Dynamic Group Communication

André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
e-mail: andre.schiper@epfl.ch

Abstract

Group communication is the basic infrastructure for implementing fault-tolerant replicated servers. While group communication is well understood in the context of static systems (in which all processes are created at the start of the computation), current specifications of dynamic group communication (in which processes can be added and removed during the computation) are not satisfactory.

The paper proposes new specifications for dynamic reliable broadcast (which we call reliable multicast*), dynamic atomic broadcast (which we call* atomic multicast*) and group membership in the primary partition model. In the special case of a static system, our specifications are identical to the well known static specifications. The specification of group membership is derived from the specification of atomic multicast.*

The paper also 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. In the context of reliable multicast, we introduce the notion of thrifty *solution, and show that such a solution can be obtained by relying on a thrifty generic multicast algorithm.*

Keywords: *Distributed systems, fault-tolerance, group communication, primary partition, specification, implementation, 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.

Group communication is well understood in the context of a static system, in which all processes are created at the start of the computation. Specification of static group communication

can be found in [7], and implementation of reliable broadcast and atomic broadcast is discussed for example in [4]. However, a static system 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 review by Chockler et al. [5], dynamic group communication has not yet reached the level of maturity of static group communication. The goal of the paper is to contribute to a better understanding of dynamic group communication.

The key component of a dynamic system is the *group membership* service, which is responsible for adding and removing processes during the computation [6, 10].[1] Group membership is strongly related to *view synchrony*, a property ensuring that processes deliver the same set of messages between two membership changes [3, 11]. However, the specifications of group membership and of view synchrony are unsatisfactory for three reasons. First, the current specifications are complex [5]. Second, these specifications do not exclude trivial solutions [2]. Finally, the specification of reliable broadcast and atomic broadcast – which result from the specifications of group membership and view synchrony – are very different from the well understood and simple specifications of reliable broadcast and atomic broadcast in a static system. For example, reliable broadcast (in a static system) and view synchrony (in a dynamic system) provide somewhat similar guarantees. However, their specification are very different.

The paper proposes new specifications for dynamic group communication in the primary partition model – specifically dynamic reliable broadcast and dynamic atomic broadcast – and shows how they can be implemented. To avoid the lengthy expression "dynamic reliable broadcast", respectively "dynamic atomic broadcast", we denote the former by *reliable multicast* and the latter by *atomic multicast*. We adopt the same approach as in [7]: we start with the specification of reliable multicast, and then extend it to atomic multicast. In the special case of a static system our specifications are identical to the well known static specifications. The specification of group membership is derived from the specification of atomic multicast.

We also discuss the solutions of atomic multicast, group membership and reliable multicast. Atomic multicast can be solved by reduction to consensus, similarly to (static) atomic broadcast [4]. Group membership can be solved using atomic multicast. The solution of atomic multicast can trivially be extended to solve reliable multicast. We discuss another "thrifty" solution, which uses generic broadcast [8, 1, 9].

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

2 Specification of Dynamic Group Communication

In this section we define *reliable multicast* (or dynamic reliable broadcast), *atomic multicast* (or dynamic atomic broadcast) and *group membership*.

[1] In this paper we consider only the so called *primary partition* membership problem.

2.1 System model

We consider an asynchronous distributed system composed of processes taken from a finite set Π, which communicate by message passing. The system is dynamic: processes may be created after the beginning of the computation (and processes may terminate before the end of the computation).

Group communication assumes "groups": a group is a subset of processes. In order to avoid unnecessary complexity, we assume in the paper that there exists only one group, and that all processes belong to this single group. This assumption means that (i) creation of process p, and (ii) p joining the group, are the same operation. Similarly, (i) termination of process p, and (ii) p leaving the group, are the same operation. As a consequence, there is no difference between "processes in the system" and "group". In the rest of the paper we consistently use only the term "group".

A process may be created only once. So, if process p has terminated or has crashed, it cannot be resurrected (p cannot be created a second time). From a practical point of view, this stills allows a crashed process to recover, but it has to take a different name. A *view* is a subset of Π that represents the membership of the group at that moment.[2] Given process p, we denote by v^p the current membership of the group as perceived by p. Since a given membership may occur more than once at a given process, e.g., for p we could have the following sequence of membership $\{p, q\}$, $\{p, q, r\}$ (r added to the membership), $\{p, q\}$ (r removed from the membership), we add the notion of *view instance*, denoted by $inst(v^p)$. The first instance of a given view at some process has instance number 1, and each subsequent instance has an instance number incremented by 1. While the membership of p is $(v^p, inst(v^p))$ we say that process p *is in view* $(v^p, inst(v^p))$ or simply that process p *is in view* v^p.[3] The initial view of the group is denoted by v_0, and the first view of each process in v_0 is v_0. If process p is in view v and later in some other view v', we say that v' is *after* v for p. We say that v is the *last* view of p if there is no view for p after v. Note that a static system is a special case of our dynamic system model, in which the view never changes and is always equal to $v_0 = \Pi$ (view $v_0 = \Pi$ is also the last view of all processes).

