Distrib. Comput. (2006) 18(5): 359–374
DOI 10.1007/s00446-005-0129-4

André Schiper

Dynamic group communication

Received: 23 August 2002 / Revised: 16 January 2005 / Accepted: 25 February 2005 / Published online: 31 December 2005

Abstract Group communication is the basic infrastructure for implementing fault-tolerant replicated servers. While group communication is well understood in the context of static groups (in which the membership does not change), current specifications of dynamic group communication (in which processes can join and leave groups during the computation) have not yet reached the same level of maturity.

The paper proposes new specifications – in the primary partition model – for dynamic reliable broadcast (simply called "reliable multicast"), dynamic atomic broadcast (simply called "atomic multicast") and group membership. In the special case of a static system, the new specifications are identical to the well known static specifications. Interestingly, not only are these new specifications "syntactically" close to the static specifications, but they are also "semantically" close to the dynamic specifications proposed in the literature. We believe that this should contribute to clarify a topic that has always been difficult to understand by outsiders.

Finally, the paper shows how to solve atomic multicast, group membership and reliable broadcast. The solution of atomic multicast is close to the (static) atomic broadcast solution based on reduction to consensus. Group membership is solved using atomic multicast. Reliable multicast can be efficiently solved by relying on a thrifty generic multicast algorithm.

Keywords Group communication · Dynamic group · Specification · Reliable broadcast · Atomic broadcast · Group membership

1 Introduction

Fault-tolerance in distributed systems is ensured by replication, which is traditionally implemented on top of a group communication infrastructure. Reliable broadcast – which ensures that all correct processes or none of them deliver a given message – and atomic broadcast – which in addition to the properties of reliable broadcast orders messages – are examples of group communication primitives. These primitives provide the right level of abstraction for implementing replicated (fault-tolerant) services. For example, atomic broadcast is the adequate group communication primitives for active replication (also called *state machine approach* [25]). Group communication also addresses the need of passive replication, as shown in [13, 19].

Group communication is well understood in the context of a static groups, in which the membership of a group does not change. The specification of static group communication primitives can be found in [14], and an implementation of reliable broadcast and atomic broadcast is discussed for example in [4]. However, a static group has practical limitations. Consider for example a replicated server with three replicas s_1, s_2, s_3. If s_3 crashes, the probability for the service to be permanently available decreases. To increase this probability a new replica s_4 must be created to replace s_3. This requires dynamic group communication. However, despite the recent good survey by Chockler et al. [6], dynamic group communication has not yet reached the level of maturity of static group communication. The goal of this paper is to contribute to a better understanding of dynamic group communication. Note that the paper only considers processes that fail by crashing (no Byzantine processes).

The key component of a dynamic group is the *group membership* service, which is responsible for adding and removing processes during the computation [7, 22]. In this paper we consider only the so called *primary partition* membership problem. Group membership is strongly related to *view synchrony*, a property ensuring that processes deliver the same set of messages between two membership changes [3, 23]. However, despite the fact that view synchrony (with dynamic groups) and reliable broadcast (with static groups) are related, their specifications are quite different. The same holds for the specification of atomic broadcast in a static *vs.* in dynamic setting (atomic broadcast is usually

A. Schiper (✉)
Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
E-mail: Andre.Schiper@epfl.ch

specified as an extension of reliable broadcast, respt. view synchrony).The paper shows that the gap between static and dynamic specifications is not ineluctable, by proposing new specifications for dynamic group communication that map to the standard static specifications in the special case when the group membership does not change. The paper also proposes a specification of the group membership problem that differs from the existing specifications. Contrary to existing specifications, *all* membership changes, including the exclusion of a process that is suspected to have crashed, result from *explicit* invocations of membership change primitives. This approach allows us to simply specify the group membership problem by properties derived from those of dynamic atomic broadcast.

The interesting feature of the new specifications is not only that they are *syntactically* close to the standard static specifications, but also that they are *semantically* close to the dynamic specifications proposed in the literature [6]. In our opinion, this is the most important contribution of the paper. However, new specifications without implementation is not satisfactory: the question of *solvability* must be addressed. The paper shows that traditional *static* solutions can be extended to solve the dynamic problems: dynamic atomic broadcast can be solved by reduction to consensus, similarly to (static) atomic broadcast [4]. We show that group membership can be solved using dynamic atomic broadcast. The solution of dynamic atomic broadcast trivially solves dynamic reliable broadcast. However, the solution is inefficient. We discuss a more efficient solution, based on the use of generic broadcast [1, 20, 21].

The rest of the paper is structured as follows. Section 2 is devoted to the specifications of dynamic reliable broadcast, dynamic atomic broadcast and group membership. Section 3 compares our new specifications with those in the recent survey by Chockler et al. [6]. Section 4 solves dynamic atomic broadcast and group membership, and proves the correctness of the solution. Section 5 extends the solution to dynamic reliable broadcast. Section 6 concludes the paper.

2 Specification of dynamic group communication

In this section we define dynamic reliable broadcast, dynamic atomic broadcast and group membership. In order to shorten names, we call *dynamic reliable broadcast* simply *atomic "multicast"*, and *dynamic atomic broadcast* simply *reliable "multicast"*. We adopt the same presentation order as in [14]: we define reliable multicast first, and then extend it to atomic multicast. We start with a preliminary discussion and some definitions.

2.1 Preliminary discussion

Contrary to the specification of static group communication concerned only with *communication primitives*, the specification of dynamic group communication has two parts: a *communication primitives* part, and another part dedicated to group *membership changes*. There are two options here: one is to specify communication primitives first, the other is to specify membership changes first.

Quite universally, e.g., [6], the membership part is specified first. This choice can be explained by existing implementations, in which the *group membership* layer is the basic layer of group communication stacks. Indeed, if the group membership layer is beneath the group communication layers in protocol stacks, it is quite natural to specify the group membership problem first (and the communication primitives in a second step). The paper takes a slightly different approach motivated by the following arguments:

- *Justification of membership changes:* Consider *atomic broadcast*, defined by *abroadcast* and *adeliver*. If some process executes *adeliver(m)*, it seems obvious to require that some process has executed previously *abroadcast(m)*: *adeliver(m)* can be explained (or justified) by *abroadcast(m)*. This seems quite natural in the context of atomic broadcast. However, for historical reasons, the situation is different in the context of the specification of group membership, where most existing specifications do not justify *all* membership changes, i.e., inclusions *and* exclusions.[1] Our opinion is that group membership specifications become more "natural" if *all* group membership changes (inclusions *and* exclusions) are justified. This can be done by introducing primitives to add and remove processes to/from a group. *Note that the* reason *for adding or removing a process is not part of the specification.* This is similar to atomic broadcast, where the specification of atomic broadcast does not explain why some process executes atomic broadcast. The decision to issue atomic broadcast is left to the application that uses the group communication infrastructure. In the same way, the decision to add or remove a process to/from a group is left to the application. We advocate this separation of concern (see [24] for a more extensive discussion).
- *Simplicity:* One of the properties usually required from a primary partition membership service is that, for all i, processes agree on the ith membership of the group. This agreement property, together with the requirement that membership changes are justified by invocations to add or remove a process, leads to a problem that has a strong flavor of atomic multicast. So, a simple solution is to derive the specification of the group membership problem from the specification of atomic multicast.

These two arguments lead us to specify atomic broadcast first, and group membership in a second step. However, the specification of group membership that we give below is independent of the specification of atomic broadcast. So, our presentation order does not enforce a dependency order between atomic multicast and group membership: *nothing prevents, once these two specifications are understood, to*

[1] Some specifications, e.g., [16] and [11], justify view changes using oracles or internal events, i.e., events not under the control of the application.

switch them. However, this is not done here, as it would not add any benefit.[2]

2.2 Definitions

We consider a distributed system composed of processes taken from a finite set Π: Π is the set of all possible processes (the universe of processes). A *group g* consists of a subset of Π. The processes in g are said to be the *members* of the group. The membership of g can change over time.

2.2.1 View

We use the standard notion of *view* to refer to the successive membership of a group. As commonly done, we define a view to be a tuple $v = (i, S)$, where i is an integer, and S is a non-empty subset of Π: the integer i is the identifier of view v, and S is the membership of v. A priori, i.e., before the specification of group membership, we do not put any constraints on views. So, we need a notation to distinguish the views perceived by different processes: we denote by $v^p = (i^p, S^p)$ the view of p with identifier i^p. Given two processes p and q, their views $v^p = (i^p, S^p)$ and $v^q = (i^q, S^q)$ are equal if $i^p = i^q$ and $S^p = S^q$. Moreover, we introduce the following terminology:

- *Process p is in view v* (or simply *p is in v*): Given view $v = (i, S)$, we say that p is in view v if $p \in S$.
- *View installation:* The event by which a process changes its view is called view installation.
- *The view of process p is v*: We say that the view of process p is v, after p has installed view v and before p installs another view.
- *Event e occurs in view v:* We say that event e occurs on process p in view v if event e occurs while the view of p is v.
- *View v is the last view of p:* We say that view v is the last view of p if p does not install any view after v.

2.2.2 g-correct vs. g-faulty process

Specifications of static group communication distinguish *correct* processes (processes that never crash) and *faulty* processes (processes that crash): obligations in the specifications (i.e., the obligation to delivery messages) are put on correct processes. This is comes from the fact that static specifications implicitly assume one single group and all processes are members of this group. So a correct process is always a member of this single group.

