 103–106.

Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 2000. LYDIAN, an extensible educational animation environment for distributed algorithms. In *Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2000)*. ACM Press, 189.

Koldehofe, B., Papatriantafilou, M., and Tsigas, P. 2003. Integrating a simulation-visualisation environment in a basic distributed systems course: A case study using LYDIAN. In *Proceedings of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'03)*, J. Impagliazzo, Ed. ACM Press, 35–39.

Koldehofe, B. and Tsigas, P. 2001. Using actors for an interactive animation in a graduate distributed system course. In *Proceedings of the 6th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2001)*. ACM press, 149–152.

Kolikant, Y. B. D., Ben-Ari, M., and Pollack, S. 2000. The anthropology of semaphores. In *Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2000)*. ACM Press, 21–24.

Kostić, D., Rodriguez, A., Albrecht, J., and Vahdat, A. 2003. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. Operating Systems Review, vol. 37, 5. ACM Press, New York, 282–297.

Kouznetsov, P., Guerraoui, R., Handurukande, S. B., and Kermarrec, A.-M. 2001. Reducing noise in gossip-based reliable broadcast. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*. IEEE, 186–189.

Kreijns, K. and Kirschner, P. A. 2001. The social affordances of computer-supported collaborative learning environments. In *Proceedings of the 31th ASEE/IEEE Frontiers in Education Conference*. 12–17.

Kshemkalyani, A. D. and Singhal, M. 1998. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing 11*, 91–111.

Kumar, V. S. 1996. Computer-supported collaborative learning: Issues for research. Tech. rep. Graduate Symposium, University of Saskatchewan.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*. 21, vol. 7. 558–565.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (Sept.), 690–691.

Last, M. Z., Daniels, M., Almstrum, V. L., Erickson, C., and Klein, B. 2000. An international student/faculty collaboration: the RUNESTONE project. In *Proceedings of the 5th annual SIGCSE/SIGCUE conference on Innovation and technology in computer science education*. ACM Press, 128–131.

Levine, B. N. and Garcia-Luna-Aceves, J. 1998. A comparison of reliable multicast protocols. *Multimedia Systems 6*, 5, 334–348.

Lin, J. C. and Paul, S. 1996. RMTP: A reliable multicast transport protocol. *Proceedings of IEEE Infocom*, 1414–1424.

Lucca, J., Romano, N. C., and Sharda, R. 2003. An overview of systems enabling computer supported collaborative learning requiring immersive presence (CSCLIP). In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS 03)*. 5–13.

Lynch, N. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Lynch, N., Malkhi, D., and Ratajczak, D. 2002. Atomic data access in distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*. LNCS, vol. 2429. Springer-Verlag, 295–305.

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzman, D. P., and Barham, P. T. 1995. Exploiting reality with multicast groups. *IEEE Computer Graphics and Applications 15*, 5 (Sept.), 38–45.

Mattern, F. 1989. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. 215–226.

Mauve, M. 2000. Consistency in replicated continuous interactive media. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW-00)*. ACM Press, 181–190.

Mehlhorn, K. and Näher, S. 1999. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England.

Mester, A., Herrmann, P., Jager, D., Mattick, V., Sensken, M., Kukasch, R., Ritter, A., Bunemann, S., Unflath, P., Bernhard, M., Austel, F., Alders, T., and Rohrbach, A. 1995. Zada: Zeus-based animations of distributed algorithms and communication protocols. *http://ls4-www.cs.uni-dortmund.de/RVS/zada.html.*

Miller, D. C. and Thorpe, J. A. 1995. SIMNET:the advent of simulator networking. In *Proceedings of the IEEE*. 83, vol. 8. 1114–1123.

Mosberger, D. 1993. Memory consistency models. Tech. rep., University of Arizona. Nov.

Moses, Y., Polunsky, Z., Tal, A., and Ulitsky, L. 1998. Algorithm visualization for distributed environments. In *IEEE Symposium on Information Visualization*. 71–78.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms.* Cambridge University Press.

Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rößling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R. J., Anderson, J., Fleischer, R., Kuittinen, M., and McNally, M. 2003. Evaluating the educational impact of visualization. *ACM SIGCSE Bulletin 35*, 4, 124–136.

NIST. 2002. Secure hash standard. Tech. Rep. FIPS PUB 180-2, National Institute of Standards and Technology (NIST), Computer Systems Laboratory. Aug.

Ousterhout, J. K. 1994. *Tcl and Tk Toolkit*. Addison-Wesley.

Papatriantafilou, M. and Tsigas, P. 1998. Towards a library of distributed algorithms and animations. In *4th International Conference on Computer Aided Learning and Instruction in Science and Engineering (CALISCE '98)*. Gothenborg, Sweden, 407–410.

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

Pittel, B. 1987. On spreading a rumor. *SIAM Journal on Applied Mathematics 47*, 1 (Feb.), 213–223.

Plaxton, C. G., Rajaraman, R., and Richa, A. W. 1997. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures SPAA'97*. ACM Press, 311–320.

Raab, M. and Steger, A. 1998. "balls into bins" — A simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*. LNCS, vol. 1518. Springer-Verlag, 159–170.

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

Ratnasamy, S., Handley, M., Karp, R., and Shenker, S. 2001. Application-level multicast using content-addressable networks. In *Proceedings of the Third International COST264 Workshop*. LNCS, vol. 2233. Springer-Verlag, 14–29.

Raynal, M., Schiper, A., and Toueg, S. 1991. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters 39*, 6 (Sept.), 343–350.

Rodrigues, L., Baldoni, R., Anceaume, E., and Raynal, M. 2000. Deadline-constrained causal order. In *Proceedings of the Third IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2000)*.

Rowstron, A. and Druschel, P. 2001. 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)*. LNCS, vol. 2218. Springer-Verlag.

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

Schneider, M. 1993. Self-stabilization. In *ACM Computing Surveys*. 45–67.

Schreiner, W. 2002. A Java toolkit for teaching distributed algorithms. In *Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'2002)*. ACM press, 111–115.

Schulmeister, R. 2003. Taxonomy of multimedia component interactivity. A contribution to the current metadata debate. *Studies in Communication Sciences – Studi di scienze della communicazione Special issue*, 61–80.

Singhal, S. and Zyda, M. 1999. *Networked Virtual Environments.* Addison Wesley.

Slavin, R. E. 1995. *Cooperative Learning: Theory, Research and Practice*. Allyn and Bacon.

Spirakis, P., Tampakas, B., Papatriantafillou, M., Konstantoulis, K., Vlaxodimitropoulos, K., Antonopoulos, V., Kazazis, P., Metallidou, T., and Spartiotis, S. 1992. Distributed system simulator (DSS). In *Proceedings of Symposion on Theoretical Aspects of Computer Science (STACS '92)*. LNCS, vol. 577. Springer-Verlag, 615–616.

Stasko, J. 1995. POLKA animation designer's package. Tech. rep., Georgia Institute of Technology.

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

Tanenbaum, A. S. 1995. *Distributed Operating Systems*. Prentice-Hall, Inc., Chapter 6, 289–375.

Tel, G. 1994. *Introduction to Distibuted Algorithms.* Cambridge Press.

ViSiDiA 2000. Visidia project: Visualization and simulation of distributed algorithms (2000). *http://www.labri.fr/Recherche/LLA/visidia/*.

Wiley, D. A. 2000. Learning object design and sequencing theory. Ph.D. thesis, Brigham Young University.

Williams, S. and Roberts, T. S. 2002. Computer supported collaborative learning: Strengths and weaknesses. In *Proceedings of the International Conference on Computers in Education (ICCE'02)*. 328–331.

Wittmann, R. and Zitterbart, M. 1999. *Multicast Communication*. Morgan Kaufmann Publishers.

Xiao, Z., van Renesse, R., and Birman, K. P. 2002. Optimizing buffer management for reliable multicast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002)*. Washington, DC, 187–196.

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

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