Processes fail by crashing (we do not consider Byzantine failures). In a static system, a process is said to be *correct* if it never crashes. We extend this definition by first introducing the notion of *correctness of a process in a view*. Process p *is correct in view* v if (1) $p \in v$ and eventually p is in view v, and

(2a) if v is the last view of all processes in v, then p never crashes, or

(2b) if v' is the view that is immediately after v for some process in v, then $p \in v'$.

Based on this definition, we can define the notion of a correct process in a dynamic system. Let v_p^{init} be the initial view of process p. We say that process p is correct if and only if p is correct in v_p^{init} and in all its subsequent views. Otherwise, p is *faulty*. Note that if the system is static, Π is the initial and the last view of all processes. By item (2a), we have the same definition as in a static system.

[2] In this paper we do not attach an identifier to a view (e.g., a view *number*).

[3] The glossary at the end of the paper summarizes our definitions.

2.2 Reliable Multicast: first tentative

The usual definition of (static) reliable broadcast assumes that all processes in Π are created at the beginning of the computation. Our definition of *reliable multicast* considers the dynamic system model introduced above.

Reliable multicast is defined by the two primitives *rmulticast* and *rdeliver*, and – as a first tentative – the following three properties:[4]

R1 *Validity:* If a 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 correct in v^p eventually *rdeliver* m.

R3 *Uniform Integrity:* For any message m, (i) every process *rdelivers* m only once, and (ii) only if m was previously *rmulticast* by $sender(m)$.

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 $v_0 = \{p, q\}$ is the initial view of the group and

- process p executes *rmulticast*(m) and later *rdelivers* m in view $v^p = v_0$,
- process q *rdelivers* m in view $v^q = \{p, q, r\}$,
- process r never *rdelivers* m, and $\{p, q, r\}$ is the last view of r (so, since r does not crash, r is correct in view $\{p, q, r\}$).

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

2.3 Reliable Multicast: second tentative

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 in v^q (for q), then $v^p = v^q$ and $inst(v^p) = inst(v^q)$.

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 system is static, i.e., if the view is always equal to Π, then the Same View Delivery property is trivially satisfied, and the other properties are identical to those that define reliable broadcast [7]. So, if the systems is static, reliable multicast is equivalent to (static) reliable broadcast.

[4] All the broadcast primitive that we define in this paper are *uniform* [7]. To simplify the notation, we drop the word "uniform" from the various broadcast types.

2.4 Atomic Multicast

We define atomic multicast by the two primitives *amulticast* and *adeliver*, and – similarly to [7] for static systems – 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 [7] forbids "holes" in the delivery sequence of messages. However, we must adapt the definition in [1] since dynamic creation of processes poses a specific problem: we do not want process p to have to deliver messages delivered before its creation. We express this using views:

A5 *Uniform Total Order:* If some process (whether correct or 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 this property 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 was created before the delivery of m, and consequently p must *adeliver* m. If p is not in v and has *adelivered* m', then p was created after the delivery of m, and does not have to *adeliver* m.

2.5 Group Membership

To complete the definition of dynamic group communication, we need to specify the events that change views. In other words we need to specify the *group membership problem*. We consider the most simple 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 the 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. We only address the second issue.

Process p requests to add process r to the group by *invoking* the operation $join(r)$. Process r is included in the view of process q (possibly $q = p$) when q *executes* $join(r)$. Similarly, process p requests to remove r from the group by invoking the operation $leave(r)$ (p can remove itself by invoking $leave(p)$). Process r is removed from the view of process q when q executes $leave(r)$. Process r learns that it has been removed from the group when it executes $leave(r)$.

According to our model (Section 2.1), a process r that executes $leave(r)$ is faulty, since it is not member of the next view. This makes our specification very simple. A process that has left the group is similar to a faulty process: if p has left the group, it is no more obliged to deliver messages and cannot join again the group later.[6]

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*). We want *join-exec* and *leave-exec* to be executed in the same total order by all processes. So we simply define group membership by the properties of atomic multicast, while renaming *join-inv* / *join-exec* and *leave-inv* / *leave-exec*:

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

[6] However, process p could join again under a different name, similarly to a crashed process, see Section 2.1.

- *join-inv*(x), $x \in \Pi$, is equivalent to *amulticast*$(add(x))$, and *join-exec*(x) is equivalent to *adeliver*$(add(x))$.
- *leave-inv*(x), $x \in \Pi$, is equivalent to *amulticast*$(remove(x))$, and *leave-exec*(x) is equivalent to *adeliver*$(remove(x))$.

However, the Uniform Integrity property of atomic multicast needs to be adapted. The reason is the following. Consider view $v = \{p, q, r\}$ and processes p, q both invoking *leave-inv*(r) in view v. We do not want *leave-exec*(r) to be executed more than once by each process, as this would lead to fictitious view changes, e.g., from view $v = \{p, q\}$ to the view $v' = \{p, q\}$. Fictitious view changes may also result from invoking for example *leave-inv*(r) in a view that does not contain r. Our specification of GM Uniform Integrity allows only "real" view changes.[7] To summarize, we define group membership by an initialization property (given later) and the following properties:

GR1 *GM Validity:* Same as the Validity property R1, with the above renaming rules.

