From Set Membership to Group Membership: A Separation of Concerns

André Schiper[*] Andre.Schiper@epfl.ch

Sam Toueg[†] sam@cs.toronto.edu

Abstract

We revisit the well-known group membership problem, and show how it can be considered a special case of a simpler problem, the set membership problem. In the set membership problem, processes maintain a set whose elements are drawn from an arbitrary universe: they can request the addition or removal of elements to/from that set, and they agree on the current value of the set. Group membership corresponds to the special case where the elements of the set happen to be processes. We exploit this new way of looking at group membership to give a simple and succint specification of this problem and to outline a simple implementation approach based on the state machine paradigm. This treatment of group membership separates several issues that are often mixed in existing specifications and/or implementation of group membership. We believe that this separation of concerns greatly simplifies the understanding of this problem.

1 Introduction

Group membership is one of the most widely studied problems in the area of fault-tolerant distributed computing. Despite the extensive literature on this problem, however, the existing specifications are still complex and difficult to understand. Since group membership is a paradigm for *simplifying* the design of fault-tolerant applications, it is important that its specification be as simple and clear as possible.

The first goal of this paper is to give a simple and succinct specification of the *primary partition group membership* problem.[1] Our starting point is to note that the complexity of the existing specifications of group membership is due to a mixing of concerns. In the literature, group membership is almost always defined as the problem of maintaining and agreeing on the set of processes that are currently up — a set that dynamically changes. We note that this problem consists of two sub-problems: (a) determining the set of processes that are currently up, and (b) ensuring that processes agree on the successive values of this set. Even though these two subproblems are orthogonal, most existing specifications and implementations deal with the combination as a single problem. Our approach is to completely decouple these two problems, and deal with each one separately, as we now explain.

We first specify a very simple problem, called *set membership*. In this problem, a set of processes Π maintain, and agree on the content of a dynamically changing set of elements drawn from an arbitrary universe. Processes can request the addition or removal of elements to/from the set, and the set changes accordingly. Each time the set changes, all processes are notified of the new value of the set. It is important to note that, in contrast to group membership, the set is *not* necessarily a set of processes. For example, set membership can be used to maintain and agree on the current value of a mailing list, a set of employees, or the set of unsold seats on a particular flight. Obviously, with set membership, the content of the set maintained by processes has nothing to do with failures.

[*]Research supported in part by the Swiss National Science Foundation under grant number 21-67715.02 and by Hasler Stiftung under grant number DICS-1825. Authors' address: Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland.

[†]Research supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. Author's address: Department of Computer Science, 10 King's College Road, University of Toronto, Toronto, Ontario, Canada M5S 3G4.

[1]This problem is different from the so-called *partitionable* group membership problem. See Section 6.

We then consider a special case of set membership, namely, the case where the set maintained by the processes in Π happens to be a subset of Π. Here processes in Π can request to add or remove *processes in* Π. We call this special case *group membership*, and the set maintained is called a *group*.

Our specification of the set and group membership problems is not concerned by the reason(s) why a process requests the addition or removal of an element to/from the set: such reasons are outside the scope of the problems. It is up to each *application* that uses group membership to decide what should cause the addition or removal of a process from the group. For example, in some application, a process may decide to request the removal of a process p from the group because the security clearance of p has been revoked. It may also decide to request the removal of p because p seems to have failed. Our specification of the group membership problem does not differentiate between these two requests.

With group membership, i.e., in the special case when the set maintained is a group of processes, there are some interesting variants which may be desirable to some applications. For example, recall that in set membership *all* the processes are notified of every set change. In contrast, in group membership, each time the group changes, only the current members of the group may have to be notified. In the paper, we consider this and a few other possible variants to the basic specification of group membership problem.

In addition to providing simple specifications to the set and membership problems, we also consider the problem of implementing such services. We first note that in *purely* asynchronous systems with failures, these problems are not solvable. These impossibility results are *not* because processes are trying to determine the set of processes are currently up (indeed the set membership problem cannot be solved even though the set maintained by processes has nothing to do with failures). Rather, it is because both problems embed some form of consensus on some set. [2]

Despite the above impossibility results, however, set membership and group membership can still be solved using randomization, unreliable failure detectors, or with other assumptions on the system (exactly as with consensus). We outline a stepwise implemention of these two services based on the well-known *state machine approach* [21, 27]. This systematic approach allows us to give simple and easily understandable implementations that clearly separate the service provided from orthogonal implementation issues (e.g., the degree of fault-tolerance desired, which processes provide the service, and the details of how it is done). In particular, for group membership, this approach clearly decouples the set of processes that provide the membership service and the set of processes that are currently in the group. This is in contrast to existing implementations, where the processes that provide the membership service (i.e., maintain the group) are the current members of the group (or a subset of this group).

Finally, we explain how our group membership service can be used for the special purpose of maintaining and agreeing on the set of processes *that are deemed to be operational* — which is the principal (and often only) goal of group membership protocols in the literature. As we mentioned above, this problem consists of two sub-problems: (a) determining the set of processes that appear to be operational, and (b) ensuring that processes agree on the successive values of this set. Roughly speaking, we can use a (reliable or unreliable) failure detector to solve sub-problem (a), and our group membership service to solve sub-problem (b).