This is no more the case with dynamic groups, even if we consider one single dynamic group g: we can have correct processes not member of g. A correct process p not in g has no obligation with respect to messages broadcast to g. So, dynamic group communication cannot be specified by putting requirements on correct processes. Instead, the requirements must be put on processes that are member of g but do not crash. We formalize this in the context of some group g, by introducing the notions of *g-correct* process and *g-faulty* process. The definition of g-correct and g-faulty is in two steps. Given a view v of g, we define first the notions *v-correct* process and *v-faulty* process.

- *v-correct process:* Consider some view v with process p in v. We say that process p is *v-correct* if:
  (i) p installs view v, and
  (ii) p does not crash while its view is v, and
  (iii) if v is not the last view of some process in v, then $\exists$ view v' installed immediately after v by some process in v such that p is in v'.

A process that is not v-correct is *v-faulty*. Although this definition does not subsumes the primary partition model (in which processes agree on the sequence of views), the definition can easily be understood in the context of the primary partition model. Consider that processes p, q, r all install view $v = (0, \{p, q, r\})$, and that process p installs view $v' = (1, \{p, q\})$. According to our definition, process p is v-correct. Moreover, since q is in v', process q is v-correct unless it crashes in v. What about process r? If we assume the primary partition model, the only view that q and r can install after v is also v'. Since r is not in v', r is v-faulty. *So, to be v-correct, a process must not crash in v and be in the next view.*

The notion of v-correct process allows us to define the notion of g-correct process. Intuitively, process p is g-correct if p is v-correct in all its views. Formally:

- *g-correct process:* Consider a group g, process p with v^p_{init} the initial view of p for g, and p in v^p_{init}. We say that p is g-correct if (i) p is v^p_{init}-correct, and (ii) there exists no view v' such that (a) p is in v' and (b) p is v'-faulty.

A process that is not g-correct is *g-faulty*. As for the definition of v-correct, this definition does not assume a primary partition membership model. In the primary partition membership model, according to this definition, a process p is g-correct if p, after having installed its initial view v^p_{init}, (i) is never removed from the group, (ii) installs all the subsequent views that are defined, and (iii) never crashes. A process that crashes or is removed from the group is g-faulty. Note that this does not mean that a g-faulty process has crashed.

2.2.3 Static group

A static group is a special case of our dynamic model with one single group g, and $v_0 = (0, S_0)$, $S_0 \subseteq \Pi$, the first and the last view of group g. If we apply the above definition to this static case, we have that p is g-correct if and only if p is correct. So, in the special case of static groups, our definition of a g-correct process is the usual definition of a

[2] Note that [18] also specifies atomic broadcast before group membership (and uses atomic broadcast to implement group membership). However, contrary to this paper, the specification of group membership does not require process exclusions to be justified.

correct process. Moreover, p is v_0-correct if and only if p is correct. These equivalences will allow us to compare our dynamic group communication specifications with the usual static specifications.

2.3 List of primitives

In order to make the specification sections easier to read, here is the full list of primitives considered below:

– *rmulticast* and *rdeliver*, used to define reliable multicast;
– *amulticast* and *adeliver*, used to define atomic multicast;
– *join-inv* and *join-exec*, used to define group membership (to add a process);
– *leave-inv* and *leave-exec*, used to define group membership (to remove a process);
– *init*, used in the context of group membership (to initialize the view of a process).

2.4 Reliable multicast: first attempt

The definition of (static) reliable broadcast assumes the static group model. Our definition of *reliable multicast* considers the dynamic model with views and some implicit group g. Reliable multicast is defined by the two primitives *rmulticast* and *rdeliver*, and – as a first attempt – the following three properties:[3]

R1 *Validity:* If a g-correct process executes *rmulticast*(m), then it eventually *rdelivers* m.
R2 *Uniform Agreement:* If a process p *rdelivers* m in view v^p, then all processes that are v^p-correct eventually *rdeliver* m.
R3 *Uniform Integrity:* For any message m, (i) every process *rdelivers* m at most once, and (ii) only if m was previously *rmulticast* by $sender(m)$.[4]

This definition has a problem. It allows runs in which the specification is satisfied with respect to some process p, and violated with respect to some other process q. Consider the following example in which no process crashes. View $v_0 = (0, \{p, q\})$ is the initial view of the group and

– process p executes *rmulticast*(m) and later *rdelivers* m in view v_0,
– process q installs view $v_1 = (1, \{p, q, r\})$ and *rdelivers* m in view $v_1 = (1, \{p, q, r\})$,
– process r installs view $v_1 = (1, \{p, q, r\})$ and never *rdelivers* m. No other view is defined and r does not crash. Since r does not crash, r is v_1-correct.

In this example, the above specification is satisfied for p (q has *rdelivered* m), but not for q (r is v_1-correct, q has *rdelivered* m in view v_1, but r never *rdelivers* m).

2.5 Reliable multicast: second attempt

To avoid the above problem, we add a property that requires message m to be delivered by all processes in the same view:

R4 *Uniform Same View Delivery:* If two processes p and q *rdeliver* m in view v^p (for p) and v^q (for q), then $v^p = v^q$.

Altogether, we define reliable multicast by the following properties: Validity, Uniform Agreement, Uniform Integrity and Same View Delivery.

It is easy to see that our definition of reliable multicast is a generalization of (static) reliable broadcast. If the group is static, i.e., if the view never changes, then the Same View Delivery property is trivially satisfied. Moreover, since v_0-correct and g-correct reduce to "correct", the other properties are identical to those that define reliable broadcast [14]. *So, if the group is static, reliable multicast is equivalent to* (*static*) *reliable broadcast.*

2.6 Atomic multicast

We define atomic multicast by the two primitives *amulticast* and *adeliver*, and – similarly to [14] for static groups – by the properties R1–R4 that define reliable multicast,[5] plus an additional ordering property. For this last property, we take the definition in [1] – which contrary to the order property in [14] forbids "holes" in the delivery sequence of messages. However, we must adapt the definition in [1] since dynamic joining of processes poses a specific problem: we do not want process p to have to deliver messages delivered before p has joined the group. We express this using views:

A5 *Uniform Total Order:* If some process (whether g-correct or g-faulty) *adelivers* message m in view v before it *adelivers* message m', then every process p in v, *adelivers* m' only after it has *adelivered* m.

To illustrate A5, consider some process q that has *adelivered* m in view v, and later m' in view v' (possibly $v = v'$). If p has *adelivered* m', and p is in v, then p has joined the group before the delivery of m, and consequently p must *adeliver* m. If p is not in v and has *adelivered* m', then p has joined the group after the delivery of m, and does not have to *adeliver* m.

As for reliable multicast, if the group is static, atomic multicast is equivalent to (*static*) *atomic broadcast.*

[3] All the broadcast primitives that we define in this section are *uniform* [14]. To simplify the notation, we drop the word "uniform" from the various broadcast types. Non uniform primitives are discussed in Sect. 2.9.

[4] As usual, since every process can multicast several messages, we assume that every message has an identifier field that makes every message that is multicast unique.

[5] In these properties *rmulticast* must be replaced with *amulticast* and *rdeliver* with *adeliver*.

2.7 Group membership

2.7.1 *Specification based on join and leave requests*

To complete the definition of dynamic group communication, we need to specify how views change, i.e., we need to specify the group membership problem. We consider the most basic specification, with only two operations: one to add a process to the group, and one to remove a process from the group.We call these operations *join*, respectively *leave*.

The join and leave operations are the only means to modify the membership. So, events such as process crashes, failure suspicions or similar events do not appear in our specification. This allows a clear separation of concerns between the question of *why* a process is excluded and the question of *how* it is excluded. As explained in Sect. 2.1, our specification addresses the second issue.

Process p requests to add process r to the group by *invoking* the operation $join(r)$. However, the view only changes when the $join(r)$ operation is scheduled for execution. In other words, process r is included in the view of some process q (possibly $q = p$) when q *executes* $join(r)$. Similarly, process p requests to remove r from the group by invoking the operation $leave(r)$, and the view changes once the operation is scheduled for execution: process r is removed from the view of process q when q executes $leave(r)$.

2.7.2 *Notation*

We introduce the following notation. The "invocation" of the operation *join* (respt. *leave*) is denoted by *join-inv* (respt. *leave-inv*). The "execution" of the operation *join* (respt. *leave*) is denoted by *join-exec* (respt. *leave-exec*). Upon occurrence of $join\text{-}exec(x)$ at process p in view $v = (i, S)$, the view of p atomically becomes $(i+1, S \cup \{x\})$. So, $join\text{-}exec(x)$ is the last event in view v for p. Upon occurrence of $leave\text{-}exec(x)$ at process p in view $v = (i, S)$, the view of p atomically becomes $(i+1, S - \{x\})$: $leave\text{-}exec(x)$ is the last event in view v for p. The notation $join\text{-}exec()^v$ (respt. $leave\text{-}exec()^v$) is used to denote that the execution of join (respt. leave) leads to the view v. When process p executes $join\text{-}exec()^v$ or $leave\text{-}exec()^v$, we say that p *installs* view v (see Sect. 2.2).

2.7.3 *Specification*

We want *join-exec* and *leave-exec* to be executed in the same total order by all processes. This can be easily specified by relying on the properties of atomic multicast:

- we map $join\text{-}inv(x)$, $x \in \Pi$, to $amulticast(add(x))$, and $adeliver(add(x))$ to $join\text{-}exec(x)$.
- we map $leave\text{-}inv(x)$, $x \in \Pi$, to $amulticast(remove(x))$, and $adeliver(remove(x))$ to $leave\text{-}exec(x)$.

