The Generic Consensus Service

Rachid Guerraoui, *Member*, *IEEE*, and André Schiper, *Member*, *IEEE*

Abstract—This paper describes a modular approach for the construction of fault-tolerant agreement protocols. The approach is based on a generic consensus service. Fault-tolerant agreement protocols are built using a client-server interaction, where the clients are the processes that must solve the agreement problem and the servers implement the consensus service. This service is accessed through *a generic consensus filter*, customized for each specific agreement problem. We illustrate our approach on the construction of various fault-tolerant agreement protocols, such as nonblocking atomic commitment, group membership, view synchronous communication, and total order multicast. Through a systematic reduction to consensus, we provide a simple way to solve agreement problems. In addition to its modularity, our approach enables efficient implementations of agreement protocols and precise characterization of the assumptions underlying their liveness and safety properties.

Index Terms—Asynchronous distributed systems, consensus, fault-tolerant agreement protocols, failure detectors, modularity, atomic commitment, group membership, view synchrony, total order multicast.

1 Introduction

Agreement protocols such as atomic commitment, group membership, and total order broadcast or multicast are at the heart of many distributed applications, including transactional and time critical applications. Based on some recent theoretical results on solving agreement problems in distributed systems [10], [9], [15], [31], we present in this paper a unified framework to develop fault-tolerant agreement protocols in a modular, correct, and efficient way.

In our framework, we suggest the use of a *generic consensus service* to build fault-tolerant agreement protocols. The *consensus service* is implemented by a set of *consensus server* processes and the number of these processes depends on the desired resilience of the service. We introduce the generic notion of *consensus filter* to customize the consensus service for specific agreement protocols. Building a fault-tolerant agreement protocol boils down to a client-server interaction where 1) the clients are the processes that have to solve the agreement problem and 2) the servers implement the consensus service, accessed through the consensus filter. The client-server interaction differs, however, from the usual client-server interaction scheme: Here, we have an n_c-n_s interaction (n_c clients, n_s servers), with $n_c > 1$, $n_s > 1$, rather than the usual 1-1 or 1-n_s interaction.

We show how various agreement protocols are built simply by adapting the consensus filter. The modularity of our infrastructure enables us to derive correctness properties of agreement protocols from the properties of the consensus service and leads to effective optimizations that trade resilience with efficiency. Behind our approach, we argue that consensus is not only a fundamental paradigm in theoretical distributed computing [23], [33], but also a useful building block for practical distributed systems.

The paper is structured as follows: Section 2 recalls some background on the development of distributed services and distributed agreement protocols. Section 3 presents our system model and recalls some results about the consensus problem. Section 4 gives an overview of our generic consensus service. Section 5 details how nonblocking atomic commitment protocols can be constructed using our consensus service. Section 6 illustrates the use of the consensus service in building protocols for group membership, while Section 7 extends the example of group membership to view synchronous communication. Section 8 considers atomic broadcast and atomic multicast protocols. Section 9 presents a cost analysis and discusses efficiency issues. Section 10 points out some possible uses and generalizations of our framework.

2 Background

General services used to build distributed applications or to implement higher level distributed services have become common in distributed systems. Examples are numerous: file servers, time servers, name servers, authentication servers, etc. However, there have been very few proposals of services specifically dedicated to the construction of fault-tolerant agreement protocols such as atomic commitment, total order broadcast, etc. Usually, these protocols are considered separately and do not rely on a common infrastructure.

A notable exception is the *group membership service* [28], which was used to implement various total order broadcast protocols [8], [12], [13], [1]. However, the group membership problem (solved by the membership service) is just one example of an *agreement* problem that arises in distributed systems. In fact, all agreement problems (atomic commitment, total order broadcast, group membership) are related to the abstract *consensus* problem [10], [36] and, thus, are subject, in asynchronous systems, to

• *The authors are with the Département de Systèmes de Communication, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland. E-mail: {Rachid.Guerraoui, Andre.Schiper}@epfl.ch.*

Manuscript received 23 June 1998; revised 7 June 1999; accepted 22 May 2000.
Recommended for acceptance by F. Schneider.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107059.