GR2 *GM Uniform Agreement:* Same as the Uniform Agreement property R2, with the above renaming rules.

GR3 *GM Uniform Integrity:* $\forall p$, every correct process q executes *join-exec*(p) (respt. *leave-exec*(p)) only once, and only if *join-inv*(p) (respt. *leave-inv*(p)) was previously invoked. Moreover, q executes *join-exec*(p) (respt. *leave-exec*(p)) only if p was not already created (respt. if p is in the current view of q).

GR4 *GM Uniform Same View Delivery:* Same as the Uniform Same View Delivery property R4, with the above renaming rules.

GA5 *GM Uniform Total Order:* Same as the Uniform Total Order property A5, with the above renaming rules.

Upon occurrence of *join-exec*(x) at process p in view v, the view of p atomically becomes $v \cup \{x\}$. Upon occurrence of *leave-exec*(x) at process p in view v, the view of p atomically becomes $v - \{x\}$. We introduce the notation *join-exec*$()^v$ (respt. *leave-exec*$()^v$) to denote that the execution of join (respt. leave) leads to the view v. When process p executes *join-exec*$()^v$ or *leave-exec*$()^v$, we say that p *installs* view v.

The first event of a process is denoted by *init*^{v}, where v is the initial view of the process. The initialization property 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:* For every process p, the first event *init*^{v} is such that (i) p belongs to view v, and (ii) either $v = v_0$ or there exists a process q ($q \neq p$) which installs view v.

[7] This would not lead to specification problems, but may be disturbing from a user perspective.

It should be noted that the ordering property GA5 is actually redundant. Let us denote by $op(x)$, $x \in \Pi$, either *join-exec*(x) or *leave-exec*(x). GA5 states that if some process, say q, 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 $op(x)$. We show that this follows from the other properties.

The proof is by induction on the number mop of membership operations $op()$ performed by q. The base case is for $mop = 2$. Assume that q executes $op(x)$ in view v before executing $op(y)$ in view v' (so v' is obtained from v by execution of $op(x)$). Moreover assume that some process p has executed $op(y)$. By GR2, p has executed $op(y)$ in view v'. Since v' is after v for q, v' cannot be the initial view v_0. So v is the initial view v_0. By hypothesis, we have $p \in v$, so $p \in v_0$. By GR3, since $p \in v$, $op(x)$ cannot be *join-exec*(p); together with G0, this means that v_0 is also the initial view of p. Since p executes $op(y)$ in view $v' \neq v_0$, p must have executed a view change operation in view v_0 leading to v'. This operation is necessarily $op(x)$, which completes the proof of the base case. The induction step can be proven in a similar way.

2.6 Examples

We illustrate the above specifications on two examples. The first example shows a message *amulticast* in view v by p and *adelivered* in a different view. The initial group view v_0 is $\{p, q\}$ and no process crashes:

- local history of p: *init*$^{\{p,q\}}$ – *amulticast*(m) – *join-exec*$(r)^{\{p,q,r\}}$ – *adeliver*(m) – ...
- local history of q: *init*$^{\{p,q\}}$ – *join-inv*(r) – *join-exec*$(r)^{\{p,q,r\}}$ – *adeliver*(m) – ...
- local history of r: *init*$^{\{p,q,r\}}$ – *adeliver*(m) – ...

The second example shows a process r that is faulty because of the execution of *leave-exec*(r). The initial group view v_0 is $\{p, q, r\}$ and no process crashes:

- local history of p: *init*$^{\{p,q,r\}}$ – *amulticast*(m) – *adeliver*(m) – *leave-exec*$(r)^{\{p,q\}}$ – ...
- local history of q: *init*$^{\{p,q,r\}}$ – *leave-inv*(r) – *adeliver*(m) – *leave-exec*$(r)^{\{p,q\}}$ – ...
- local history of r: *init*$^{\{p,q,r\}}$ – *adeliver*(m) – *leave-exec*$(r)^{\{p,q\}}$

This second example shows an interesting feature of our specification. Process r, despite the fact that it is not correct because of the execution of *leave-exec*(r), has to execute *adeliver*(m). The reason is the following. Process p executes *adeliver*(m) in view $\{p, q, r\}$ before it executes *leave-exec*(r). By the Uniform Total Order property (Section 2.4), process r in view $\{p, q, r\}$ can execute *leave-exec*(r) only after it has executed *adeliver*(m).

More generally, any process r that is not correct because of the execution of *leave-exec*(r) must execute all operations (i.e., message delivery, join/leave operations) that take place *after* r joined the group and *before* the execution of *leave-exec*(r).

3 Comparison with current specifications

We compare now the above specifications with those of Chockler et al. [5]. We discuss the properties in the same order as in [5]: first safety and then liveness properties. Moreover, the safety properties in [5] start with the properties of group membership, which is usual for current specifications.

3.1 Group membership safety properties

In [5], 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.*

  Our specification is weaker than Self Inclusion.
  By our GM Initialization property, the initial view of a process satisfies the Self Inclusion property. Self Inclusion is also trivially satisfied by all subsequent views installed by some process p, until p executes *leave-exec*(p). However, after *leave-exec*(p) process p is allowed to install new views to which it does not belong.