Roadmap

The rest of the paper is organized as follows. The set membership and group membership services are specified in Sections 2 and 3, respectively. The problem of implementing such services is addressed in Section 4. Specifically, in Section 4.1 we show that these services cannot be implemented in purely asynchronous systems, and in Section 4.2 we outline a simple implementation approach based on the state machine paradigm (for systems where consensus or atomic broadcast can be solved, e.g., asynchronous systems with unreliable failure detectors or randomized algorithms). In Section 5, we explain how our group membership service can be used for the special purpose of maintaining and agreeing on

[2]The impossibility of a very weak form of group membership, one that did not try to track process failures, was already pointed out in [4]. The proof given in [4], however, was for systems with $n \geq 4$ processes. Our impossibility proof for the set and group membership problems holds for $n \geq 2$.

the (dynamically changing) set of processes that are currently operational. A brief section on related work and a conclusion complete the paper.

2 Specification of the set membership service

2.1 Preliminaries

We define a set membership service (SMS) that allows processes to maintain, and agree on the content of a dynamically changing set of elements drawn from an arbitrary universe. Processes are allowed to issue operations to add or remove elements of the set. The service executes these operations sequentially and notifies all the processes of the content of the set after each operation. With these notifications, processes agree on the i-th incarnation of the set for every i. We now describe the specification of this set membership service in more detail.

Consider a set Π of processes that maintain a dynamically changing set of elements drawn from an arbitrary (countable) universe $\mathcal{U}$. A *view* of the maintained set is a tuple (j, S), where $j \in N$, and $S \subseteq \mathcal{U}$: intuitively, view (j, S) indicates that the value of the j-th incarnation of the set is S. Processes issue operations to add or remove elements to/from the set. Operations are tuples of the form $(p, k, op(e))$, where $p \in \Pi$, $k \in N$, $op \in \{add, remove\}$, and $e \in \mathcal{U}$: intuitively, $(p, k, op(e))$ denotes that the k-th operation issued by process p is $op(e)$.

The *local history of process* p, denoted H_p, is a finite or infinite sequence of operations and views.[3] If a view v is in H_p, we say that p *installs* v. If o is the k-th operation in H_p, o is of the form $(p, k, *)$, and we say that p *issues operation* o. Moreover, if v is the last view before operation o in H_p, we say that p issues operation o *in view* v.

From the point of view of the set membership service, the set has some initial value, the service successively applies the process operations on the current value of the set, and each operation results in a new value of this set, i.e., a new view. Thus, the *global history of the membership service*, denoted H, is a finite or infinite sequence of alternating views and operations of the form: $H = v_0 — o_1 — v_1 — o_2 — v_2 — \ldots$ We say that v_0 is the *initial view of* H. Moreover, if H is finite it must terminate with a view, called the *final view of* H. We require that for all $i \geq 0$:

1. $v_i = (i, S)$ for some $S \subseteq \mathcal{U}$;
2. o_{i+1} is an operation (we say that o_{i+1} is *executed in view* v_i); and
3. operation o_{i+1} applied to view v_i results in view v_{i+1} in the natural way. More precisely, if $v_i = (i, S)$ and $o_{i+1} = (*, *, op(e))$ then $v_{i+1} = (i + 1, S')$ where S' is the set that results by applying $op(e)$ to set S.[4]

We assume that processes may fail, but they can do so only by benign failures [18], e.g., by crashing. In the following, C denotes the set of correct processes.

2.2 Specification of the basic SMS

A set of local histories of processes $\{H_p \mid p \in \Pi\}$ satisfies the specification of the basic set membership service, if there is a global history of the membership service H with the following properties:

S1: *View Sequence Agreement:* For every process p, the sequence of views in the local history H_p of p is a subsequence of the sequence of views in the global history H of the membership service.

S2: *Integrity:* Every operation executed by the membership service is requested by some process, and it is executed only once: $\forall o \in H \Rightarrow \exists p \in \Pi : o \in H_p$ and $\forall o, o' \in H : o \neq o'$

L1: *View installation:* Every view generated by the membership service is installed by every correct process: $\forall p \in C, \forall v \in H$: $v \in H_p$

L2: *Operation execution:* Every operation issued by a correct process is executed by the membership service: $\forall p \in C, \forall o \in H_p$: $o \in H$

[3]This sequence is arbitrary, and in particular operations and views do not have to alternate in a local history.
[4]Note that it is possible to have $S' = S$.

2.3 A variant of SMS

For some applications, a simple modification of the basic set membership service specified above may be desirable. To see this, suppose that the current view at process p is $v = (j, S)$, for some index j and set value S, and that p issues an operation to add or remove an element to/from S. This operation, which is issued in view v, may be semantically tied to v: the view v is the "context" of p's request. If this operation is received by the set membership service after the view v has changed (it may have been modified by an operation issued by another process), then this operation may no longer be appropriate in the new context. So, for some applications, operations issued in a given context (i.e., a view) should not be executed in a different context. We can enforce this restriction by adding the following safety requirement:

S3: *Same context:* An operation o is executed in view v only if o was issued in v.

This safety requirement however is incompatible with the liveness requirement that *all* operations of correct processes must be executed by the set membership service (requirement **L2**). To see this, suppose the initial view is $v_0 = (0, \{a, b\})$, and p issues $add(c)$ in v_0, while concurrently q issues $add(d)$ also in v_0 (and these are the only operations ever issued). By **S3**, it is clear that only one of these two operations can be executed by the set membership service, violating **L2**. Thus an application that requires **S3** must weaken **L2**. One possibility is to require that if a correct process p issues an operation o in view v then *some* operation, maybe a different one from o, is executed in v, i.e., the view v eventually changes. More precisely, we replace **L2** with the following weaker liveness requirement:

L2a: *Operation execution:* If a correct process p issues an operation o in a view v, then v is not the final view of the global history H of the membership service.