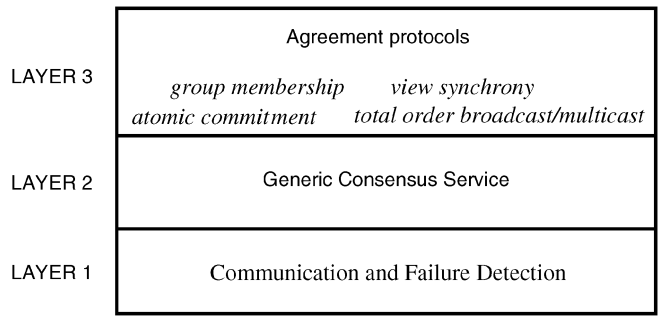

Fig. 1. The architecture model.

the Fischer-Lynch-Paterson impossibility result [14], [9], [11].[1] By defining a unified consensus-based framework for solving various agreement problems, we provide a systematic way to reuse the results of Chandra and Toueg on failure detectors and consensus in proving the correctness of agreement protocols.

Our work can be viewed as continuation of the work of Schneider [33], who suggested the use of consensus as a central paradigm for reliable distributed programming. We go a step further by describing a generic and systematic way to transform various agreement problems into consensus. Our transformation generalizes the solutions for problems like group membership and view synchronous communication presented in [25], [24] and leads to highlight their common characteristics with problems usually considered separately like nonblocking atomic commitment.

3 System Architecture and Model

Our system architecture is depicted in Fig. 1. We describe below the process model and the communication layer, then we recall the failure detection abstraction (layer 1) and the definition of the consensus problem. The generic consensus layer (layer 2) is described in Section 4. Examples of using the generic consensus service to solve various agreement problems (layer 3) are given in Sections 5, 6, 7, and 8.

3.1 Processes

We consider a distributed system composed of processes denoted by $p_1, p_2, \ldots, p_i, \ldots$. The processes are completely connected through a set of channels. Every process can send a message, receive a message, and perform a local computation (e.g., modify its state or consult its local failure detector module). We do not make any assumption on process relative speeds, but we assume a crash-stop failure model: A process fails by crashing and, after it does so, the process never executes any action. We do not consider for instance Byzantine failures, i.e., we assume that processes do not behave maliciously.[2] A correct process is a process that does not crash and a process that crashes is said to be faulty.

1. We recall the definition of the consensus problem later in the paper. The Fischer-Lynch-Paterson impossibility result states that there is no deterministic algorithm that solves consensus in an asynchronous system when one process can crash [14].

2. In Section 10, we will discuss the generalization of our framework to other fault models.

3.2 Communication Primitives

We consider an asynchronous communication model, i.e., we do not assume any bound on the time it takes for a message to be transmitted from the sender to a destination process. We assume, however, that the channels are *quasi–reliable* [2], which ensures the following property:

- A message sent by a process p_i to a process p_j is eventually received by p_j if p_i and p_j are both correct.

Quasi-reliable channels can be implemented over fair lossy channels [5] by retransmitting messages. They do not exclude the possibility of temporary link failures (temporary partitions). A quasi-reliable channel is weaker than a *reliable* channel [5] which ensures that a message m sent by p_i to p_j is eventually received by p_j if p_j is correct, i.e., the latter definition does not require p_i to be correct. This means that reliable channels do not lose messages, whereas quasi-reliable reliable channels may lose messages and, hence, more adequately model real communication links.

For the modularity of our construction, we introduce the following communication primitives, which can be built using quasi-reliable channels.

- $Rmulticast(m)$ to $Dst(m)$: reliable multicast of m to the set of processes $Dst(m)$. This primitive ensures that, if the sender is correct, or if one correct process p_j in $Dst(m)$ receives m, then every correct process in $Dst(m)$ eventually receives m.
- $multisend(m)$ to $Dst(m)$: equivalent to **for every** $p_j \in Dst(m)$, $send(m)$ to p_j.