- *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.*

  Local Monotonicity does not apply to our specifications, since we do not attach identifiers to views (Section 2.1).

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

  This property is trivially ensured by our GM Initialization property, which requires *init*^{v} to be the first event executed by a process.

There is only one non-basic safety property in [5]:

- *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 [5].

  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 GM Uniform Total Order property 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 GM Initialization property, 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 satisfies the relevant safety properties of [5]. The opposite is not true. For example, the GM Uniform Integrity property does not hold in [5], where view changes are not required to be "justified".

[8] We give only the informal specification of [5]. For additional information, please refer to [5].

3.2 Multicast safety properties

In [5], 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.

- *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.

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 Same View Delivery. This property could be added, but as noticed in [5], 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. As mentioned in [5], Same View Delivery has been introduced to avoid blocking of the application that can happen with Sending View Delivery.

Three properties are given in the category *virtual synchrony*:

- *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'.*

  Our specification ensures this property. Consider two processes p and q that install view v in view v', and message m delivered (*rdelivered* or *adelivered*) by p in view v'. Because of the Self-Inclusion Property 3.1, since p and q install view v, we have $p, q \in v$. If $q \in v$, then according to our model, q is correct in view v'. Since p has delivered m in view v', by the Uniform Agreement and the Uniform Same View Delivery properties, q also delivers m in view v'.

- *Property 4.6 (Transitional Set)* We skip this complex property, which is not relevant in our context.

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

- *Property 4.7 (Agreement on Successors): If a process p installs view v in view v', and if some process q also installs v and q is a member of v' then q also installs v in v'.* This property is presented as an alternative to the Transitional Set property.

  (a) To show that our specification satisfies this property, consider the function f defined as above: (i) $f(v_0) = 0$, (ii) if some process installs view v' immediately after view v, then $f(v') = f(v) + 1$. By the GM Uniform Total Order property f is a function. Moreover, given two processes p and q and two views v^p and v^q such that $f(v^p) = f(v^q)$, the GM Uniform Total Order property ensures that $v^p = v^q$.

  (b) Because of the Self-Inclusion Property 3.1, and since p and q install view v, we have $p, q \in v$. Since $q \in v'$, v is not the initial view of q: let $v'' \neq v'$ be the view that q installs immediately before v. So p installs v' immediately before v, and q installs v'' immediately before v, i.e., we have $f(v') = f(v) - 1$ and $f(v'') = f(v) - 1$, and also $f(v') = f(v'')$. By the result of (a), we conclude that $v' = v''$, which means that q installs v' immediately before v (or using the terminology of [1], q installs v in v').

3.3 Ordering and reliability properties

The ordering and reliability properties in [5] 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 is stronger than Strong Total Order, as it requires processes to deliver a prefix of the common sequence of messages delivered within each view [5]. Our specification, by the Uniform Total Order property, ensures Reliable Total Order.

3.4 Liveness properties

The liveness properties of [5] are difficult to compare to our specifications. This is because events that trigger view changes do not appear in the specification of [5]. For example, our GM Validity and GM Uniform Agreement properties do not hold in [5]. Even the weaker *Non-Uniform* GM Agreement property does not hold.

Concerning message multicast, we can make the following observations. Firstly, [5] does not ensure our Validity property: a correct process executing *amulticast*(m) or *rmulticast*(m) is not forced to deliver m. Secondly, [5] does not ensure our Uniform Agreement property. It does not even satisfy the non-uniform version of the same property. Consider some correct process p that delivers m (in view v): nothing requires another correct process q to deliver m.

3.5 Discussion

The comparison has shown that our specification satisfies the safety properties of [5] that are meaningful in our context. Moreover, our specification has safety and liveness properties that do not appear in [5].

4 Solving Atomic Multicast with membership changes

We start by describing the solution of atomic multicast, which is simpler to discuss than the solution of reliable multicast.

4.1 Extending the system model

We complete the system model of Section 2.1. 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 $propose(k, v, val)$ and $decide(k, decision)$. When process p executes $propose(k, v, val)$, the parameter k identifies a specific instance of consensus, v (which stands for *view*) denotes the set of processes that have to reach agreement, and val is p's initial value. Given instance k_0 of consensus, the consensus oracle ensures the following property. If all processes that execute $propose(k_0, v, val)$ do so with the same parameter v, then all processes in v that do not crash eventually decide (Termination), the $decision$ is one of the initial values val (Validity), and no two processes in v decide differently (Uniform Agreement) [4].

4.2 Solution by reduction to consensus

Atomic multicast, with view changes, can be solved by reduction to consensus, much like atomic broadcast [4]. We explain mainly the differences between the reduction in a static setting and the reduction in our dynamic setting.

4.2.1 Two parts: Algorithm 1 and Algorithm 2

We decompose the solution in two parts. Algorithm 1 (page 13) transforms calls to *join-inv*(x), respt. *leave-inv*(x), into calls to atomic multicast of $add(x)$, respt. $remove(x)$. Algorithm 2 (page 14) solves atomic multicast in a constant view. In order to express the transformation of Algorithm 1, we introduce the primitive *atomulticast*$(type, m)$,[10] where $type$ can take the values add, $remove$ or am. A call to *amulticast*(m) translates to *atomulticast*(am, m), a call to *join-inv*(x) translates to *atomulticast*(add, x), and a call to *leave-inv*(x) translates to *atomulticast*$(remove, x)$ (see Algorithm 1, Task 1, Task 2 and Task 3).