3 The group membership service: a specialization of SMS

The group membership service, denoted GMS, can be considered a special case of the set membership service where the elements of the set being maintained happen to be processes. More precisely, GMS is just the special case of SMS when $\mathcal{U} = \Pi$: the set included in each view is now a subset of Π. This subset represents the membership of the "group".

Since processes now issue operations to add and remove *processes* (a self-reference) to/from the set, some simple variants of the GMS requirements may be desirable for some applications. We consider below some of the possible variants (they make sense only for the special case where $\mathcal{U} = \Pi$). In the following, if view $v = (j, S)$ and process $p \in S$, we say that p *is in view* v.

1. In SMS every correct process must install every view of the set (requirement **L1**). For some GMS applications, however, it may not be necessary for a correct process to install the views that it does not belong to. For such an application, the liveness requirement **L1** can be weakened to:

   L1a: *View installation:* A correct process must install every view it belongs to:
   $\forall p \in C, \forall v \in H \colon p \in v \Rightarrow v \in H_p$ [5]

2. In SMS every process can issue an operation (i.e., request a group membership change) at any time. In GMS, an application may want to restrict the authority to request membership changes: a process may be allowed to request a membership change only if it belongs to the group, according to its current view. This optional requirement is specified by the following safety property:

   S4: *Authority to request a view change:* A process p can issue an operation in a view v only if p is in v.

3. To understand another possible requirement, consider the following scenario. Process p is in the current group and requests a group membership change, but when the membership service receives

[5] Note that this does not *forbid* a process to install views it does not belong to. In fact, when a process p is removed from a group, it may be useful for p to install the first view that does not contain it. This is what makes p aware that it was removed from the group.

the request, p is no longer in the group. Should the group membership service execute p's request? For some applications, processes "expelled" from the group should not have the ability to modify the group. This is enforced by the following safety requirement:

S5: *Authority to execute a view change:* An operation o issued by a process p is executed in a view v only if p is in v.

Note that it makes no sense to require **S5** but not **S4**, because doing so would mean the following: On one hand, by not requiring **S4**, we allow a process p that is not in the current view to issue an operation, while on the other hand, by requiring **S5**, we prevent p's operation to be executed, unless, by some lucky event the view changes to include p and p's operation happens to be received after this view change. In other words, by requiring **S5** but not **S4**, we allow p to issue its operation even though p is not in the current view, and "hope" that this operation will be received in some future view that does include p (otherwise the operation will be ignored, and so it is useless). This is not reasonable, and so we assume that any application that requires **S5** also requires **S4**.

Note that safety requirement **S5** is incompatible with liveness requirement **L2**. To see this, suppose the initial view is $v_0 = (0, \{p, q\})$, and p issues $remove(q)$ in v_0, while concurrently q issues $remove(p)$ also in v_0 (and these are the only operations ever issued). By **S5**, it is clear that only one of these two operations can be executed by the membership service, violating **L2**. Thus an application that requires both **S4** and **S5** must weaken **L2**.

Instead of **L2** we could require **L2a** (Section 2.3). But **L2a** is too weak: it allows the group membership service to ignore an operation if the view changes, *even when the issuer of this operation remains in the new view*, i.e., even when the execution of this operation is still authorized according to **S5**. A stronger liveness requirement that better matches **S5** is as follows. If a process p issues an operation in a view v[6], then p's operation must be executed by the membership service, unless the membership service eventually removes p from the view. More precisely, with **S4** and **S5** we can require:

L2b: *Operation execution:* If a correct process p issues an operation o in a view v, then either o is executed by the membership service, i.e., $o \in H$, or there exists a view v' after v in H such that p is not in v'.

Remarks: A few remarks about optional requirements **S3**, **S4** and **S5** are now in order. First note that the combination of **S3** and **S4** implies **S5**. Moreover, if an application requires **S1** through **S5**, the appropriate liveness requirement is **L2a** rather than **L2b**. This is because requirement **L2b** is not compatible with **S3**. To see this, suppose p issues operation o in view v, and the group membership service changes v to v' before o is executed (this could be due to another operation that was issued concurrently with o). Assume that p is still in v'. On one hand, liveness requirement **L2b** requires o to be executed (because the issuer of o is still in the view), on the other hand **S3** forbids the execution of o in v' (because the context of o changed) — these two requirements are incompatible.

Finally, note that a GMS that satisfies **S4** (and/or **S5**) has the following behavior: if the group ever becomes empty, it will stop evolving. This is because processes outside the group are not authorized to issue operations to modify the group. Thus, a GMS that satisfies **S4** should not be started with the "empty" initial view $v_0 = (0, \emptyset)$. Furthermore, if such a GMS ever generates a view with an empty group, it is up to the application that uses this GMS to restart the GMS (with some non-empty initial view) if it wishes to do so.

A summary of the specifications of the set and group membership problems is given in Appendix A.

4 On implementing SMS and GMS

4.1 Impossibility results

In this section we prove that in a purely asynchronous system subject to process crashes, it is impossible to solve the basic SMS problem or any of its variants that we considered in this paper. In particular, all

[6]By **S4**, p is necessarily in v.

the variants of the GMS problems that we defined here are also unsolvable in such a system.

It is important to note that these impossibility results are *not* because the membership service is trying to keep track of which processes are up or down (a task that is *trivially* impossible in an asynchronous system). In fact, the set maintained by a set membership service is not necessarily a set of processes (e.g., it could be the set of unsold seats on a flight), and so it may have nothing to do with the issue of which processes are currently up or down in the system. The impossibility of SMS (and all its variants) stems from another simple reason: it is because this service allows processes to *agree* on the various incarnations of a dynamically changing set — a form of consensus that cannot be achieved in a purely asynchronous system with failures.