The primitive *multisend* is introduced as a convenient notation, whereas *Rmulticast* provides a stronger semantics. To understand the difference, consider 1) $Rmulticast(m)$ to $Dst(m)$ and 2) $multisend(m)$ to $Dst(m)$, both performed by some process p_i. If p_i crashes, then $multisend(m)$ to $Dst(m)$ can lead to partial reception of m: Some correct process p_j in $Dst(m)$ might receive m and some other correct process p_k in $Dst(m)$ might never receive m. Such a situation does not occur with *Rmulticast*. A *multisend* is implemented simply by sending multiple messages, whereas an *Rmulticast* requires message retransmission by a destination process (see [10] for more details on implementation of reliable multicast).

3.3 Failure Detectors

Failure detectors have been formally introduced in [10], [9] for solving the consensus problem. A failure detector can be viewed as a distributed oracle. Each process p_i has access to a local failure detector module D_i. This module maintains a list of processes that it currently *suspects* to have crashed.

In this paper, as we consider consensus as a black box,[3] we do not recall the formal characterization of failure detectors for solving consensus (see [10]). In our scheme, failure detectors are also used outside of the consensus box. The properties they are required to satisfy depend, however, on the agreement problem we are trying to solve. Whenever we make use of failure detectors in the paper (outside of the consensus box), we will specify the properties that are needed.

3. We come back to this in Section 9.

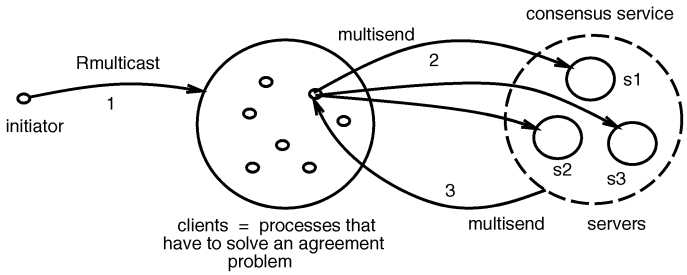

Fig. 2. Interaction from a client's point of view.

3.4 Consensus

The consensus problem is defined over a set of processes. Every process p_i in this set starts with an initial value v_i and the processes have to decide on a common value v. Consensus is defined by the following three properties [10]:

Uniform Agreement. No two processes decide differently.

Termination. Every correct process eventually decides.

Uniform Validity. If a process decides v, then v is the initial value of some process.

The definition considered above specifies the *uniform* version of the consensus problem. It requires agreement and validity properties to be satisfied even by faulty processes. We do not discuss here specific algorithms that solve consensus: We just assume the existence of such algorithms. The reader interested in learning more about solving consensus in an asynchronous system model augmented with failure detectors can consult [10], [31].

4 The Consensus Framework

In this section, we give an abstract view of our consensus service-based framework. Our description is abstract in the sense that we do not consider here any specific agreement problem. Examples of solving agreement problems in our framework are given in Sections 5, 6, 7, and 8.

4.1 The Roles: Overview

Our framework distinguishes the following process roles:

- The "*initiator*" of an agreement problem.
- The processes that have to solve an agreement problem. These processes play the role of "*clients*" of the consensus service.
- The processes that solve consensus. These processes are the "*server*" processes. We assume that at least one of these server processes is correct.

The different roles can overlap: An initiator process can also be a client process and the role of the server processes can be played by all or by a subset of the client processes. In practice, this would be the typical scenario (we will come back to this in Section 9). We will also see that, depending on the agreement problem, the initiator can be either a client process or distinct from the client processes. However, for simplicity of presentation, we will mainly consider the case where the initiator, the client processes, and the server processes are distinct. We will denote the server processes by $s_1, s_2, \ldots, s_m$. The number m of these processes depends on the desired resilience of the service.

The interaction between the initiator, the clients, and the consensus servers is based on the *Rmulticast* and the *multisend* communication primitives defined in the previous section. A basic interaction has three phases (Fig. 2):

1. An *initiator* process starts by multicasting a message to the set of client processes, using the *Rmulticast* primitive (Arrow 1).
2. Clients invoke the consensus service using a *multisend* primitive (Arrow 2).
3. The consensus service sends a decision back to the clients using a *multisend* primitive (Arrow 3).