[10] The low level primitive *atomulticast* should not be confused with *amulticast*.

Calls to ***atomulticast***$(type, m)$ atomically multicasts typed messages. These messages are delivered upon execution of $atomdeliver(type, m)$ (line 1/13).[11] Algorithms 2 implements ***atomulticast***$(type, m)$ and ensures the following additional property: each sequence of events $atomdeliver(add, -)$, $atomdeliver(remove, -)$ is terminated by the fictitious delivery of a message of type $processCreation$ (line 1/17), which triggers the creation of the joining processes. Here is an example:[12]

$$\ldots,\ ad(am, m)\ ,\ ad(add, r)\ ,\ ad(remove, p)\ ,\ ad(processCreation, -)\ ,\ ad(am, m')\ , \ldots$$

In this example, if $ad(am, m)$ takes place in some view v, then $ad(am, m')$ takes place in the new view $v' = v \cup \{r\} - \{p\}$.

The initialization of a newly created process is done in Algorithm 1 by executing $init1\text{-}send(c, v, joiningProcesses)$ (line 1/20), which sends (over a reliable channel) the new view v – together with the set c of already created processes and the set of joining processes – to the joining processes. We use the notation $init1\text{-}send$ and $init1\text{-}receive$ (line 1/2) to distinguish them from the standard $send$ and $receive$ primitives used in Algorithm 1.

The initialization in Algorithm 1 satisfies the following property: if not all processes in the intersection of the old and the new view crash before sending the new view, then all joining processes get the new view v', and start execution. Moreover, if one correct joining process gets the initial view v', then by line 1/4 all correct joining processes get the initial view v'.

4.2.2 Algorithm 2: atomic multicast in a constant view

Algorithm 2 solves atomic multicast in a constant view by reduction to consensus. The algorithm 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 with our dynamic algorithm is that the sequence of consensus is no more executed by a constant set of processes. Consensus #k is executed by the processes that are members of the group when consensus #k is started. Our dynamic algorithm has four additional differences that need to be explained:

1. Initialization of the joining processes (lines 2/1 to 2/4, and line 2/20).[13]
2. Line 2/22, by which p sends the messages received but not yet delivered (i.e., $received_p - adelivered_p$) to the joining processes.
3. The deterministic delivery function, which delivers messages of type am before messages of type add or $remove$ (line 2/17).

[11] We use the following notation: *line* f/l stands for line l in Figure f.

[12] $atomdeliver(type, m)$ is simply denoted by $ad(type, m)$.

[13] Note that the initialization of Algorithm 1 and of Algorithm 2 are independent. Each algorithm independently starts when its initialization part is completed.

Algorithm 1 Dynamic group communication: atomic multicast (main code of process p)

1: **Initialization**:
2: **wait until** $init1\text{-}receive(created, view, newProcesses)$
3: $c \leftarrow created$; $v \leftarrow view$; $prevView \leftarrow \emptyset$
4: $init1\text{-}send(c, v, newProcesses)$ to all processes in $newProcesses$
5: execute $init^v$

6: **Once Initialization done**:

7: To execute *amulticast*(m): {***Task 1***}
8: $atomulticast(am, m)$

9: To execute *join(x)*: {***Task 2***}
10: $atomulticast(add, x)$

11: To execute *leave(x)*: {***Task 3***}
12: $atomulticast(remove, x)$

13: **upon** $atomdeliver(type, m)$: {***Task 4***}
14: **if** $type = am$ **then** *adeliver*(m)
15: **if** ($type = add$) and ($m \notin c$) **then** *join-exec*(m) {v becomes $v \cup \{m\}$ }
16: **if** ($type = remove$) and ($m \in v$) **then** *leave-exec*(m) {v becomes $v - \{m\}$ }
17: **if** $type = processCreation$ **then**
18: *create-processes*$(v - prevView)$
19: $c \leftarrow c \cup v$
20: $init1\text{-}send(c, v, v - prevView)$ to all processes in $(v - prevView)$
21: $prevView \leftarrow v$

4. The delivery of the fictitious message of type $processCreation$ (line 2/18), which leads Algorithm 1 to create processes and to send them the initial view.