With this mapping, we specify the group membership problem for some implicit group g by the properties R1–R4, A5 of atomic multicast, plus an additional initialization property G0. Property G0 defines the initial view of process p to be either the initial view v_0 of the group, or a view v installed by some other process q:

G0 *GM Initialization:* The initial view of the group is $v_0 = (0, S_0)$, $S_0 \subseteq \Pi$. For every process p, its initial view v, defined by the execution of the primitive $init^v$, is such that (i) p is in view v, and (ii) either $v = v_0$ or there exists a process q different from p that installs view v.

GR1 *GM Validity:* If a g-correct process executes $join\text{-}inv(x)$ (respt. $leave\text{-}inv(x)$), then it eventually executes $join\text{-}exec(x)$ (respt. $leave\text{-}exec(x)$).

GR2 *GM Uniform Agreement:* If a process p executes $join\text{-}exec(x)$ (respt. $leave\text{-}exec(x)$) in view v^p, then all processes that are v^p-correct eventually execute $join\text{-}exec(x)$ (respt. $leave\text{-}exec(x)$).

GR3 *GM Uniform Integrity:* For any process p, every process q executes $join\text{-}exec(p)$ (respt. $leave\text{-}exec(p)$) at most once, and only if $join\text{-}inv(p)$ (respt. $leave\text{-}inv(p)$) was previously invoked.

GR4 *GM Uniform Same View Delivery:* If two processes p and q execute $join\text{-}exec(x)$ (respt. $leave\text{-}exec(x)$) in view v^p (for p) and v^q (for q), then $v^p = v^q$.

GA5 *GM Uniform Total Order:* Let $op(x)$ denote either $join\text{-}exec(x)$ or $leave\text{-}exec(x)$. If some process (whether g-correct or g-faulty) executes $op(x)$ in view v before it executes $op(y)$, then every process p in v, executes $op(y)$ only after it has executed $op(x)$.

Remark 1 GR3 prevents $join\text{-}exec(x)$, respt. $leave\text{-}exec(x)$, to be executed more than once. This prevents fictitious view changes (view changes where the old and the new view have the same membership). Consider for example view $v = (i, \{p, q, r\})$ and processes p, q both invoking $leave\text{-}inv(r)$ in view v. GR3 requires $leave\text{-}exec(r)$ to be executed only once. This prevents to have a first view change from $(i, \{p, q, r\})$ to $(i + 1, \{p, q\})$, followed by a second view change from $(i + 1, \{p, q\})$ to $(i + 2, \{p, q\})$.

Remark 2 The ordering property GA5 is actually redundant. It is easy to show that GA5 follows from GR4. The proof is by contradiction. Assume that process q executes $op(x)$ in view v before executing $op(y)$ in view v', and process p in v executes $op(y)$ before executing $op(x)$. Let i be the identifier of view v, and i' the identifier of view v'. Since q executes $op(x)$ before $op(y)$, we have $i < i'$ (*). By the Uniform Same View Delivery property GR4, p executes $op(x)$ in view v and $op(y)$ in view v'. Since p executes $op(x)$ before $op(y)$, we have $i' < i$: a contradiction with (*).

Remark 3 We define $init^{(i,S)}$ followed immediately by $join\text{-}exec(x)$ (i.e., no message broadcast or delivered in-between) to be equivalent to $init^{(i+1, S\cup\{x\})}$. This property is exploited by the implementation in Sect. 4.

2.8 Examples

We illustrate now our specifications on a few examples (in the context of one implicit group g).

Example 1 In the first example, the initial view is $v_0 = (0, \{p, q\})$, no process crashes, and all processes are g-correct. Message m is *amulticast* in view v_0 by process p and *adelivered* in view $(1, \{p, q, r\})$:

– local history of p:
$\mathit{init}^{(0,\{p,q\})}$; $\mathit{amulticast}(m)$; $\mathit{join\text{-}exec}(r)^{(1,\{p,q,r\})}$; $\mathit{adeliver}(m)$
– local history of q:
$\mathit{init}^{(0,\{p,q\})}$; $\mathit{join\text{-}inv}(r)$; $\mathit{join\text{-}exec}(r)^{(1,\{p,q,r\})}$; $\mathit{adeliver}(m)$
– local history of r:
$\mathit{init}^{(1,\{p,q,r\})}$; $\mathit{adeliver}(m)$

In this example, process q executes $\mathit{join\text{-}inv}(r)$ to add process r. Process q may have done so based on its own initiative, in order to increase the size of the group. Or process q may have done so because it was contacted by r that wanted to join the group. The exact reason is outside the scope of our specifications. Moreover, if p was contacted by r, the life of r *before* being member of the group is outside of our model.

Example 2 The second example shows a process r that is g-faulty because of the execution of $\mathit{leave\text{-}exec}(r)$: process r is v_0-faulty with $v_0 = (0, \{p, q, r\})$ since it is not member of the next view $(1, \{p, q\})$. Being v_0-faulty, process r has no obligation to deliver m. Processes p and q are g-correct:

–local history of p:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{amulticast}(m)$; $\mathit{adeliver}(m)$; $\mathit{leave\text{-}exec}(r)^{(1,\{p,q\})}$
–local history of q:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{leave\text{-}inv}(r)$; $\mathit{adeliver}(m)$; $\mathit{leave\text{-}exec}(r)^{(1,\{p,q\})}$
– local history of r:
$\mathit{init}^{(0,\{p,q,r\})}$

Example 3 The following history, slightly changed with respect to Example 2, is also allowed by the specification. Our specifications do not restrict the behavior of g-faulty processes. The g-faulty process r may deliver m and install view $(1, \{p, q\})$:

– local history of p:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{amulticast}(m)$; $\mathit{adeliver}(m)$; $\mathit{leave\text{-}exec}(r)^{(1,\{p,q\})}$
– local history of q:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{leave\text{-}inv}(r)$; $\mathit{adeliver}(m)$; $\mathit{leave\text{-}exec}(r)^{(1,\{p,q\})}$
– local history of r:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{adeliver}(m)$; $\mathit{leave\text{-}exec}(r)^{(1,\{p,q\})}$

Example 4 This example shows a process r that crashes. Our specification does not require r to be removed for the group. The following history is allowed by the specification:

– local history of p: $\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{amulticast}(m)$;$\mathit{adeliver}(m)$
– local history of q: $\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{adeliver}(m)$
– local history of r: $\mathit{init}^{(0,\{p,q,r\})}$; *crash*

Example 5 Consider again the crash of process r in Example 4. Even though our specification does not require r to be removed from the group, the application (i.e., the software that issues *amulticast*, *join-inv* and *leave-inv*) may suspect the crash of r (using whatever mechanism) and ask to remove r. This could lead to the following history:[6]

– local history of p:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{amcast}(m)$; $\mathit{adlvr}(m)$; $\mathit{lv\text{-}inv}(r)$; $\mathit{lv\text{-}exec}(r)^{(1,\{p,q\})}$
– local history of q:
$\mathit{init}^{(0,\{p,q,r\})}$; $\mathit{adlvr}(m)$; $\mathit{lv\text{-}exec}(r)^{(1,\{p,q\})}$
– local history of r:
$\mathit{init}^{(0,\{p,q,r\})}$; *crash*

2.9 Uniform vs. non uniform specifications

As mentioned in Sect. 3, the specifications given for reliable multicast, atomic multicast are uniform. Some of the *uniform* properties could be weaken to *non uniform* properties, by simply requiring the properties to hold only for *g-correct* processes:

R2a *Agreement:* If a g-correct process p *rdelivers* m in view v^p, then all processes that are v^p-correct eventually *rdeliver* m.
R3a *Integrity:* For any message m, (i) every g-correct process *rdelivers* m at most once, and (ii) only if m was previously *rmulticast* by *sender*(m).
R4a *Same View Delivery:* If two g-correct processes p and q *rdeliver* m in view v^p (for p) and v^q (for q), then $v^p = v^q$.
A5a *Total Order:* If some g-correct process *adelivers* message m in view v before it *adelivers* message m', then every process p in v, *adelivers* m' only after it has *adelivered* m.

Are non uniform properties desirable, and what are their advantages over uniform specifications? Non uniform specifications have an advantage only if they allow for less costly implementation. This is however not always the case. Défago et al. [8] show the following result for (static) atomic broadcast: any algorithms that solves non uniform atomic broadcast with the failure detectors $\diamond\mathcal{P}$ (respt. $\mathcal{S}$, $\diamond\mathcal{S}$) [4], solves also the uniform version of atomic broadcast with $\diamond\mathcal{P}$ (respt. $\mathcal{S}$, $\diamond\mathcal{S}$).[7] So, in this context, non uniform specifications do not provide any advantage.

However, existing group communication system do provide non uniform communication primitives that are less costly than the uniform ones. This is possible because these systems rely on a different system model, called *process controlled crash*. This model gives the ability to processes to kill other processes (see [8]). However, in this context, non uniform primitives can have drawbacks that balance their efficiency. They can lead a faulty process, before it crashes, to

[6] To have the history fit on one line, the name of the primitives have been shortened, e.g., *amulticast* → *amcast*, *adeliver* → *adlvr*.
[7] The result was initially established for consensus in [12].

disseminate information in the system based on an inconsistent state. As a result, the whole system state can become inconsistent.