We will see throughout the paper that many agreement problems can be solved through the above three phase interaction. In most of the cases (Sections 5, 6, and 7), there is a 1-1 correspondence between one instance of an agreement problem and one instance of consensus. We will also briefly mention in Section 8 the case of a n-1 correspondence, where several instances of an agreement problem (total order broadcast) correspond to one single instance of consensus.

4.2 The Roles: Description

The initiator. The invocation of the consensus service is started by an *initiator* process which reliably multicasts ($Rmulticast$ primitive) the message $(cid, data, clients)$ to the set $clients$ (Arrow 1 in Fig. 2; and Fig. 3). The parameter cid ($consensus\ id$) uniquely identifies the interaction with the consensus service, $data$ contains some problem specific information, and the parameter $clients$ is the set of processes that have to solve an agreement problem.

The clients. Upon reception of the message $(cid, data, clients)$ multicast by the initiator, every process $p_i \in clients$ computes $data_i'$ (which contains problem specific information), multisends the message $(cid, data_i', clients)$ to the consensus service, and waits for the decision of the consensus service (Fig. 4).

The servers. The interaction between the clients processes and the consensus service is illustrated from the point of view of a server process in Figs. 5 and 6. The

1	compute $data$, the set $clients$, and get a consensus identifier cid ;
2	$Rmulticast(cid, data, clients)$ to $clients$;

Fig. 3. Initiator's algorithm.

```
1  upon reception of (cid, data, clients) by a client p_i:
2     compute data'_i ;
3     multisend(cid, data'_i, clients)
            to the members of the consensus service ;
4     wait reception of (cid, decision)
            from the consensus service ;
```

Fig. 4. Algorithm of a client p_i.

```
1  wait reception of (cid, data'_i, clients) from clients
       until CallInitValue(cid) ;
2  v_j ← InitValue(
       {data'_i | message (cid, data'_i, clients) received} ) ;
3  decision ← consensus(cid, v_j) ;
4  multisend(cid, decision) to clients ;
```

Fig. 5. Algorithm of a server s_j.

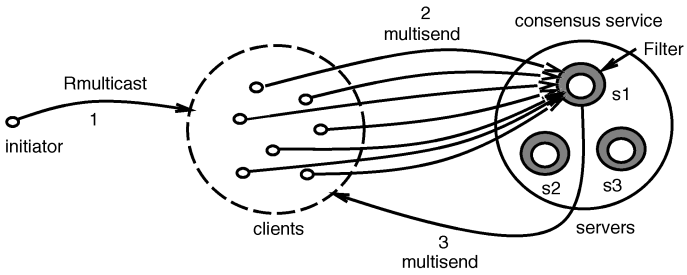

Fig. 6. Invocation-reply from the point of view of server s_1.

genericity of the consensus service is obtained thanks to the notion of *"consensus filter,"* depicted in Fig. 6 as a shaded ring (arrows *to* and *from* s_2 and s_3 have not been drawn to keep the figure simple). The consensus filter allows one to tailor the consensus service to specific agreement problems. The filter transforms the messages received by a server process s_j into a consensus initial value v_j for s_j.

The consensus filter. A consensus filter, attached to every server process s_j, is defined by two parameters: 1) a predicate $CallInitValue$ and 2) a function $InitValue$ (Fig. 5). The predicate $CallInitValue$ defines the condition under which the function $InitValue$ can be called and the consensus protocol started. It is a *stable* predicate, i.e., if $CallInitValue$ is true at a time t, it is true for any time $t' > t$. As soon as the predicate $CallInitValue$ returns $true$ (line 1, Fig. 5), the function $InitValue$ is called (line 2, Fig. 5). $InitValue$ returns the initial value for the consensus. In Fig. 5 (line 3), the consensus protocol is represented as a function $consensus(cid, v_j)$. The consensus decision, once known, is multisent to the set $clients$ (line 4, Fig. 5).