One way to show this, is to just apply the impossibility result given in [4]: it is easy to see that all the GMS variants that we consider here satisfy the Weak Group Membership (WGM) specification defined in [4]. However, the impossibility of WGM was shown only for systems with $n \geq 4$ processes. For this reason, we prefer to give here a simple unified proof that holds for $n \geq 2$ (for all the variants of the SMS problem that we gave).

Theorem 1 *For all $n \geq 2$, the basic version of the SMS problem, and each one of its variants considered here, cannot be solved in asynchronous systems with process crashes. This holds even if we assume that at most one process may crash and all links are reliable.*

PROOF. Consider an asynchronous system with $n \geq 2$ processes $\Pi = \{p_1, p_2, \ldots, p_n\}$, and assume that at most one of them may crash. Suppose, for contradiction, that algorithm $\mathcal{A}$ solves (one of the variants of) the SMS problem in this system. In the following, processes in the system use $\mathcal{A}$ to maintain a set S whose elements are processes, i.e., $S \subseteq \Pi$.[7]. We consider two cases.

1. $n = 2$, i.e., $\Pi = \{p_1, p_2\}$. In this case the result follows from a standard partitioning argument. Partition Π into $\{p_1\}$ and $\{p_2\}$. Suppose that the initial view of both p_1 and p_2 is $v_0 = (0, \{p_1, p_2\})$. In run R1 of algorithm $\mathcal{A}$, p_1 is correct and executes $remove(p_2)$ in view v_0, while p_2 is dead. It is easy to see that, *for each variant of the SMS problem that we defined*, p_1 must eventually install the new view $v = (1, \{p_1\})$; say it does so by time t_1. Run R2 of algorithm $\mathcal{A}$ is symmetric: p_2 is correct and executes $remove(p_1)$ in view v_0, while p_1 is dead; p_2 must eventually install the new view $v' = (1, \{p_2\})$; say it does so by time t_2. In run R3 of $\mathcal{A}$, p_1 executes $remove(p_2)$, while p_2 executes $remove(p_1)$, in view v_0. All the messages between p_1 and p_2 are delayed to a time t greater than $\max(t_1, t_2)$. For process p_1, runs R3 and R1 are indistinguishable up to time t, and p_1 installs $v = (1, \{p_1\})$ by time t; for process p_2, runs R3 and R2 are indistinguishable up to time t, and so p_2 installs $v' = (1, \{p_2\})$ by time t. Thus, run R3 of algorithm $\mathcal{A}$ violates S_1, the view sequence agreement property of (every variant of) the SMS problem — a contradiction.

2. $n > 2$. In this case, the result can be obtained by reducing consensus to the SMS problem solved by algorithm $\mathcal{A}$, and then applying the well-known impossibility result of [15]. Processes in Π use the SMS algorithm $\mathcal{A}$ to solve binary consensus as follows. The initial view at every process is $v_0 = (0, \Pi)$. Every process with initial value 0 uses $\mathcal{A}$ to issue the operation $remove(p_1)$, while every process with initial value 1 uses $\mathcal{A}$ to issue the operation $remove(p_2)$. Then each process p waits to install a new view after v_0. Let v be the first view that p installs after v_0. If $v = (1, \Pi - \{p_1\})$ then p decides 0, and if $v = (1, \Pi - \{p_2\})$ then p decides 1. Note that some process q (either p_1 or p_2) does *not* belong to the new view v, and so, in some variants of the SMS problem (those with liveness requirement **L1a** instead of **L1**) process q is *not* required to install v. Thus, the reduction algorithm that we just described does not force q to decide. But since, $n > 2$ and at most one process may crash, Π has at least two correct processes, and so the set $\Pi - \{q\}$ contains at least one correct process other than q. That correct process is forced to install v (in every variant of the SMS problem that we considered), and hence to decide. Recall that the impossibility result of [15] holds even if only one process is required to decide. It is now easy to

[7]To show the impossibility of SMS, we could take $S \subseteq \{a_1, a_2, \ldots, a_n\}$, a set unrelated to processes. But to ensure that our proof also applies to all the GMS variants, we take the special case that $\mathcal{U} = \Pi$.

verify that all the properties of consensus considered in [15] are indeed satisfied by this reduction, independent of the variant of the SMS problem solved by algorithm $\mathcal{A}$. So $\mathcal{A}$ can be used to solve consensus — a contradiction to the impossibility result of [15]. $\square$

4.2 Possibility results

Despite the impossibility results described in the previous section, SMS and its variants can be implemented in practice. In fact, just as with consensus, there are approaches that can be used to effectively circumvent such impossibility results, e.g., the use of randomization [8] or unreliable failure detectors [5].

In this section, we first give simple implementations that assume a single non-faulty server, and then refer to the well-known *state-machine approach* [21] to replace this non-faulty server with a fault-tolerant replicated one. Our goal here is not to give the most efficient implementation of every possible SMS and GMS variant, or to give all the details. Rather, we want to show that it is possible to implement them incrementally, in a relatively simple way that does not mix the service provided with orthogonal implementation issues.

After showing how our GMS can be implemented, in Section 5 we explain how this GMS can be used for the special purpose of maintaining and agreeing on the set of processes that are currently deemed to be operational.

4.2.1 Using a non-faulty server

In this section, we assume the availability of a non-faulty server, and show that with this assumption it is very easy to implement all the variants of SMS and GMS that we considered in this paper.