Finally, even though our model allows us to force a process to become g-faulty (by removing the process from the group), the model that we use in Sect. 4 to implement our specifications, together with the failure detectors $\diamond\mathcal{P}$, $\mathcal{S}$ or $\diamond\mathcal{S}$, does not allow us to exploit the non uniform properties. The argument is basically the same as the one given in [8] for (static) atomic broadcast. Interestingly, this shows that contrary to a common belief, it is not the ability to force processes to become faulty (or g-faulty) that allows the exploitation of non uniform specifications. The exploitation of non uniform specifications is only possible if the atomic broadcast algorithm has the right to force processes to become faulty (or g-faulty). We do not allow this. *In our specification, removing processes from the group is not under the control of the atomic broadcast algorithm.*

3 Comparison with current specifications

We compare now the above specifications with those of Chockler et al. [6]. The paper surveys over thirty published group communication specifications. As stated by the authors, the goal is *to serve as a unifying framework for the comparison of group communication systems*. So, the comparison of our specifications with those of Chockler et al. indirectly allows the comparison of our specifications with those surveyed in [6]. Note however that the specifications in [6] have a broader scope: they address group communication in the primary partition model and in the partitionable model. The comparison below is with the subset of [6] that is relevant to the primary partition model.

We discuss the properties in the same order as in [6]: first safety and then liveness properties. The safety properties in [6] start with the properties of group membership.

3.1 Group membership safety properties

In [6], the safety properties of group membership are split into (1) *basic* and (2) *primary* vs. *partitionable* properties. There are three basic properties:[8]

– *Property 3.1 (Self Inclusion): If process p installs view v, then p is a member of v.*

The property is uniform (it must hold also for faulty – or g-faulty – processes). Our specification, which is non uniform with respect to self inclusion, is weaker. We explain why.

By our Initialization property G0, the initial view of a process satisfies the Self Inclusion property. Self Inclusion is also satisfied by all subsequent views installed by some process p, until p is removed from the group. Our specification does not prevent a g-faulty process p (i.e., a process removed from the group) to install a view that does not include it. This can be useful: it allows the process that is removed to know when this removal exactly takes place.

– *Property 3.2 (Local Monotonicity): If a process p installs view v after installing view v', then the identifier of v' is greater than that of v.*

In our model views also have identifiers, and view identifiers are incremented by one for each new view (see Sect. 2.7.2), i.e., Property 3.2 holds. However, strictly speaking, this is not part of our specification.

– *Property 3.3 (Initial View Event): Every $send$, $recv$ and $safe\_prefix$ event occurs within some view.*

This property is not ensured by our specification, which allows a process to execute events without being member of a group (e.g., before joining a group).

There is only one non-basic safety property in [6]:

– *Property 3.4 (Primary Component Membership): There is a one to one function f from the set of views installed in the trace to the natural numbers, such that f satisfies the following property: for every view v with $f(v) > 1$ there exists a view v', such that $f(v) = f(v') + 1$, and a member p of v that installs v in v' (i.e., v is the successor of v' at process p).* This property implies that for every pair of consecutive views, there is a process that survives from the first view to the second [6].

Our specification satisfies this property. Consider the first part of Property 3.4, and let f be defined recursively as follows: (i) $f(v_0) = 0$, (ii) if some process installs view v' immediately after view v, then $f(v') = f(v) + 1$. By the Uniform Total Order property GA5, f is indeed a function. We show by contradiction that the second part of Property 3.4 also holds. Assume for contradiction that there exists views v and v' such that $f(v) = f(v') + 1$ and that no process in v' installs v. By definition, v is not the initial view v_0. By the Initialization property G0, there exists a process q – different from p – that installs v. Since $f(v) = f(v')+1$, q installs v in view v'.

To summarize, our specification of group membership implies the relevant safety properties of [6]. The opposite is not true. For example, the Uniform Integrity property GR3 does not hold in [6], where view changes are not required to be "justified".

Note that [6] defines two other group membership safety properties: Property 4.6 (Transitional Set) and Property 4.7 (Agreement on Successor). However, since these two properties are related to the partitionable membership model, we do not discuss them here.

[8] We give only the informal specification of [6]. For additional information, please refer to [6].

3.2 Multicast safety properties

In [6], the multicast safety properties are split into (1) *basic*, (2) *sending view delivery and weaker alternatives*, and (3) *virtual synchrony* properties. There are two basic properties:[9]

- *Property 4.1 (Delivery Integrity): For every $recv$ event there is a preceding $send$ event of the same message.*

This property corresponds to part (ii) of our Uniform Integrity property R3.

- *Property 4.2 (No Duplication): Two different $recv$ events with the same content cannot occur at the same process.*

This corresponds to part (i) of our Uniform Integrity property R3.

There are two properties in the category *sending view delivery and weaker alternatives*:

- *Property 4.3 (Sending View Delivery): If a process receives message m in view v, and some process q (possibly $p = q$) sends m in view v', then $v = v'$.*

Our specification does not require Sending View Delivery. This property could be added, but as noticed in [6], Same View Delivery (see below) is the *basic* property (rather than Sending View Delivery).

- *Property 4.4 (Same View Delivery): If processes p and q both receive message m, they receive m in the same view.*

This property is ensured by our Uniform Same View Delivery property R4.

Three properties are given in the category *virtual synchrony*, but only one is relevant to the primary partition model:

- *Property 4.5 (Virtual Synchrony): If processes p and q install the same view v in the same previous view v', then any message received by p in v' is also received by q in v'.*

In [6], by the Self Inclusion property, if p and q install views v and v', then p and q are in v and v'. Our specification does not prevent a g-faulty process to install a view to which it does not belong. If we assume that p and q are in v and v', then our specification satisfies the Virtual Synchrony property.

Consider processes p and q that install view v in view v', and message m delivered (*rdelivered* or *adelivered*) by p in view v'. If q is in v and v', and q installs view v, according to our definition, q is v'-correct. Since p has delivered m in view v', by the Uniform Agreement property R2 and the Uniform Same View Delivery property R4, q also delivers m in view v'.

[9] As in Sect. 3.1, we give only the informal specification.

To summarize, our specifications imply the properties of [6], except for "sending view delivery". The opposite is also true: the specifications in [6] imply our safety properties R3 and R4.

3.3 Ordering and reliability properties

The ordering and reliability properties in [6] are split into *FIFO multicast*, *causal multicast*, and *total order multicast*. Since we do not consider FIFO order and causal order, we discuss only the total order multicast category:

Property 6.5 (Strong Total Order): There is a timestamp function f such that messages are received at all processes in an order consistent with f.

Property 6.7 (Reliable Total Order): There exists a timestamp function f such that if a process p receives a message m', and messages m and m' were sent in the same view, and $f(m) < f(m')$, then q receives m before m'.

Reliable Total Order requires processes to deliver messages in the same order if they were sent in the same view. Note that Strong Total Order and Reliable Total Order are incomparable. Our specification, by the Uniform Total Order property A5, implies the Strong Total Order property: messages are delivered in the same order even if they were sent in different views.

3.4 Liveness properties

The liveness properties of [6] related to group membership are difficult to compare to our specifications. This is because events that trigger view changes do not appear in the specification of [6]. For example, our Validity property GR1 and our Uniform Agreement property GR2 do not hold in [6]. Even the weaker non-uniform version of GR2 does not hold.

Concerning message multicast, we can make the following observations. Whereas [6] requires the properties to hold only in runs in which there exists a stable component and the failure detector behaves like $\diamond\mathcal{P}$, our properties do not require such a condition. This makes it difficult to formally compare the two specifications.

3.5 Discussion

The comparison has shown strong similarities between our new specifications and those in [6] that are relevant in the primary partition model. The main difference is with respect to liveness properties, which in [6] are required to hold only if the system stabilizes. Our liveness properties, which are expressed much like in [14], do not depend on such a condition.

4 Solving atomic multicast with membership changes

Giving new specifications without addressing the issue of solvability is not satisfactory. This is the purpose of this section, which shows how to solve the atomic multicast problem with membership changes. The solution also trivially solves the reliable multicast problem, however inefficiently. An improved (and more complex) solution for reliable multicast is discussed in Sect. 5.

Since atomic multicast and group membership require a strong form of agreement, and the FLP impossibility result [10] shows that agreement cannot be built on top of weak communication primitives in asynchronous systems, we solve static atomic multicast over a basic layer solving consensus. Moreover, since our specification of group membership is close to the specification of atomic multicast, we simply solve group membership using static atomic multicast. The solution is given by Algorithm 1.

Note that our solution is derived from existing solutions and ideas. The idea of implementing total order by reduction to consensus has been described in [4]. Algorithm 1 is derived from this solution (we explain below the differences). Using total order broadcast to solve the group membership problem was already proposed in [18]. However, the solution does not separate atomic broadcast/multicast from its use to change the membership as clearly as in our solution. The idea of using total order to solve the group membership problem appears also in [15], where it is suggested to consider the membership as part of the state managed by a state machine (i.e., a membership change appears as a state change). Finally, a related idea, which consists of solving the group membership directly by reduction to consensus, instead of indirectly as done here, was proposed in [17].

4.1 System model

The definitions in Sect. 2.2 are related to the specification of reliable/atomic multicast and group membership. The issue was not solvability, which is a different issue.[10] We give now the system model that we assume to address the solvability issue. With respect to communication, we assume reliable channels, defined by the primitives *send(m)* and *receive(m)*, which have the following properties: (i) if process q receives message m from p, then p has sent m to q (*no creation*), (ii) q receives m from p at most once (*no duplication*), and (iii) if p sends m to q, and q is correct, then q eventually receives m (*no loss*).