We say that a consensus filter is *live* at a correct server process s_j if the predicate $CallInitValue$ eventually becomes $true$ and the function $InitValue$ eventually returns some value at s_j.

4.3 Correctness

Here, we present two properties that are ensured by our generic framework and from which we derive the correctness proofs of agreement protocols (see Sections 5, 6, 7, and 8).

CS-Agreement. No two client processes receive two different decision messages $(cid, decision)$.

CS-Termination. If the consensus filter is live, then the decision message $(cid, decision)$ is eventually received by every client.

The CS-Agreement (Consensus Service Agreement) property directly follows from consensus (Section 3.4). Given a live consensus filter, the CS-Termination (Consensus Service Termination) property follows from the termination property of consensus (Section 3.4), the assumption that at least one server is correct, and the assumption of quasi-reliable channels.

5 Nonblocking Atomic Commitment

Throughout this section, we show how a modular nonblocking atomic commitment protocol can be built using our consensus service together with an adequate filter. We first recall the problem, then we show how it can be solved in our consensus framework. In Section 9, we will compare the performance of our protocol with that of the well-known nonblocking atomic protocols proposed by Skeen, namely the Three Phase Commit protocol [34].

5.1 Background

A transaction originates at a process called the *transaction manager*, which issues read and write operations to *data manager* processes [6]. At the end of the transaction, the data managers must solve an *Atomic Commitment* problem in order to decide on the *commit* or *abort* outcome of the transaction. We consider here the *"Nonblocking" Atomic Commitment* problem (NB-AC for short), where correct data managers eventually decide despite failures [34]. The

outcome of the transaction depends on votes from the data managers. A data manager votes *yes* to indicate that it is able to make the temporary writes permanent and votes *no* otherwise. If the outcome of the NB-AC protocol is *commit*, then all the temporary writes are made permanent; if the outcome is *abort*, then all temporary writes are ignored. The NB-AC problem is defined by the following properties:

NB-AC-Agreement. No two data managers ever decide differently.

NB-AC-Termination. Every correct data manager eventually decides.

NB-AC-Validity. The decision must be *abort* if one data manager votes *no* and the decision must be *commit* if all data managers vote *yes* and no data manager crashes.

5.2 NB-AC Based on a Consensus Service

In the following, we show how an NB-AC protocol is derived from our consensus service framework (Section 4): The data managers are the clients of the consensus service. As proven in [15], NB-AC, unlike consensus, cannot be solved with unreliable failure detectors. Our NB-AC solution assumes a perfect failure detector that satisfies *strong completeness* (a crashed process is eventually suspected by every correct process) and *strong accuracy* (no process is suspected unless it has crashed) [10]. We assume a perfect failure detector among all the processes in the system, i.e., clients (data managers) and consensus servers.[4]

We first focus on the NB-AC-Agreement and NB-AC-Termination properties. Then, we describe a consensus filter adapted to the NB-AC-Validity property.

5.2.1 NB-AC: Agreement and Termination

The transaction manager is the initiator of an interaction with the consensus service. Arrow 1 in Fig. 2 represents the message (*tid*, *vote-request*, *data-managers*) sent by the transaction manager to the data managers at the commitment of the transaction: The transaction identifier tid is the consensus id, the generic $data$ field is instantiated as *vote-request*, and *data-managers* is the set of data managers accessed by the transaction. The $data_i'$ value (Fig. 4) is the *yes/no* vote of the data manager p_i and the decision awaited from the consensus service is either *commit* or *abort*.

NB-AC-Agreement follows directly from the CS-Agreement property of the consensus service (Section 4.3) and, if we assume that the consensus filter is live (see below), NB-AC-Termination follows from the CS-Termination property of the consensus service (Section 4.3).

5.2.2 NB-AC: Validity

NB-AC consensus filter. The consensus filter, given below, tailors the consensus service to the NB-AC-Validity property. The *NB-AC-CallInitValue* predicate is defined in such a way that the votes from all nonsuspected clients (data managers) are received by the servers. In other words, *NB-AC-CallInitValue* at a server s_j returns $true$ as soon as, for every client process p_i, either 1) the message $(cid, vote_i, clients)$ from p_i has been received by s_j or 2) p_i is suspected by s_j.