The basic idea is quite simple: (a) the non-faulty server maintains the current value of the view; (b) processes wishing to issue an operation (to add or remove an element from the set) send their operations to the server; (c) the serve executes these operations sequentially, and after each operation it sends the resulting new view to processes; and (d) processes install every view they receive from the server.

This scheme, which implements the basic set membership service, is shown in more detail in Figure 1. It works for asynchronous systems where processes (the clients that issue operations to the set membership server) are subject to crashes or other benign failures, and the links to/from the non-faulty server are reliable and FIFO.[8] It is straightforward to verify that under these assumptions, the algorithm in Figure 1 indeed satisfies the complete specification of the basic SMS, i.e., requirements **S1**, **S2**, **L1** and **L2**.

All the SMS and GMS variants that we considered in this paper can be implemented by simple modifications to the basic algorithm in Figure 1. To enforce the additional safety requirement **S3** (an operation can be executed only in the context in which it was issued), a process that sends an operation to the server also includes its local view in the message, and when the server receives such an operation it executes it only if the current view (according to the server) is equal to the view associated with the operation.[9] This simple modification is shown in Figure 2. This algorithm implements the variant of the SMS requiring **S1**, **S2**, and **S3**, as well as **L1** and **L2a**.

Now consider the GMS variants (where $\mathcal{U} = \Pi$). To replace **L1** with the weaker requirement **L1a** (the correct processes in a view are the only ones required to install this view), the server sends each new view only to the current members of this view. To satisfy **S4**, a process p does not issue an operation unless it is a member of the group (according to its p's current view). To satisfy **S5**, the server executes an operation issued by a process p only if p is a member of the group (according to the server's current view). These three simple modifications to the basic algorithm are shown in Figure 3. It is easy to see that the algorithm given in this figure implements a GMS that satisfies the optional requirements **S4** and **S5** (in addition to **S1** and **S2**) as well as **L1a** and **L2b**.

[8]The FIFO assumption is just to simplify the presentation of the algorithm: it can be easily enforced using the sequence number that each operation and view carries.

[9]By the view sequence agreement property, each view is uniquely identified by its index, so a process can send the *index* of its local view rather than the whole view, and the server can compare this index with the index of its current view. We omit this obvious optimization from the code.

4.2.2 Using a fault-tolerant replicated server

In the above, we showed how SMS and GMS can be implemented assuming the existence of a single non-faulty server. This was done to better understand the services that we want to implement, and to decouple them from complex implementation concerns that may blur the simplicity of these services.

To remove the assumption of a single non-faulty server, we can use the well-known *state-machine approach* [21]. This method replaces the non-faulty server with a replicated server and uses standard techniques (based on consensus or atomic broadcast) to ensure that the replicated server works as if it were a single failure-free server.

For example, we could replace the single non-faulty server of Figures 1, 2, and 3, with a replicated server consisting of a static set of $2t+1$ processes, t of which may crash, and run consensus among these $2t+1$ processes to ensure that they behave consistently (i.e., they execute the same set of operations issued by the clients, in the same order). Even though consensus cannot be solved in purely asynchronous systems, it can be solved using randomization [8], unreliable failure detectors [5], or assuming some model of partial synchrony [12, 13, 11]. For example, in a recent work, Urbán *et. al.* implemented an extremely robust replicated server using the failure-detector based consensus algorithm of [5]. In this implementation, the replicated server continued to work even under the most severe workload, one that saturated the network [29, 28].

Note that the replicated server consisting of $2t+1$ processes need not be static, nor separate from the set of processes Π: these $2t+1$ processes can be in Π and dynamically change. In fact, the set of $2t+1$ processes that form the replicated server can dynamically agree on changing the membership of this set. They can do so by using the same consensus (or atomic broadcast) algorithm that they use to agree on the order of the clients operations. A detailed discussion of this technique, however, is beyond the scope of this paper, and therefore omitted.

As a final remark, it is worth emphasizing that failure detectors are *not* necessary to solve SMS and GMS in asynchronous systems with crashes. As we mentioned above, one can use randomization instead. Furthermore, if we do use failure detectors to solve SMS or GMS, it is only to enforce agreement among the $2t+1$ processes that form the fault-tolerant replicated server. But these failure detectors are *not* used to decide the *content* of the view (i.e., to decide whether to include a process in the current view).

5 Group membership and agreeing on the set of operational processes

In the literature, group membership is often defined and implemented as a service to maintain and agree on the content of one particular set, namely, *the set of processes that are currently deemed to be operational*. To provide this service, one must solve two orthogonal problems: (a) determining the set of processes that appear to be up in the system, and (b) agreeing on each successive view of this dynamically changing set.[10]

We consider the above service to be just one possible application of our GMS. In fact, to implement this service an application can use one of our GMS variants, together with some failure detector $\mathcal{D}$ that gives (reliable or unreliable) information about which processes have crashed: Processes use $\mathcal{D}$ to decide on when to issue operations to add or remove processes from the group, and they use a variant of our GMS to process these operations and agree on the current membership of the group. Since failure detector $\mathcal{D}$ ultimately determines which processes are in and out of the group, its Quality of Service [6] can be chosen by each application according to its particular needs (e.g., how fast the application wants a crashed process to be expelled from the group, and how costly it is for the application to erroneously remove a non-faulty process from the group).

[10]Note that both problems are unsolvable in purely asynchronous systems with failures. The first one is impossible for obvious reasons. The impossibility of the second one is non-trivial: as shown in Theorem 1, it is mostly based on the well-known result of [15].