Algorithm 2 Atomic multicast (code of process p)

1: **Initialization**:
2: **wait until** $init2\text{-}receive(aSet, i, newProcesses)$
3: $k \leftarrow i$; $received \leftarrow \emptyset$; $adelivered \leftarrow aSet$
4: $init2\text{-}send(aSet, i, newProcesses)$ to all processes in $newProcesses$

5: **Once Initialization done**:

6: **upon** *atomulticast*$(type, m)$: {***Task 1***}
7: $send(type, m)$ to all processes in v

8: **upon** $receive(type, m)$ for the first time : {***Task 2***}
9: **if** $sender(m) \neq p$ **then** $send(type, m)$ to all processes in v
10: $received \leftarrow received \cup \{(type, m)\}$

11: **upon** $received - adelivered \neq \emptyset$: {***Task 3***}
12: $k \leftarrow k + 1$
13: $a\_undelivered \leftarrow received - adelivered$
14: $propose(k, v, a\_undelivered)$
15: **wait until** $decide(k, adeliver^k)$

16: $prevView \leftarrow v$
17: atomically $atomdeliver$ all messages in $adeliver^k$ in some deterministic order such that messages of type add or $remove$ are delivered last
18: **if** atomdelivered messages of type add or $remove$ **then** $atomdeliver(processCreation, -)$
19: $adelivered \leftarrow adelivered \cup adeliver^k$
20: $init2\text{-}send(adelivered, k, v - prevView)$ to all processes in $(v - prevView)$
21: **if** $p \in v$ **then**
22: $\forall (type, m) \in (received - adelivered) : send(type, m)$ to all processes in $(v - preView)$

We comment now these four 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 $v_{new} - v_{prev}$ to initialize their set $adelivered$ (messages already delivered) and their counter k (used to identify consensus instances). Line 2/4 is needed in the case all processes in $v_{prev} - v_{new}$ crash before or during execution of line 2/20: it ensures that if one newly joining process terminates its initialization, then all joining processes do so unless they crash.

(2): Line 2/22 is for Validity (if a correct process executes *amulticast*(m), then it eventually *adelivers* m). Consider a process p executing *atomulticast*$(type, m)$ (line 2/6) in view v. To

guarantee that $(type, m)$ is eventually $atomdelivered$ 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 2/22) (if p is not in the new view, p is 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 $v = \{p, q\}$, 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);
ii) delivery of (am, m) followed by the delivery of (add, r).

In case (i), the new view $v' = \{p, q, r\}$ is first installed, and m is delivered in the new view v'. According to the specification, 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.

(4): Once all messages of the current batch have been delivered, the joining processes can be created. This is the goal of the fictitious message of type $processCreation$ (line 2/18).

4.2.3 System initialization

Algorithm 3 System initialization

1: **System initialization**:
2: $v_0 \leftarrow$ any subset of Π
3: *create-processes*(v_0)
4: $init1\text{-}send(v_0, v_0, v_0)$ to all processes in v_0
5: $init2\text{-}send(\emptyset, 0, \emptyset)$ to all processes in v_0

Algorithm 3 is the code to be executed at system initialization. First the initial view v_0 is defined (line 2), then the processes in v_0 are created (line 3), and finally these processes are initialized upon reception of message (v_0, v_0, v_0) – line 2 of Algorithm 1 – and of message $(\emptyset, 0, \emptyset)$ – line 2 of Algorithm 2.

4.3 Proof of atomic multicast

We prove in this section that Algorithm 1 together with Algorithm 2 satisfy the properties R1 - R4 and A5 of atomic multicast.

Lemma 4.1

Let $a\_undelivered_p^k$ denote the value of $a\_undelivered_p$ when p executes $propose(k, v, -)$, $adelivered_p^k$ the value of $adelivered_p$ when p executes $propose(k, v, -)$, and v_p^k the view of p when p executes $propose(k, v, -)$. 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 correct and $q \in v$, then eventually $m \in received_q$ or $m \in adelivered_q$.*
4. *If p executes $propose(k, v, -)$, then q correct in v eventually executes $propose(k, v, -)$*
5. *If, after $propose(k, v, -)$, p adelivers messages in $adeliver_p^k$, then q correct in v eventually adelivers messages in $adeliver_q^k$, and $adeliver_p^k = adeliver_q^k$.*[14] [15]

PROOF: The proof is by simultaneous induction on (1), (2), (3), (4) and (5).[16]

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 2/8. Since $q \in v$, p has sent m to q at line 2/9. If q is correct, since channels are reliable, q eventually receives m, and inserts m in $received_q$ (line 2/10).

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

Base step (5): Finally, we show that if p adelivers messages in $adelivered_p^1$, then q correct in v 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 $propose(1, v, -)$. From part (4) of the lemma, all processes correct in v eventually execute $propose(1, v, -)$. By termination and uniform integrity of consensus, every process correct in view v eventually executes $decide(1, adeliver^1)$ and then atomdelivers and later *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,

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

[15] To simply the notation, in Section 4.3 $adeliver$ means $adeliver$ (line 2/14) or *join-exec* (line 2/15) or *leave-exec* (line 2/16).

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

Case (i): By line 2/19, 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 $adelivered_p^{l-1} = adelivered_q^{l-1}$. 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 2/20 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 2/20 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 applies 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. For p 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 2/9), including to q. Since q is correct, eventually $m \in received_q$.
Case (ii): Since $m \notin a\_undelivered_p^{l-1}$, p receives m either in view v, or in the view v' that precedes immediately v. If p has received m in view v, it has sent m to all processes in v, including to q. Since q is correct, eventually $m \in received_q$. If p has received m in view v' and $q \in v'$, by the same argument, eventually $m \in received_q$. If p has received m in view v' and $q \notin v'$, then v is the initial view of q. If $m \in adeliver_p^{l-1}$, then p sends m to q at line 2/20, and since q is 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 2/22, and since q is correct, eventually $m \in received_q$.
Case (iii): Let v' be the view that precedes immediately view v. If $q \in v'$, the result follows immediately from the induction hypothesis. If $q \notin v'$, 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 2/20, and since q is 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 2/22, and since q is correct, eventually $m \in received_q$.