The function *NB-AC-InitValue* at a server s_j returns *commit* if and only if a yes vote has been received by s_j from every process in $clients$. Otherwise, if any process in $clients$ has been suspected, or has voted no, then the function *NB-AC-InitValue* returns *abort* (note that the *commit/abort* values returned by the function *NB-AC-InitValue* are here initial values for the consensus service and not yet the decision of the consensus).

The consensus filter for a NB-AC protocol is thus specified as follows at every server process s_j:

Predicate *NB-AC-CallInitValue(cid)* :
  if [for every process $p_i \in clients$:
    s_j has received $(cid, vote_i, clients)$ from p_i
    or s_j suspects p_i]
  then return $true$ **else return** $false$.

Function *NB-AC-InitValue(*$dataReceived_j$*)* :
  if [for every process $p_i \in clients$:
    $(cid, vote_i, clients) \in dataReceived_j$
    and $vote_i = yes$]
  then return $commit$ **else return** $abort$.[5]

We show now that the NB-AC consensus filter is live (property needed to prove the NB-AC-Termination) and ensures the NB-AC-Validity property.

Liveness of the NB-AC consensus filter. If the initiator is correct or some correct process $p_i \in clients$ has received the message (*cid*, *vote-request*, *clients*) sent by the initiator, then, by the properties of the reliable multicast, every correct client receives the message (*cid*, *vote-request*, *clients*) and multisends the message $(cid, vote_i, clients)$ to the members of the consensus service. For every client p_i, there are two cases to consider: 1) p_i is correct or 2) p_i crashes. In case 1, p_i's message $(cid, vote_i, clients)$ is eventually received by all correct servers. In case 2, p_i is eventually suspected by all correct servers (ensured by the strong completeness property of our failure detector). In both cases, at every server process, the predicate *CallInitValue* eventually returns $true$ and the function *NB-AC-InitValue* eventually returns some value: The consensus filter of NB-AC is thus live.

NB-AC-Validity is satisfied. The NB-AC-Validity property states that 1) the decision must be $commit$ if all data managers vote yes and no data manager crashes and 2) the decision must be $abort$ if any data manager votes no. Consider 1): If no client crashes, because of the assumption of a perfect failure detector, no client is ever suspected (by any process). In this case, *CallInitValue* waits for the vote of every process in $clients$. If all the votes are yes, then *InitValue* ensures that every server starts consensus with the initial value *commit*. By the validity property of consensus

4. Obviously, if we consider a solution where the data managers also play the role of the consensus servers, a perfect failure detector would only be needed among the data managers. Furthermore, with a weaker assumption about the failure detector, one can still use the very same solution to solve a weaker problem, called *Weak* NB-AC [15], where the data managers are allowed to decide *abort* in case of a failure suspicion (even if this suspicion is false).

5. Notice that, depending on the failure suspicions, one server s_j might start the consensus with the initial value *commit*, while another server s_k starts the consensus with the initial value *abort*. In this case, the two possible outcomes of the consensus service, i.e., *commit* and *abort*, both satisfy the specification given in Section 5.1.

(Section 3.4), the decision can only be *commit*. Now consider 2): If one data manager votes no or one data manager crashes (i.e., is suspected), then *CallInitValue* ensures that every correct server starts consensus with the initial value *abort*. By the validity property of consensus, the decision can only be *abort*.

5.3 Variations on the Consensus Filter

The definition of atomic commitment we have considered so far (Section 5.1) is the classical definition usually given in the literature. According to this definition, the $commit$ decision requires a yes vote from all the data managers involved in the transaction (NB-AC-Validity property). This requirement is too strong in specific situations where the data managers maintain replicated data. In this case, one might require a weaker NB-AC-Validity property where $commit$ can be decided when, for every logical $data_i$ that is replicated, a majority of data managers for $data_i$ vote yes.[6]

We show in the following how to solve this variation of the NB-AC problem, defined by the classical NB-AC-Agreement and NB-AC-Termination properties and the modified NB-AC-Validity property. We consider first the case of one single replicated data and then the case of multiple replicated data. We obtain adequate protocols simply by modifying the consensus filter. This conveys the flexibility gained by our modular approach.