We also assume a *consensus-oracle* that solves consensus. The consensus-oracle is defined by *propose(k,S,val)* and *decide(k,decision)*. When process p executes *propose(k,S,val)*, the parameter k identifies a specific instance of consensus, S denotes the set of processes that have to reach agreement, and val is p's initial value. Given instance k_0 of consensus and a set S_{k_0}, the consensus oracle ensures the following property. If all processes p in S_{k_0} that do not crash execute $propose(k_0, S_{k_0}, val_p)$, then all those processes eventually decide (Termination), the *decision* is one of the initial values val_p (Validity), and no two processes in S_{k_0} decide differently (Uniform Agreement) [4]. Solving consensus is discussed for example in [4].

4.2 Algorithm 1: atomic multicast and membership changes

Algorithm 1 consists of four tasks (Task 1 to Task 4) that execute atomically, and one task (Task 5) consisting of *two* atomic blocks, the first corresponding to lines 21–24, the second to lines 25–37. Algorithm 1 is close to the static atomic broadcast algorithm in [4] that works as follows. Processes execute a sequence of consensus numbered 1, 2, ... The initial value and the decision of each consensus is a set of messages. Let $adeliver^k$ be the set of messages decided by consensus #k: (1) the messages in the set $adeliver^k$ are delivered before the messages in the set $adeliver^{k+1}$, and (2) the messages in the set $adeliver^k$ are delivered according to a deterministic function.

The main difference between [4] and our dynamic algorithm is that the sequence of consensus is no more executed by a constant set of processes. Instead, consensus #k is executed by the processes that are members of the group when consensus #k is started. The management of this dynamism requires additional changes. The first change is the introduction of message types, in order to distinguish the atomic multicast of ordinary messages (messages of type *am*, lines 12,13 of Algorithm 1), from the multicast of join requests (messages of type *add*, lines 14,15), respt. leave requests (messages of type *remove*, lines 16,17). The other changes are the following:

1. Initialization of the joining processes (line 36 and lines 7-8).
2. Line 37, by which p sends the messages received but not yet delivered (i.e., $received_p - adelivered_p$) to the joining processes.
3. The delivery order, by which messages of type *am* (line 27) are delivered first, then messages of type $remove$ (line 28) and finally messages of type add (line 30).

We explain these three points.

1. Let us denote by v_{prev} the view preceding a view change, and by v_{new} the view after the view change. The initialization allows processes in $v_{new}.S - v_{prev}.S$ to initialize their variables, see lines 7-8 (set of processes *joined*, view v, set of messages *adelivered*, counter k used to identify consensus instances). The last field *newProcesses* (line 36, line 8) identifies the newly joining processes. This information is used in line 9, where the message received in line 8 is sent to the set *newProcesses*. Line 9 is needed in the case all processes in $v_{prev}.S - v_{new}.S$ crash before or during execution of

[10] For example, consensus can be specified independently of its solvability.

Algorithm 1 Atomic multicast and membership change (code of process p)

```
 1: Variables:
 2:   $v$                                   {current view;  notation: $v \equiv (v.id,\ v.S)$}
 3:   $joined$                              {set of processes that have joined the group}
 4:   $k$                                   {integer used to identify the different instances of consensus}
 5:   $received, adelivered$                {set of messages received, respt. adelivered}

 6: Initialization:
 7:   $received \leftarrow \emptyset$
 8:   wait until $init1\text{-}receive(joined, v, adelivered, k, newProcesses)$
 9:   $init1\text{-}send(joined, v, adelivered, k, newProcesses)$ to $newProcesses$
10:   execute $init^v$

11: Once Initialization done:

12:   To execute $amulticast(m)$:                                    {Task 1}
13:     $send(am, m)$ to $v.S$

14:   To execute $join\text{-}inv(x)$:                               {Task 2}
15:     $send(add, x)$ to $v.S$

16:   To execute $leave\text{-}inv(x)$:                              {Task 3}
17:     $send(remove, x)$ to $v.S$

18:   upon $receive(type, m)$ for the first time :                   {Task 4}
19:     if $sender(m) \neq p$ then $send(type, m)$ to $v.S$
20:     $received \leftarrow received \cup \{(type, m)\}$

21:   upon $received - adelivered \neq \emptyset$ :                  {Task 5}
22:     $k \leftarrow k + 1$
23:     $a\_undelivered \leftarrow received - adelivered$
24:     $propose(k, v.S, a\_undelivered)$

25:     wait until $decide(k, adeliver^k)$
26:     $prevView \leftarrow v$
27:     for all messages $(am, m)$ in $adeliver^k$ in some deterministic order: $adeliver(m)$
28:     for all messages $(remove, x)$ in $adeliver^k$ in some deterministic order:
29:       if $(x \in v.S)$ then $leave\text{-}exec(x)$           {view $v$ becomes $(v.id{+}1,\ v.S - \{x\})$}
30:     for all messages $(add, x)$ in $adeliver^k$ in some deterministic order:
31:       if $(x \notin joined)$ then $join\text{-}exec(m)$      {view $v$ becomes $(v.id{+}1,\ v.S \cup \{x\})$}
32:     $adelivered \leftarrow adelivered \cup adeliver^k$
33:     if $prevView \neq v$ then
34:       $newProcesses \leftarrow v.S - prevView.S$
35:       $joined \leftarrow joined \cup newProcesses$
36:       $init1\text{-}send(joined, v, adelivered, k, newProcesses)$ to $newProcesses$
37:       if $p \in v.S$ then $\forall (type, m) \in (received - adelivered) : send(type, m)$ to $newProcesses$
```

line 36: it ensures that if one newly joining process terminates its initialization, then all joining processes do so unless they crash.

2. Line 37 is for Validity R1 (respt. GR1): if a g-correct process executes *amulticast*(m), then it eventually *adelivers* m. Consider a process p executing *amulticast*(m) (line 12) in view v. To guarantee that m is eventually *adelivered* by p, there must exist a view v' in which for all non crashed processes q, we have $m \in received_q$. For this purpose, whenever the view changes, if p is in the new view, it sends the messages received but not yet delivered to the joining processes (line 37) (if p is not in the new view, p is g-faulty, i.e., the Validity property is trivially ensured).

3. In each batch $adeliver^k$, messages of type *am* are delivered before messages of type *add* or *remove* for the following reason. Let consensus #k be executed by the processes in the current view $v = (i, \{p, q\})$, and let $adeliver^k = \{(add, r), (am, m)\}$ be the decision. Consider the following two options:

   (i) delivery of (*add,r*) followed by the delivery of (am, m), or

   (ii) delivery of (am, m) followed by the delivery of (*add,r*).

In case (i), the new view $v' = (i + 1, \{p, q, r\})$ is first installed, and m is delivered in the new view v'. According to the specification (property R2), process r must also deliver m, which requires a special mechanism. In case (ii), m is delivered in view v, and then the new view v' is installed. Here r does not have to deliver m. Delivering messages of type *am* before messages of type *add* or *remove* makes the solution simpler.

The simplicity argument also leads us to deliver messages of type *remove* before messages of type *add*: with this solution, the newly joining processes do not have to start by executing *leave-exec*() actions. Finally, consider the case of two joining processes r and s, where r is added before s, i.e., we have $(i+1, \{p, q, r\})$ and $(i+2, \{p, q, r, s\})$. Formally, after the initialization of r, process r should execute *join-exec*(s). With our solution, r and s both install $(i+2, \{p, q, r, s\})$ as their initial view. However, according to Remark 3 at the end of Sect. 2.7.3, the two solutions are equivalent.

4.3 Algorithm 2: Group initialization

Algorithm 2 is the code to be executed when initializing some group g. First the initial view v_0 is defined (line 2), and then the group state of processes in view v_0 is initialized by the message sent at line 3.

Algorithm 2 Group initialization

```
1: Group initialization:
2:     $v_0 \leftarrow (0, \text{any subset of } \Pi)$
3:     init1-send($v_0.S$, $v_0$, $\emptyset$, 0, $v_0.S$) to $v_0.S$
```

4.4 Proof of atomic multicast

We prove in this section that Algorithm 1 solves atomic multicast defined by the properties R1-R4 and A5.

Lemma 1
Let $a\_undelivered_p^k$ denote the value of $a\_undelivered_p$ when p executes $propose(k, v.S, -)$, let $adelivered_p^k$ denote the value of $adelivered_p$ when p executes $propose(k, v.S, -)$, and let v_p^k denote the view of p when p executes $propose(k, v.S, -)$. For two processes p and q and all $k \geq 1$:

1. *If $adelivered_p^k$ and $adelivered_q^k$ are both defined, then we have $adelivered_p^k = adelivered_q^k$.*
2. *If v_p^k and v_q^k are both defined, then we have $v_p^k = v_q^k$.*
3. *If $a\_undelivered_p^k$ is defined, for any message $m \in a\_undelivered_p^k$, if q is g-correct and q is in v, then eventually $m \in received_q$ or $m \in adelivered_q$.*
4. *If p executes $propose(k, v.S, -)$, then if q is v-correct, it eventually executes $propose(k, v.S, -)$.*
5. *If, after $propose(k, v.S, -)$, p adelivers messages in $adeliver_p^k$, then q v-correct eventually adelivers messages in $adeliver_q^k$, and $adeliver_p^k = adeliver_q^k$.*[11], [12]

Proof The proof is by simultaneous induction on (1), (2), (3), (4) and (5).[13]

Base step (1): (1) trivially holds, since $adelivered_p^1 = adelivered_q^1 = \emptyset$.

Base step (2): (2) also trivially holds, since $v_p^1 = v_q^1 = v_0$.