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

We prove the result by contradiction. Assume that q never executes $propose(l, v, -)$. When p executes $propose(l, v, -)$, $received_p$ 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 $adelivered_q^l$.
Since q never executes $propose(l, v, -)$ and $m \notin adelivered_q^l$, then $m \notin adelivered_q$. By part (3) of this lemma, since $a\_undelivered_p^l$ is defined, $m \in a\_undelivered_p^l$ and q correct in v, 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 3 (line 2/11) becomes true in view v. So

q eventually executes Task 3 and $propose(l, v, -)$. 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 $propose(l, v, -)$. By part (4) of this lemma, all processes correct in v eventually execute $propose(l, v, -)$. 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 correct in v is equivalent to q is 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 4.2 *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 correct in v eventually ***adeliver*** m. So assume that p ***adelivers*** m in view v. By line 2/17 (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 $propose(k, view, -)$ where $view = v$. So, if p ***adelivers*** m in view v, this happens after p has executed $propose(k, v, -)$. By Lemma 4.1 part (5), q eventually ***adelivers*** m. □

Lemma 4.3 *The Uniform Total Order property of atomic multicast is satisfied.*

PROOF: Immediate from Lemma 4.1 part (5), and the fact that processes ***adeliver*** messages in each batch in the same deterministic order. □

Lemma 4.4 *The Validity property of atomic multicast is satisfied.*

PROOF:[17] We have to prove that if a correct process executes ***amulticast***(m), then it eventually ***adelivers*** m. The proof is by contradiction. Suppose a correct process p ***amulticasts*** m in view v_{i_0}, but never ***adelivers*** m. By Lemma 4.2 no process ever ***adelivers*** m.

At line 2/7 or 2/9, p sends m to all processes in view v_{i_0}. Let v_{i_0}, v_{i_1}, v_{i_2}, ..., $v_{i_{last}}$, be the sequence of views of p after it has ***amulticast*** m. The sequence is finite, since the set Π is finite (Sect. 2.1), 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}, $i_0 \leq i_j \leq i_{last}$, message m is never in the decision $adeliver^k$ of any consensus. By line 2/20, process p sends m to all processes in $v_{i_1} - v_{i_0}$, $v_{i_2} - v_{i_1}$, ..., $v_{i_{last}} - v_{i_{last-1}}$. So p sends m to all processes in $v_{i_{last}}$, and every correct process q in $v_{i_{last}}$ eventually receives m and inserts it in $received_q$. Since processes never ***adeliver*** m, they never insert m in $adelivered$. Thus for every correct process q in $v_{i_{last}}$, there is a time after which m is permanently in $received_q - adelivered_q$. From Algorithm 2 and Lemma 4.1 part (4), there is a k_1 such that for all $l \geq k_1$, all correct processes in $v_{i_{last}}$ execute $propose(l, v_{i_{last}})$, and they do so with sets that always include m.

[17] Adapted from [4].

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

Lemma 4.5 *The Uniform Integrity property of atomic multicast is satisfied.*

PROOF: Same argument as in the proof in [4].

Lemma 4.6 *The Uniform Same View Delivery property of atomic multicast is satisfied.*

PROOF: We have to prove that if two 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 4.1 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 2/17). □

Theorem 4.7 *Algorithm 1 and Algorithm 2 solve atomic multicast.*

PROOF: Follows directly from Lemma 4.2 to Lemma 4.6 □

4.4 Proof of group membership

The proof of the group membership properties GR1 - GR4 (and GA5) are straightforward. Properties GR1, GR2, GR4 (and GA5) follow immediately from the corresponding properties R1, R2, R4 (and A5) of atomic broadcast. The first part of GR3 – $\forall p$, every correct process q executes *join-exec*(p) (respt. *leave-exec*(p)) only once, and only if *join-inv*(p) (respt. *leave-inv*(p)) was previously invoked – follows from R4. The second part of GR4 – q executes *join-exec*(p) (respt. *leave-exec*(p)) only if p was not already created (respt. p is in the current view of q) – follows from line 15 (respt. line 16) of Algorithm 1.

5 Solving Reliable Multicast with membership changes

In this section we discuss the solution of reliable multicast. The system model is as above (see Section 4.1).

5.1 Reliable Multicast with membership changes: brute force solution

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 has a drawback. 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 invoked. We would like a 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 consensus oracle. Such a solution can be obtained by using generic multicast instead of atomic multicast.

5.2 Static Generic Broadcast vs. Dynamic Generic Multicast

Static generic broadcast is a flexible group communications primitive [8, 1, 9] defined by *gbroadcast*, *gdeliver* and parametrized by a (symmetric and non-reflexive) conflict relation (denoted by $\sim$) 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. Intuitively, our conflict relation would be 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 would be ordered with respect to view change messages, but not with respect to other reliable multicast messages.