5.3.1 Atomic Commitment on One Replicated Data

Consider a transaction on *one* single replicated data and denote by $clients$ the set of data managers that handle these replicas. Assume that a majority of data managers is correct. A simple modification of the consensus filter given in Section 5.2 allows us to adapt the generic framework to this specific atomic commitment problem. Consider a server s_j and let r stands for "*replication*" and o for "*one*" data:

Predicate *ro-NB-AC-CallInitValue(cid)* :
if [for a majority of processes $p_i \in clients$:
s_j has received $(cid, vote_i, clients)$ from p_i]
then return $true$ **else return** $false$.

Function *ro-NB-AC-InitValue($dataReceived_j$)* :
if [for every $(cid, vote_i, clients)$
$\in dataReceived_j$, $vote_i = yes$]
then return $commit$ **else return** $abort$.

The predicate *ro-NB-AC-CallInitValue* returns $true$ as soon as a majority of votes have been received. The function *ro-NB-AC-InitValue* returns $commit$ only if all these votes are yes.

5.3.2 Atomic Commitment on Multiple Replicated Data

Consider now a transaction on *multiple* replicated data. The previous filter can easily be extended to handle this case. Let us denote, by $clients_i$, the set of data managers that handle the replicas of $data_i$ and, by $clients$, the union of the sets $clients_i$. The consensus filter follows immediately (r stands for "*replication*," m for "*multiple*" data):

Predicate *rm-NB-AC-CallInitValue(cid)* :
if [for every set $clients_i$:
for a majority of processes $p_i \in clients_i$:
s_j has received $(cid, vote_i, clients)$ from p_i]
then return $true$ **else return** $false$.

Function *rm-NB-AC-InitValue($dataReceived_j$)* :
if [for every set $clients_i$:
for a majority of processes $p_i \in clients_i$:
$(cid, vote_i, clients) \in dataReceived_j$
and $vote_i = yes$]
then return $commit$ **else return** $abort$.

The predicate *rm-NB-AC-CallInitValue* returns $true$ as soon as a majority of votes have been received from every set of data managers $clients_i$. The function *rm-NB-AC-InitValue* returns $commit$ only if there is a majority of yes votes from every set of data managers $clients_i$.

6 Group Membership

The generic construction of agreement protocols based on a consensus service has been illustrated in the previous section on the nonblocking atomic commitment problem. In this section, we illustrate our approach on the *group membership* problem.

6.1 Background

Roughly speaking, the *group membership* problem for a group of processes consists of agreeing on the set of operational processes within the group. A process calls this information its *view* of the group. As processes may join or leave a group, a process view of the group membership may change over time. When a process changes its view, we say that it *installs* a new view. We consider here the so-called *primary partition membership* [8] where, for any given group, a unique totally ordered sequence of views is defined (i.e., we do not consider the case where concurrent views may coexist [4]). For simplicity of presentation and to convey the modularity of our approach, we first consider a group membership problem where processes can only be excluded from a view. We call it the GM-L problem (L stands for *leave*) and we show how this problem can be solved using our consensus-based transformation. Then, we define what we call the (*full*) GM problem by extending the specification to handle process joins and we show how this problem is solved using a simple variation of the GM-L consensus filter.

It is important to notice that our objective here is not to discuss the differences between various specifications of the group membership problem. The specifications we consider are aimed at illustrating our modular approach, i.e., the transformation of a specification to a consensus problem. In particular, the specifications we consider are rather strong and the solutions we suggest require the use of a perfect failure detector [10]. Later in this section, we will discuss the impact of weakening the problem specifications (to cope with unreliable failure detection) on our consensus transformations.

6. This is just one example of a possible alternative NB-AC-Validity property. Our goal here is to show how validity conditions translate into a consensus filter. Notice, however, that our majority condition requires, for every logical $data_i$, a majority of correct data manager replicas.