Base step (3): We now show that (3) holds for $k = 1$. If $m \in a\_undelivered_p^1$, since $adelivered_p = \emptyset$, p has received m at line 18. Since q is in v, p has sent m to q at line 19. If q is g-correct (and so correct), since channels are reliable, q eventually receives m, and inserts m in $received_q$ (line 20).

Base step (4): We next show that if p executes $propose(1, v_0.S, -)$, then q that is v-correct eventually executes $propose(1, v_0.S, -)$. We distinguish two cases: (i) v_0 is not the last view of q; (ii) v_0 is the last view of q. In case (i), by definition q installs a view after v_0. So q must have executed $propose(1, v_0.S, -)$.
Case (ii): If v_0 is the last view of q, then q is v_0-correct is equivalent to q is g-correct. If p executes $propose(1, v_0.S, -)$, then $received_p$ must contain some message m. If q never executes $propose(1, v_0.S, -)$, $received_q$ remains empty (since $adelivered_q$ is initially empty). A contradiction with (3). Thus, q eventually executes Task 5 and $propose(1, v_0.S, -)$.

Base step (5): Finally, we show that if p adelivers messages in $adelivered_p^1$, then q that is v-correct eventually adelivers messages in $adeliver_q^1$, and $adeliver_p^1 = adeliver_q^1$. From the algorithm, if p adelivers messages in $adeliver_p^1$, it previously executed $propose(1, v.S, -)$. From part (4) of the lemma, all v-correct processes eventually execute $propose(1, v.S, -)$. By termination and uniform integrity of consensus, every process that is v-correct eventually executes $decide(1, adeliver^1)$ and *adelivers* messages in $adeliver^1$. By uniform agreement of consensus, all processes that execute $decide(1, adeliver^1)$ do so with the same value $adeliver^1$.

Induction step (1): We assume that the lemma holds for all $1 \leq k \leq l-1$ and show that $adelivered_p^l = adelivered_q^l$. We consider three cases: (i) v is not the initial view of p nor of q, (ii) v is the initial view of p only, and (iii) v is the initial view of p and q.
Case (i) By line 32, for $k > 1$ we have $adelivered_p^k = adelivered_p^{k-1} \cup adeliver_p^{k-1}$ and the same for q. By the induction hypothesis of part (1) of this lemma we have

[11] $adeliver_p^k$ is the decision value of p following $propose(k, v.S, -)$, not to be confused with $adelivered_p^k$.

[12] To simplify the notation *adeliver* means here $adeliver$ (line 27), *leave-exec* (line 29) and *join-exec* (line 31).

[13] The proof of (4) and (5) is adapted from [4].

$adelivered_p^{l-1} = adelivered_q^{l-1}$. By the induction hypothesis of part (5) of this lemma we have $adeliver_p^{l-1} = adeliver_q^{l-1}$. Together we have that $adelivered_p^l = adelivered_q^l$.
Case (ii) By line 36 there exists some process r for which v is not the initial view, such that $adelivered_p^l = adelivered_r^l$. By case (i), we have that $adelivered_r^l = adelivered_q^l$, and so $adelivered_p^l = adelivered_q^l$.
Case (iii) By line 36 there exists processes r and s (possibly $r = s$) for which v is not the initial view, such that $adelivered_p^l = adelivered_r^l$ and $adelivered_q^l = adelivered_s^l$. By case (i) we have $adelivered_r^l = adelivered_s^l$. Together we have that $adelivered_p^l = adelivered_q^l$.

Induction step (2): We now show that $v_p^l = v_q^l$. By the induction hypothesis of part (2) and (5), we have (i) $v_p^{l-1} = v_q^{l-1}$ and (ii) $adeliver_p^{l-1} = adeliver_q^{l-1}$. By (ii), the view changes applied by p to v_p^{l-1} are the same as the view changes applied by q to v_q^{l-1}. Together with (i), we have $v_p^l = v_q^l$.

Induction step (3): We now show that (3) holds for l. We consider three cases: (i) $a\_undelivered_p^{l-1}$ not defined, (ii) $a\_undelivered_p^{l-1}$ defined and $m \notin a\_undelivered_p^{l-1}$, and (iii) $a\_undelivered_p^{l-1}$ defined and $m \in a\_undelivered_p^{l-1}$.
Case (i) Here view v is the initial view of p. In this case, p has received m in view v, and has sent m to all processes in v (line 19), including to q. Since q is g-correct, eventually $m \in received_q$.
Case (ii) Since $m \notin a\_undelivered_p^{l-1}$, by the definition of $a\_undelivered_p^{l-1}$, p has received m either in view v or in view v_p^{l-1}. If p has received m in view v, it has sent m to all processes in v, including to q. Since q is g-correct, eventually $m \in received_q$. If p has received m in view v_p^{l-1} and q in v_p^{l-1}, by the same argument, eventually $m \in received_q$. If p has received m in view v_p^{l-1} and q not in v_p^{l-1}, then v is the initial view of q. If $m \in adeliver_p^{l-1}$, then p sends m to q at line 36, and since q is g-correct, eventually $m \in adelivered_q$. If $m \notin adeliver_p^{l-1}$, since $m \in received_p$, then p sends m to q at line 37, and since q is g-correct, eventually $m \in received_q$.
Case (iii) Consider view v_p^{l-1}. If q in v_p^{l-1}, the result follows immediately from the induction hypothesis. If q not in v_p^{l-1}, then v is the initial view of q, and we can apply the same reasoning as in case (ii).

If $m \in adeliver_p^{l-1}$, then p sends m to q at line 36, and since q is g-correct, eventually $m \in adelivered_q$. If $m \notin adeliver_p^{l-1}$, since $m \in received_p$, then p sends m to q at line 37, and since q is g-correct, eventually $m \in received_q$.

Induction step (4): We show that if p executes $propose(l, v.S, -)$, then q that is v-correct eventually executes $propose(l, v.S, -)$. If v is not the last view of q, then by definition q installs a view after v, i.e., executes $propose(l, v.S, -)$. So let us assume that v is the last view of q.

We prove the result by contradiction. Assume that q never executes $propose(l, v.S, -)$. When p executes $propose(l, v.S, -)$, $received_p$ must contain some message m that is not in $adelivered_p$. Thus m is not in $adelivered_p^l$. From part (1) of this lemma, $adelivered_p^l = adelivered_q^l$. So m is not in $adelivered_q^l$.
By part (3) of this lemma, since $a\_undelivered_p^l$ is defined, $m \in a\_undelivered_p^l$ and q v-correct, eventually $m \in received_q$ in view v. Since $m \notin adelivered_q^l$, there is a time after which the condition $received - adelivered_p \neq \emptyset$ that triggers Task 5 (line 21) becomes true in view v for q. So q eventually executes Task 5 and $propose(l, v.S, -)$. A contradiction.

Induction step (5): We now show that if p *adelivers* messages in $adeliver_p^l$, then q *adelivers* messages in $adeliver_q^l$ and $adeliver_p^l = adeliver_q^l$. Since p *adelivers* messages in $adeliver_p^l$, it must have executed $propose(l, v.S, -)$. By part (4) of this lemma, all v-correct processes eventually execute $propose(l, v.S, -)$. If v is not the last view of q, then by definition q eventually installs a view after v, i.e., executes $decide(l, -)$. If v is the last view of q, then q is v-correct is equivalent to q is g-correct. By termination of consensus, q eventually executes $decide(l, -)$ and *adelivers* messages in $adeliver_q^l$. By uniform agreement of consensus, all processes that execute $decide(l, adeliver^l)$ do so with the same $adeliver^l$. So $adeliver_p^l = adeliver_q^l$. □

Lemma 2 *The Uniform Agreement property of atomic multicast is satisfied.*

Proof We prove that if process p *adelivers* message m in view v, then all processes that are v-correct eventually *adeliver* m. So assume that p *adelivers* m in view v. By line 27 (messages of type *am* are *adelivered* before messages of type *add* or *remove*), no message is *adelivered* by any process in view v before the first execution of $propose(k, view.S, -)$ where $view = v$. So, if p *adelivers* m in view v, this happens after p has executed $propose(k, v.S, -)$. By Lemma 1 part (5), q eventually *adelivers* m. □

Lemma 3 *The Uniform Total Order property of atomic multicast is satisfied.*

Proof Immediate from Lemma 1 part (5), and the fact that processes *adeliver* messages in each batch in the same deterministic order. □

Lemma 4 *The Validity property of atomic multicast is satisfied.*

Proof [14] We have to prove that if a g-correct process executes *amulticast*(m), then it eventually *adelivers* m.

[14] Adapted from [4].

The proof is by contradiction. Suppose a g-correct process p *amulticasts* m in view v_{i_0}, but never *adelivers* m. By Lemma 2 no process ever *adelivers* m.

At line 13 or 19, p sends m to all processes in view v_{i_0}. Let $v_{i_0}, v_{i_1}, v_{i_2}, \ldots, v_{i_{\text{last}}}$ be the sequence of views of p after it has *amulticast* m. The sequence is finite, since the set Π is finite (Sect. 2.2), and a process can be added and removed from the view at most once. Since m is never *adelivered* by p, for all the views v_{i_j}, $i_0 \leq i_j \leq i_{\text{last}}$, message m is never in the decision $adeliver^k$ of any consensus. By line 36, process p sends m to all processes in $v_{i_1} - v_{i_0}$, $v_{i_2} - v_{i_1}$, $\ldots$, $v_{i_{\text{last}}} - v_{i_{\text{last}-1}}$. So p sends m to all processes in $v_{i_{\text{last}}}$, and every $v_{i_{\text{last}}}$-correct process q eventually receives m and inserts it in $received_q$. Since processes never *adeliver* m, they never insert m in $adelivered$. Thus for every $v_{i_{\text{last}}}$-correct process q, there is a time after which m is permanently in $received_q - adelivered_q$. From Algorithm 1 and Lemma 1 part (4), there is a k_1 such that for all $l \geq k_1$, all $v_{i_{\text{last}}}$-correct processes execute $propose(l, v_{i_{\text{last}}}.S, -)$, and they do so with sets that always include m.