Formally, (static) generic broadcast is defined by the properties that define (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 message m', then a process *gdelivers* m' only after it has *gdelivered* m.

We define (dynamic) *generic multicast* in a similar way. The primitives *gmulticast* and *gdeliver* are defined by the properties R1 - R4 of reliable multicast together with the above Uniform Generalized Order property.

5.3 Thrifty solution

Thriftiness with respect to an oracle has been introduced in [1] as an implementation of generic broadcast that satisfies the following two properties:

- If all the messages that are broadcast do not conflict with each other, then the oracle is never used.
- If there is a time after which the messages broadcast do not conflict with each other, then there is a time after which the oracle is not used.

A thrifty generic broadcast solution allows us easily to get a thrifty reliable multicast solution. Algorithm 4 (page 22) gives the main code of process p. Compared to Algorithm 1, a new message type *rm* (for reliable multicast) has been added, calls to $atomulticast(type, m)$ have been replaced by calls to $gmulticast(type, m)$ (lines 4/8, 4/10, 4/12), and the conflict relation is the following:

- Messages of type rm do not conflict with themselves. By Algorithm 4, line 8, *rmulticast*(m) and *rmulticast*(m') lead to *gmulticast*(rm, m) and *gmulticast*(rm, m'). So messages m and m' and not ordered by the generic multicast algorithm.
- Messages of type add conflict with all other messages, and messages of type $remove$ conflict with all other messages. By Algorithm 4, lines 10 and 12, *join-inv*(x) respt. *leave-inv*(x), lead to *gmulticast*(add, x) respt. *gmulticast*$(remove, x)$. So, view change messages are ordered (1) with respect to other view change messages, and (2) also with respect to reliable multicast messages. (1) is required by the GM Uniform Total Order property GA5, while (2) is required by the Uniform Same View Delivery property R4.

A thrifty generic multicast algorithm can be obtained by adapting one of the generic broadcast algorithms of [1] that are thrifty with respect to the atomic broadcast oracle. Calls to the atomic broadcast oracle would result in calling our atomic multicast algorithm (Algorithm 2), which in turn would invoke the consensus oracle. We do not discuss this issue further, as it would lead us to discuss mainly details related to generic broadcast, which is outside the scope of this paper.

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 the 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 a new perspective on the implementation of dynamic group communication. While group membership has always 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.

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 useful comments on an earlier version of this paper.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'2000)*, October 2000.

Algorithm 4 Dynamic group communication: reliable multicast (main code of process p)

1: **Initialization**:
2: **wait until** $init1\text{-}receive(created, view, newProcesses)$
3: $c \leftarrow created$; $v \leftarrow view$; $prevView \leftarrow \emptyset$
4: $init1\text{-}send(c, v, newProcesses)$ to all processes in $newProcesses$
5: execute $init^v$

6: **Once Initialization done**:

7: To execute *rmulticast*(m): {***Task 1***}
8: $gmulticast(rm, m)$

9: To execute *join(x)*: {***Task 2***}
10: $gmulticast(add, x)$

11: To execute *leave(x)*: {***Task 3***}
12: $gmulticast(remove, x)$

13: **upon** $gdeliver(type, m)$: {***Task 4***}
14: **if** $type = rm$ **then** *rdeliver*(m)
15: **if** ($type = add$) and ($m \notin c$) **then** *join-exec*(m) {v becomes $v \cup \{m\}$ }
16: **if** ($type = remove$) and ($m \in v$) **then** *leave-exec*(m) {v becomes $v - \{m\}$ }
17: **if** $type = processCreation$ **then**
18: *create-processes*$(v - prevView)$
19: $c \leftarrow c \cup v$
20: $init1\text{-}send(c, v, v - prevView)$ to all processes in $(v - prevView)$
21: $prevView \leftarrow v$

[2] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report 95-1534, Department of Computer Science, Cornell University, August 1995.

[3] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.

[4] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[5] G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *Computing Surveys*, 4(33):1–43, December 2001.

[6] Flaviu Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–187, April 1991.

[7] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[8] F. Pedone and A. Schiper. Generic Broadcast. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*, pages 94–108. Springer Verlag, LNCS 1693, September 1999.

[9] F. Pedone and A. Schiper. Handling Message Semanticas with Generic Broadcast Protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[10] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[11] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 1993.

Glossary

Summary of the definitions used in the paper:

- *View:* A view v is a subset of the (finite) set Π of processes.
- *Instance of a view:* $inst(v)$ denotes the instance number of view v.
- *Process p is in view v:* While the membership of p is $(v, inst(v))$, we say that p is in view $(v, inst(v))$, or simply that p is in view v.
- *View v' is after view v for p:* Process p is in view v and later in view v'.

- *Last view of p:* View v is the last view of v if there is no view after v for p.
- *Process p is correct in view v:* Either (a) v is the last view of all processes in v and p never crashes, or (b) there is a view v' that is immediately after v for some process in v and $p \in v'$.
- *Process p is correct:* If and only if p is correct with respect to all its views.
- *Process p installs view v:* We say that p installs view v when p executes *join-exec*^{v} or *leave-exec*^{v}.