An important remark is now in order. In existing implementations of the group membership service, the *same* failure detector is often used to solve both problems (a) and (b). This is not necessary, and in practice it is better to decouple the mechanisms that are used to solve these two orthogonal problems. In fact, as we mentioned earlier, failure detection is not even necessary for solving problem (b), e.g., one can use randomization instead. Moreover, even if we use failure detectors for solving both problems (a) and (b), these failure detectors are used for radically different purposes, and so they have different QoS requirements, as we now explain.

For problem (a), the failure detector is used to decide which processes are operational. In this case, the cost of a failure detector mistake (i.e., suspecting that a process p has crashed while it is actually up) is usually very high: the erroneously suspected process p is first removed from the group, and when this mistake is later recognized, p is reinserted in the group. This may involve expensive protocols and a costly state transfer done by the application. So an appropriate failure detector for problem (a) is likely to be one that favors higher accuracy at the expense of a slower detection time [6]. For these reasons, typical group membership protocols have failure detectors with large timeouts (on the order of tens of seconds).

For problem (b), however, the purpose of the failure detector is quite different: it is used by the replicated server to reach agreement on each new view of the group (it is *not* used to decide whether to add or remove processes from the group). In this case, the cost of a failure detector mistake is usually much smaller. For example, suppose that we implement a fault-tolerant group membership server using $2t+1$ processes, and that these processes reach agreement on each new view of the group by running the rotating-coordinator consensus algorithm in [5] or in [26, 24] that uses failure detector $\diamond S$[11]. We first note that with these consensus algorithms, a failure detector mistake can be harmful only if this mistake happens to be about the current coordinator. And even in this case, the cost of such a mistake is not very high: *no process is removed*, another coordinator just takes over. For this reason, an appropriate failure detector for problem (b) is likely to be one that favors a faster detection time at the cost of lower accuracy. A good timeout for such a failure detector may be orders of magnitude smaller than the one used for problem (a).

6 Related Work

The group membership problem has been extensively studied, and many specifications and implementations exist in the literature. The first papers to study the group membership problem were [10] (in the context of synchronous systems) and [25] (in the context of asynchronous systems). These two papers considered the *primary partition* version of the group membership problem, which requires that group views are totally ordered. In another version of group membership problem, called *partitionable*, group views are only partially ordered (e.g., [1, 3, 14]). A recent survey of group communication and group membership appeared in [7].

Despite their many differences, most (if not all) papers on the subject consider the problem of keeping track of who is up, down, or partitioned away in the system, to be an integral part of group membership. This is evident from the following quotes (taken from papers whose publication dates span a decade):

> [10]: *The specification requires crashed processes to be excluded within a given delay D.*
>
> [25]: *The specification requires to react to failure detection.*
>
> [23]: *Membership is determined by whether or not a processor has failed.*
>
> [19]: *Membership deals with the problem of keeping track of which processes are faulty and which are fault-free.*
>
> [20]: *A membership service is* live *with respect to failure if it is guaranteed to report that type of change eventually.*
>
> [9]: *The group membership service implements a kind of failure detector that allows surviving processes to agree on which processors have failed.*

[11] Roughly speaking, with $\diamond S$ every process that crashes is eventually suspected, and there is a time after which some non-faulty process is not suspected.

> [7]: *The task of a membership service is to maintain a list of the currently active and connected processes.*
>
> [3]: *In the absence of partitionings, every correct process should install the same view and this view should include exactly those members that have not crashed.*

In contrast to the above, our specification of set and group membership does not relate the content of the set/group being maintained to (reliable or unreliable) information about processes failures or network partitions. Indeed, our specification is not concerned by the reasons why elements are added or removed from the maintained set/group.

It is widely recognized that the task of giving a precise specification of group membership is difficult, and in fact many of the existing specifications are either difficult to understand or problematic [2]. It has been claimed that the source of this difficulty is the impossibility of reaching consensus, i.e., process agreement, in asynchronous environment with failures. For example:

> [16]: *Most of the difficulties in building a specification for the Group Membership Problem arise from the impossibilities result in [4] and [15].*

But this reason is not entirely valid: the impossibility of consensus and atomic broadcast in asynchronous systems with failures was never an obstacle to the simple and precise specifications of these problems.

We believe that the difficulty in specifying group membership originated from the mixing of two orthogonal concerns, namely, *determining* who is up or down, and *agreeing* on this set. As we mentioned before, it may be better to decouple the mechanisms used to solve them. This is usually not the case, e.g., in [22], an oracle is used to decide whether to exclude a process from the group and the same oracle is also used to achieve consensus on each view, and in [25] the same failure detection mechanism is used to determine who is down and to agree on each view. Finally, note that the decoupling of the two mechanisms implicitly appeared in [17] where a consensus service is used to solve agreement problems.

7 Conclusion

Our treatment of group membership clarifies the problem by separating several orthogonal concerns that are often mixed in existing specifications and/or implementations of group membership services.

First, we separate the problem of agreeing on the content of a dynamically changing set (the SMS problem) from the issue of the domain of that set. In the special case of group membership (the GMS problem), this set contains processes.

Second, in our formulation of GMS, we decouple the issue of *why* processes are added or removed from the group — a concern that does not even appear in our specification of GMS — from *how* they are added or removed. Existing group membership services usually tie the removal or addition of a process to some (reliable or unreliable) information about this process failure or recovery.

Third, in our GMS implementation outline, we decouple the service from the issue of which processes actually provide the service and what is the degree of fault-tolerance provided. Specifically, the $2t + 1$ processes that provide the GMS service are not required to be current members of the group. Moreover, the size of the current group may be smaller or larger than $2t+1$. In contrast, in most existing group membership services, the service is provided by the processes in the current group, and the degree of fault-tolerance is proportional to the current size of the group.