Since all faulty processes eventually crash, there is a k_2 such that no faulty process, and so no g-faulty process, executes $propose(l, v_{i_{\text{last}}}.S, -)$ with $l \geq k_2$. Let $k = max(k_1, k_2)$. Since all $v_{i_{\text{last}}}$-correct processes execute $propose(k, v_{i_{\text{last}}}.S, -)$, by termination and uniform agreement of consensus, all $v_{i_{\text{last}}}$-correct processes execute $decide(k, adeliver^k)$ with the same $adeliver^k$. By uniform validity of consensus, some process q has executed $propose(k, v_{i_{\text{last}}}.S, adeliver^k)$. From our definition of k, $adeliver^k$ contains m. Thus all $v_{i_{\text{last}}}$-correct processes, including p, *adeliver* m. A contradiction that concludes the proof. □

Lemma 5 *The Uniform Integrity property of atomic multicast is satisfied.*

Proof Same argument as in the proof in [4]. □

Lemma 6 *The Uniform Same View Delivery property of atomic multicast is satisfied.*

Proof We have to prove that if two g-correct process p and q *adeliver* m in view v^p (for p) and in v^q (for q), then $v^p = v^q$. The result follows immediately from Lemma 1 part (5), which holds for messages of type am, as well as for messages of type add and $remove$, and from the fact that processes *adeliver* messages in each batch in the same deterministic order (line 27). □

Theorem 1 *Algorithm 1 solves atomic multicast.*

Proof Follows directly from Lemma 2 to Lemma 6. □

4.5 Proof of group membership

The proof of the group membership properties G0, GR1–GR4 and GA5 are straightforward. Properties GR1, GR2, GR4, GA5 follow immediately from the corresponding properties R1, R2, R4, A5 of atomic multicast. The first part of GR3 – $\forall p$, every g-correct process q executes *join-exec*(p) (respt. *leave-exec*(p)) at most once – follows from lines 31 (respt. line 29) of Algorithm 1. The second part of GR3 – $\forall p$, every g-correct process q executes *join-exec*(p) (respt. *leave-exec*(p)) only if *join-inv*(p) (respt. *leave-inv*(p)) was previously invoked – follows from R3.

We discuss now G0. For the processes in the initial view v_0, the property of the initial view follows trivially from the initialization algorithm (Algorithm 2). For the other processes, the property of the initial view follows from the lines 31 and 36 of Algorithm 1.

4.6 Practical issues

4.6.1 Evaluation of the solution

We now briefly evaluate our atomic multicast algorithm. The evaluation can be done from various points of view: (1) efficiency of the algorithm compared to a non uniform solution, (2) efficiency of the algorithm compared to a solution based on a group membership service, (3) robustness of the algorithm. We also discuss possible optimizations of the algorithm.

Uniform vs. non uniform algorithm. As already observed in Sect. 2.9, the above algorithm, which solves the *uniform* atomic multicast problem, is less efficient than a non uniform algorithm. However, non uniform algorithms also have drawbacks: they may lead to an inconsistent system state (see Sect. 2.9).

Atomic broadcast: Group membership or failure detector based solution. In a recent study, Urbán et al. [28] compare the performance of (1) the atomic broadcast algorithm of [4] when consensus is solved using the Chandra-Toueg $\diamond\mathcal{S}$ rotating coordinator algorithm, with (2) the performance of a (uniform) atomic broadcast algorithm based on group membership (the algorithm uses a sequencer; if the sequencer process is suspected, it is removed from the group by the membership service, and a new process becomes the sequencer). Since the above algorithm is close to the atomic broadcast algorithm of [4], the results of [28] can directly be reused to evaluate the above solution. These results, obtained by simulation, show the latency as a function of the throughput for various scenarios: (1) *normal-steady*, i.e., the performance with no crashes nor failure suspicions, (2) *crash-steady*, i.e., the performance in the steady state long after a crash, (3) *suspicion-steady*, i.e., the performance with wrong failure suspicions, and (4) *crash-transient*, i.e., the performance immediately after a crash. In the normal-steady scenario, the two algorithms have the same performance. In the crash-steady scenario, the group membership based algorithm performs slightly better. In the suspicion-steady and in the crash-transient scenarios, the failure detector based algorithm outperforms the group membership based algorithm.

The results of the suspicion-steady scenario show a very important property of algorithms based on failure detectors compared to algorithms based on group membership: better performance in the case of wrong failure suspicions (which can happen frequently whenever the failure detection timeout has been set to a value close to the average message transmission delay, to ensure a short crash detection time). Indeed, upon a false failure suspicion, an algorithm based on group membership performs two costly operations: removal of the suspected process from the group, followed by a join of the same process.

While an atomic broadcast algorithm does not "require" a group membership service, it may nevertheless be necessary at some point to remove a silent process, in order to get back system resources, e.g., buffer space. However, it is better to base this removal on the base of system resources, rather than on timeouts [5].

Robustness of the solution. Tolerating wrong failure suspicions can not only improve the efficiency of the algorithm, it also leads to a very robust algorithm. In [27], the atomic broadcast algorithm of [4] has been evaluated on a cluster of PCs under extreme conditions for the case $n = 3$ (three replicas): very high load of atomic broadcasts (10 000 per second) and a very small timeout value, approaching the resolution of the clock (1 ms). The algorithm always terminated, even under these extreme conditions. Since our atomic multicast algorithm is close to the atomic broadcast algorithm of this experiment, it has the same robustness.

Optimizations. Can the performance of our atomic multicast algorithm be improved? Some solutions have been proposed in the literature. [15] suggests to increase the parallelism among the different atomic broadcast instances. [2] explores another solution, in which there is uncertainty among the participants to consensus. However, experiments are needed to understand the performance gain that can actually be obtained by these techniques. Indeed, the time complexity or message complexity of an algorithm are not necessarily good predictors of its real performance [26]. Typically, time complexity ignores contention on the network and contention on the CPU (for sending or receiving messages).

4.6.2 About reliable channels

Despite the fact that in Sect. 4.1 we assume *reliable* channels (an assumption common to many papers, e.g., [4]), our solution does not exclude link failures and network partitions. To do so, an implementation must provide a low level layer that implements reliable channels over (fair) lossy links. Does TCP provide such a layer? Not really. Indeed, consider a process p on node n_p that executes $send(m)$ to a correct process q located on a different node n_q. After the TCP $send$ function returns on p, there is no guarantee that m will eventually be received by q. For example, if node n_p crashes while m is in the TCP buffer on n_p, and has not yet been sent to q, then m is lost.

From a practical point of view, the adequate abstraction is the *quasi-reliable* channel, where the "no loss" property of Sect. 4.1 is replaced with the following weaker "no loss" property: if p sends m to q, and q *as well as* p are correct, then q eventually receives m.[15] Algorithm 1 remains correct with quasi-reliable channels instead of reliable channels. The same holds for the Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm.

A quasi-reliable channel from p to q can be opened on p using the primitive *open-qr-channel(q)*, and closed by the primitive *close-qr-channel(q)* (*qr* stands for *quasi-reliable*). These two primitives are related as follows to the view change mechanism. Whenever p installs a new view v that includes some new process q, then p invokes *open-qr-channel(q)*; whenever p installs a new view v from which some process q is removed, then p invokes *close-qr-channel(q)*. By executing *close-qr-channel(q)*, the channel between p and q looses its quasi-reliable property: any message sent by p but not yet received by q might get lost.

5 Solving reliable multicast with membership changes

5.1 Thrifty solution

In this section we discuss the solution of reliable multicast. As already noted, a trivial solution is obtained by using atomic multicast to solve reliable multicast:

- upon *rmulticast*(m), execute *amulticast*(m);
- upon *adeliver*(m), execute *rdeliver*(m).

This solution is however unnecessarily costly, since the consensus oracle is used in every run, although it is obviously not needed in runs in which *join-inv* and *leave-inv* are not called. We want a cheaper solution that satisfies the following thriftiness properties (adapted from [1]):

- If *join-inv* and *leave-inv* are not invoked, then the consensus oracle is never used.
- If there is a time after which *join-inv* and *leave-inv* are no more invoked, then there is a time after which the consensus oracle is no more used.

A reliable multicast solution that satisfies these two properties is said to be *thrifty* with respect to the consen-

[15] Actually, TCP does not provide this abstraction, since a TCP connection can be broken even though the two endpoints have not crashed. However, TCP can easily be extended to provide the semantics of quasi-reliable channels, see [9].

sus oracle. Such a solution can be obtained by using *generic multicast* (instead of atomic multicast).

5.2 Generic multicast *vs.* generic broadcast

We define *generic multicast* (or *dynamic generic broadcast*) as the dynamic extension of the (static) generic broadcast group communication primitive [1, 20, 21]. Generic broadcast is a flexible primitive defined by *gbroadcast*, *gdeliver* and parametrized by a (symmetric and non-reflexive) conflict relation on the set of messages: conflicting messages are delivered in the same order on all processes, while non-conflicting messages may be delivered in any order. Our conflict relation is the following: view change messages (of type add or $remove$) conflict with all other messages, while reliable multicast messages (of type rm) do not conflict with other reliable multicast messages. So reliable multicast messages are ordered with respect to view change messages, but not with respect to other reliable multicast messages.

Generic broadcast (static) is formally defined by the properties of (static) reliable broadcast (Validity, Uniform Agreement, Uniform Integrity) and the following Uniform Generalized Order property:

- *Uniform Generalized Order:* If messages m and m' conflict, and some process (whether correct or faulty) *gdelivers* messages m before it *gdelivers* message m', then a process *gdelivers* m' only after it has *gdelivered* m.

We define (dynamic) *generic multicast* similarly. The primitives *gmulticast* and *gdeliver* are defined by the properties R1 – R4 of reliable multicast and the following Modified Uniform Generalized Order property:

- *Modified Uniform Generalized Order:* If messages m and m' conflict, and some process (whether g-correct or g-faulty) *gdelivers* messages m in view v before it *gdelivers* message m', then every process p in v *gdelivers* m' only after it has *gdelivered* m.

5.3 Thrifty solution based on generic multicast

In the same way as the atomic broadcast algorithm of [4] has been adapted to solve atomic multicast (see Algorithm 1), the thrifty generic broadcast algorithms of [1] can be adapted to provide a thrifty solution for generic multicast.[16] Using this solution, reliable multicast is easy to solve:

- *rmulticast*(m) translates to *gmulticast*(rm, m), where rm is the type of the message;
- *join-inv*(x) translates to *gmulticast*(add, x), where add is the type of the message;
- *leave-inv*(x) translates to *gmulticast*($remove, x$), where $remove$ is the type of the message.

Messages of type rm do not conflict with themselves, but conflict with messages of type add and $remove$, while messages of type add and $remove$ conflict with all messages. With this conflict relation, view change messages are ordered (1) with respect to other view change messages, and (2) with respect to reliable multicast messages. (1) ensures the GM Uniform Total Order property GA5, and (2) ensures the Uniform Same View Delivery property R4.

[16] The algorithms in [1] are thrifty with respect to the atomic broadcast oracle. In these algorithms, calls to the atomic broadcast oracle need to be replaced with calls to our atomic multicast algorithm (Algorithm 1), which invokes the consensus oracle.

6 Conclusion

The paper has brought a new insight to the specification of dynamic reliable broadcast – called reliable multicast – and dynamic atomic broadcast – called atomic multicast. The specifications that we have given in this paper are simple and close to those of static group communication. This shows that the gap between static and dynamic group communication can be made very small.

The paper has also given another perspective on the implementation of dynamic group communication. While group membership has almost universally been considered to be the basic layer of a group communication infrastructure, the paper proposes a different – and probably simpler – solution, in which atomic multicast is the basic layer – on top of which group membership can easily be solved, an idea that already appears in [15] and [18].

Incidentally, the paper shows that, contrary to a common belief, it is not the ability to force processes to become faulty (or g-faulty) by excluding them from the group that *alone* allows to exploit non uniform specifications. The exploitation of non uniform specifications requires the ability to force processes to become faulty (or g-faulty) to be under the control of the atomic broadcast algorithm.

To summarize, the paper has shown that the specification and the implementation of dynamic group communication can be simple, i.e., easily understood. This should contribute to clarify a topic that has always been difficult to understand by outsiders.

Acknowledgements I would like to thank Sergio Mena for his comments on an earlier version of this paper, Sam Toueg for the numerous discussions about the specification of the group membership problem and the anonymous referees for their valuable comments and suggestions.

Research funded by the Swiss National Science Foundation under grant number 21-67715.02 and by Hasler Stiftung under grant number DICS-1825.

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Thrifty generic broadcast. In: Proceedings of the 14th International Symposium on Distributed Computing (DISC), pp. 268–282. Toledo, Spain, October 2000. Springer Berlin Heidelberg New York, LNCS (1914)

2. Bar-Joseph, Z., Keidar, I., Lynch, N.: Early delivery dynamic atomic broadcast. In: 16th Int Symposium on Distributed Computing (DISC), pp. 1–16. Toulouse, France. Springer Berlin Heidelberg New York, LNCS 2508 (2002)
3. Birman, K., Joseph, T.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. **5**(1), 47–76 (1987)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
5. Charron-Bost, B., Défago, X., Schiper, A.: Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently. In: Proceedings of 21st IEEE Symposium on Reliable Distributed Systems (SRDS), pp. 244–249. Osaka, Japan (2002)
6. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surveys **4**(33), 1–43 (2001)
7. Cristian, F.: Reaching agreement on processor group membership in synchronous distributed systems. Distrib. Comput., **4**(4), 175–187 (1991)
8. Défago, X., Schiper, A., Urban, P.: Totally ordered broadcast and multicast algorithms: taxonomy and survey. ACM Comput. Surveys **4**(36), 1–50 (2004)
9. Ekwall, R., Urbán, P., Schiper, A.: Robust TCP connections for fault-tolerant computing. J. Inf. Sci. Eng. **19**, 503–516 (2002)
10. Fischer, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**, 374–382 (1985)
11. Friedman, R., van Renesse, R.: Strong and weak virtual synchrony in horus. Technical Report 95-1537. Department of Computer Science, Cornell University (1995)
12. Guerraoui, R.: Revisiting the relationship between non-blocking atomic commitment and consensus. In: 9th Intl. Workshop on Distributed Algorithms (WDAG), pp. 87–100. Le Mont-St-Michel, France. Springer Berlin Heidelber New York, LNCS 972 (1995)
13. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. IEEE Comput. **30**(4), 68–74 (1997)
14. Hadzilacos, V., Toueg, S.: Fault-Tolerant broadcasts and related problems. Technical Report 94–1425. Department of Computer Science, Cornell University (1994)
15. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
16. Lin, K., Hadzilacos, V.: Asynchronous group membership service. In: 13th. Intl. Symposium on Distributed Computing (DISC), pp. 79–93. Bratislava, Slovakia, September 1999. Springer Berlin Heidelber New York, LNCS (1693)
17. Malloth, C., Schiper, A.: View synchronous communication in large scale networks. In: ESPRIT Basic Research BROADCAST, Third Year Report, Vol. 4 (1995)
18. Melliar-Smith, P.M., Moser, L.E., Agrawala, V.: Processor membership in asynchronous distributed systems. IEEE Trans. Parallel Distrib. Syst. **5**(5), 459–473 (1994)
19. Mena, S., Schiper, A., Wojciechowski, P.: A step towards a new generation of group communication systems. In: Proc. Int. Middleware Conference, pp. 414–432. Rio de Janeiro, Brazil. Springer Berlin Heidelberg New York, LNCS 2672 (2003)
20. Pedone, F., Schiper, A.: Generic broadcast. In: 13th. Intl. Symposium on Distributed Computing (DISC), pp. 94–108. Bratislava, Slovakia. Springer Berlin Heidelberg New York, LNCS (1693) (1999)
21. Pedone, F., Schiper, A.: Handling message semanticas with generic broadcast protocols. Distrib. Comput. **15**(2), 97–107 (2002)
22. Ricciardi, A.M., Birman, K.P.: Using process groups to implement failure detection in asynchronous environments. In: Proc. of the 10th ACM Symposium on Principles of Distributed Computing (PODC), pp. 341–352. Montreal, Quebec, Canada (1991)
23. Schiper, A., Sandoz, A.: Uniform reliable multicast in a virtually synchronous environment. In: IEEE 13th Intl. Conf. Distributed Computing Systems, pp. 561–568. Pittsburgh, Pennsylvania, USA (1993)
24. Schiper, A., Toueg, S.: From set membership to group membership: a separation of concerns. Technical Report IC/2003/56. EPFL-IC Faculty (2003)
25. Schneider, F.B.: Implementing fault tolerant services using the state machine approach: a tutorial. Comput. Surveys **22**(4), 299–319 (1990)
26. Urbán, P., Défago, X., Schiper, A.: Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms. In: IEEE 9th Int Conf on Computer Communications and Networks (ICCCN), pp. 582–589. Las Vegas, USA (2000)
27. Urbán, P., Défago, X., Schiper, A.: Chasing the FLP impossibility result in a lan or how robust can a fault tolerant server be? In: 20th IEEE Symp. on Reliable Distributed Systems (SRDS), pp. 190–193. New Orleans, USA (2001)
28. Péter, U., Ilya, S., André, S.: Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In: IEEE Int Conf on Dependable Systems and Networks (DSN), pp. 645–654. San Fransisco, USA (2003)

André Schiper graduated in Physics from the ETHZ in Zurich in 1973 and received the PhD degree in Computer Science from the EPFL (Federal Institute of Technology in Lausanne, Switzerland) in 1980. He has been a professor of computer science at EPFL since 1985, leading the Distributed Systems Laboratory. During the academic year 1992–1993, he was on sabbatical leave at the University of Cornell, Ithaca, New York, and in 2004-2005 at the Ecole Polytechnique near Paris. His research interests are in the area of dependable distributed systems, middleware support for dependable systems, replication techniques (including for database systems), group communication, distributed transactions, and, recently MANETs (mobile ad-hoc networks). From 2000 to 2002, he was the chair of the steering committee of the International Symposium on Distributed Computing (DISC). He has taken part in several European projects. He is currently a member of the editorial board of Distributed Computing, and of IEEE Transactions on Dependable and Secure Computing.