Fourth, our treatment of group membership decouples two types of failures: (a) the failure of a process that belongs to the group (which may cause a request to remove this process from the group), and (b) the failure of a process that provides the membership service (which may hinder reaching agreement on the new value of the group).

Finally, our treatment of group membership also separates the two mechanisms needed to (a) determine and (b) agree on the set of processes that are deemed to be operational: a failure detector can be used for (a), but is not necessary for (b) — e.g., randomization can be used instead. Moreover, as we explained in Section 5, even if we decide to use failure detectors for both (a) and (b), they should be decoupled: each one solves a different problem and has different requirements in practice.

Acknowledgments. We would like to thank Marcos K. Aguilera, Carole Delporte-Gallet, Jonathan Eisler, Hugues Fauconnier, and Arnas Kupšys for their insightful comments on an earlier version of the paper.

```
CODE FOR SET MEMBERSHIP SERVER:

on initialization:
    $S := S_0$ { initial value of the set, such that $S_0 \subseteq \mathcal{U}$ }
    $i := 0$ { initial index of the view }
    $v := (i, S)$ { initial view of the set }
    send ($v$) to every $p \in \Pi$ { send initial view to all }
    start task 0

task 0: { To process requests to modify the set and generate new views accordingly }
    upon receive $(p, k, op(e))$ { $op(e)$ is an operation to add or remove $e$ to/from the set }
        if $op = add$ then $S := S \cup \{e\}$ else $S := S \setminus \{e\}$ {execute $op(e)$ on the current value of the set }
        $i := i + 1$ { increment index of the view }
        $v := (i, S)$ { generate new view of the set }
        send ($v$) to every $p \in \Pi$ { send new view to all }

CODE FOR EACH PROCESS $p \in \Pi$:

on initialization:
    $k := 0$ { number of operations issued by $p$ }
    $localView := \perp$ { $p$'s local view of the set }
    start tasks 1 and 2

task 1: { To issue requests to modify the set }
    repeat forever
        if $p$ wants to issue an operation $op(e)$ then { $op \in \{add, remove\}$ and $e \in \mathcal{U}$ }
            $k := k + 1$
            send $(p, k, op(e))$ to the set membership server

task 2: { To install new views of the set }
    upon receive $(v)$ { a new view generated by the set membership server }
        $localView := v$ { $p$ installs the new view }
```

Figure 1: Basic SMS implementation (using a non-faulty server and FIFO reliable links).

```
CODE FOR SET MEMBERSHIP SERVER:

on initialization:
 $S := S_0$ { initial value of the set, such that $S_0 \subseteq \mathcal{U}$ }
 $i := 0$ { initial index of the view }
 $v := (i, S)$ { initial view of the set }
 **send** ($v$) **to** every $p \in \Pi$ { send initial view to all }
 **start** task 0

task 0: { *To process requests to modify the set and generate new views accordingly* }
 **upon receive** $[localView, (p, k, op(e))]$ { operation $op(e)$ was issued in $localView$ }
     **if** $v = localView$ **then** { if the current view is equal to $localView$ }
         **if** $op = add$ **then** $S := S \cup \{e\}$ **else** $S := S \setminus \{e\}$ {execute $op(e)$ }
         $i := i + 1$ { increment index of the view }
         $v := (i, S)$ { generate new view of the set }
         **send** ($v$) **to** every $p \in \Pi$ { send new view to all }

CODE FOR EACH PROCESS $p \in \Pi$:

on initialization:
 $k := 0$ { number of operations issued by $p$ }
 $localView := \perp$ { $p$'s local view of the set }
 **start** tasks 1 and 2

task 1: { *To issue requests to modify the set* }
 **repeat forever**
     **if** $p$ wants to issue an operation $op(e)$ **then** { $op \in \{add, remove\}$ and $e \in \mathcal{U}$ }
         $k := k + 1$
         **send** $[localView, (p, k, op(e))]$ **to** the set membership server

task 2: { *To install new views of the set* }
 **upon receive** $(v)$ { a new view generated by the set membership server }
     $localView := v$ { $p$ installs the new view }
```

Figure 2: Basic SMS implementation satisfying optional requirement **S3**.

```
CODE FOR GROUP MEMBERSHIP SERVER:

on initialization:
  $S := S_0$ { initial value of the group, such that $S_0 \subseteq \Pi$ }
  $i := 0$ { initial index of the view }
  $v := (i, S)$ { initial view of the group }
  send ($v$) to every $p \in S$ { send initial view to every process in the initial group }
  start task 0

task 0: { To process requests to modify the group and generate new views accordingly }
  upon receive $(p, k, op(e))$ { operation $op(e)$ was issued by $p$ }
      if $p \in S$ then { if $p$ is a member of the current group then }
          if $op = add$ then $S := S \cup \{e\}$ else $S := S \setminus \{e\}$ {execute operation requested by $p$ }
          $i := i + 1$ { increment index of the view }
          $v := (i, S)$ { generate new view of the group }
          send ($v$) to every $p \in S$ { send new view to every process in the current group }

CODE FOR EACH PROCESS $p \in \Pi$:

on initialization:
  $k := 0$ { number of operations issued by $p$ }
  $localView := \perp$ { $p$'s local view of the group }
  start tasks 1 and 2

task 1: { To issue requests to modify the group }
  repeat forever
      if $p$ is in $localView$ and $p$ wants to issue an operation $op(e)$ then { $op \in \{add, remove\}$ and $e \in \Pi$ }
          $k := k + 1$
          send $(p, k, op(e))$ to the group membership server

task 2: { To install new views of the group }
  upon receive $(v)$ { a new view generated by the group membership server }
      $localView := v$ { $p$ installs the new view }
```

Figure 3: Implementation of GMS with optional requirements **L1a**, **S4** and **S5** (here $\mathcal{U} = \Pi$).

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.

[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] O. Babaoglu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Trans. on Software Engineering*, 27(4):308–336, 2001.

[4] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, Pennsylvania, USA, May 1996.

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

[6] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 5(51):561–580, May 2002.

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

[8] B. Chor and C. Dwork. Randomization in Byzantine Agreement. In S. Micali, editor, *Advances in Computing Research, Randomness in Computation*, volume 5, pages 443–497. JAI Press, 1989.

[9] M. Clegg and K. Marzullo. A Low-Cost Group Membership Protocol for a Hard Real-Time Distributed System. In *Proc. 18th IEEE Real-Time Systems Symposiun (RTSS'97)*, pages 90–98, December 1997.

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

[11] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel & Distributed Systems*, 10(6):642–657, June 1999.

[12] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of ACM*, 34(1):77–97, January 1987.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.

[14] A. Fekete, N. Lynch, and A. A. Shvartsman. Specifying and Using a Group Communication Service. *ACM Trans. on Computer Systems*, 19(2):171–216, May 2001.

[15] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.

[16] M. Franceschetti and J. Bruck. A Group Membership Algorithm with a Practical Specification. *IEEE Transactions on Parallel & Distributed Systems*, 12(11):1190–1200, November 2001.

[17] R. Guerraoui and A. Schiper. The Generic Consensus Service. *IEEE Trans. on Software Engineering*, 27(1):29–41, January 2001.

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

[19] M. A. Hiltunen. Membership and System Diagnosis. In *14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, pages 208–217, Bad Neuenahr, Germany, September 1995.

[20] M. A. Hiltunen and R. D. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[21] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 1978.

[22] K. Lin and V. Hadzilacos. Asynchronous Group Membership Service. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*, pages 79–93. Springer Verlag, LNCS 1693, September 1999.

[23] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Processor Membership in Asynchronous Distributed Systems. *IEEE Transactions on Parallel & Distributed Systems*, 5(5):459–473, May 1994.

[24] A. Mostefaoui and M. Raynal. Solving Consensus using Chandra-Toueg's Unreliable Failure Detectors: A Synthetic Approach. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*. Springer Verlag, LNCS 1693, September 1999.

[25] 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.

[26] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[27] F. B. Schneider. Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[28] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2003. Number 2824.

[29] P. Urbán, X. Défago, and A. Schiper. 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-20)*, pages 190–193, New Orleans, USA, October 2001.

A Appendix: Specification summary

Processes in Π maintain a set of elements from an arbitrary universe $\mathcal{U}$. They issue operations to add and remove an element to/from that set, and install views of the maintained set (a view consists of a set value and an index). The set membership service executes these operations sequentially, and notifies processes of each new view of the set that it generates.

A local history H_p of a process p is a sequence of operations (issued by p) and views (installed by p). A global history of the set membership service H is an alternating sequence of operations (executed by the service) and views (generated by the service): H starts with an initial view of the set, and each operation in H changes the view that precedes it into the view that follows it, in the natural way.

A.1 The set membership service (SMS)

A set of local histories (one H_p for each process p in Π) satisfies the basic SMS specification if there is a global history of the membership service H with the following properties:

S1: *View Sequence Agreement:* For every process p, the sequence of views in the local history H_p of p is a subsequence of the sequence of views in the global history H of the membership service.

S2: *Integrity:* Every operation executed by the membership service is requested by some process, and it is executed only once: $\forall o \in H \Rightarrow \exists p \in \Pi : o \in H_p$ and $\forall o, o' \in H : o \neq o'$

L1: *View installation:* Every view generated by the membership service is installed by every correct process: $\forall p \in C, \forall v \in H$: $v \in H_p$

L2: *Operation execution:* Every operation issued by a correct process is executed by the membership service: $\forall p \in C, \forall o \in H_p$: $o \in H$

The following additional requirement to the basic SMS is optional:

S3: *Same context:* An operation o is executed in view v only if o was issued in v.

For applications requiring the optional safety property **S3**, liveness requirement **L2** must be weakened to:

L2a: *Operation execution:* If a correct process p issues an operation o in a view v, then v is not the final view of the global history H of the membership service.

A.2 The group membership service (GMS)

The GMS specification is identical to the SMS specification where $\mathcal{U} = \Pi$. Some GMS variants may be desirable, and we list some of them here (they make sense only because $\mathcal{U} = \Pi$). We first note that liveness requirement **L1** may be weakened to:

L1a: *View installation:* A correct process must install every view it belongs to: $\forall p \in C, \forall v \in H$: $p \in v \Rightarrow v \in H_p$.

Some of the following additional safety requirements may be also desirable:

S4: *Authority to request a view change:* A process p can issue an operation in a view v only if p is in v.

S5: *Authority to execute a view change:* An operation o issued by a process p is executed in a view v only if p is in v.

If an application requires **S5**, it should also require **S4**. Moreover, if **S4** and **S5** are required, we must weaken liveness requirement **L2** to **L2a**, or to **L2b** given below:

L2b: *Operation execution:* If a correct process p issues an operation o in a view v, then either o is executed by the membership service, i.e., $o \in H$, or there exists a view v' after v in H such that p is not in v'.