The specification below is patterned after [11]. We consider a given group g, an integer $i \geq 0$, and we assume that all processes in $v_i(g)$ have installed the view $v_i(g)$ (initially, $v_0(g) = g$). We then define the problem for the processes in $v_i(g)$ to install a new view $v_{i+1}(g)$ through the following properties:

GM-L-Termination. If a process $p_k \in v_i(g)$ wishes to leave $v_i(g)$ or crashes, then every correct process in $v_i(g)$ eventually installs $v_{i+1}(g)$.

GM-L-Agreement. If a process $p_k \in v_i(g)$ installs $v_{i+1}(g)$ and a process $p_{k'} \in v_i(g)$ installs $v'_{i+1}(g)$, then $v_{i+1}(g) = v'_{i+1}(g)$.

GM-L-Validity. If a process installs $v_{i+1}(g)$, then it can be said that $v_{i+1}(g) \subset v_i(g)$ and, for every process $p_k \in v_i(g) \setminus v_{i+1}(g)$, either p_k has crashed or p_k has wished to leave $v_i(g)$.

6.2 Solving the GM-L Problem Based on a Consensus Service

We show below how the GM-L problem can be solved using the consensus service under the assumption of a perfect failure detector [10] (among the client processes).

The consensus service is used as follows to enable processes in view $v_i(g)$ to install a new view $v_{i+1}(g)$:

The initiator. If a process p_k suspects some process in $v_i(g)$ or if p_k wishes to leave $v_i(g)$, then p_k reliably multicasts the message $(cid, \textit{view-change}, v_i(g))$ to the set of clients $v_i(g)$. Process p_k is the initiator of the consensus interaction. The consensus id "*cid*" is the pair $(gid, i+1)$, where gid is g's group id and i is the current view number of process p_k.

The clients. Upon reception of the message sent by the initiator,[7] a client process p_k "multisends" to the consensus servers either 1) the message $(cid, p_k\text{-}no, v_i(g))$ if p_k wishes to leave $v_i(g)$ or 2) the message $(cid, p_k\text{-}yes, v_i(g))$ otherwise. The decision computed by the consensus service is the new view $v_{i+1}(g)$.

GM-L-Agreement and GM-L-Termination follow from the consensus service (CS-Agreement and CS-Termination properties) and from the liveness of the GM-L consensus filter (see below). The GM-L-Validity follows from the consensus service (CS-Validity), the assumption of a perfect failure detector, and the GM-L consensus filter below.

The consensus filter. The GM-L consensus filter at server s_j is defined as follows:

Predicate *GM-L-CallInitValue(cid)* :
  if received $(cid, \text{-}, clients)$ from one process in $clients$
    and for every process $p_i \in clients$:
      [received $(cid, \text{-}, clients)$ from p_i or s_j suspects p_i]
  then return $true$ **else return** $false$.

Function *GM-L-InitValue(dataReceived$_j$)* :
  return$\{p_k \mid (cid, p_k\text{-}yes, clients) \in dataReceived_j\}$.

We show in the following that the GM-L consensus filter is live and, given the assumption of a perfect failure detector, it ensures the GM-L-Validity property.

7. Notice that there can be more than one initiator for the view change from $v_i(g)$ to $v_{i+1}(g)$. Our protocol also works in the case of multiple initiators.

Liveness of the GM-L consensus filter. We assume at least one correct process p_k in $v_i(g)$ (otherwise, the GM-L-Termination property is trivially ensured). Let p_k be a process that wishes to leave view $v_i(g)$. Case 1: If p_k is correct, then p_k initiates an interaction with the consensus service and all correct members of $v_i(g)$ send messages $(cid, \text{-}, v_i(g))$ to the consensus servers. Case 2: If p_k crashes, then, by the strong completeness property of the failure detector, p_k is eventually suspected by all correct processes of $v_i(g)$ and at least one correct process in $v_i(g)$ initiates an interaction with the consensus service. In both Cases 1 and 2, every correct consensus server receives at least one message $(cid, \text{-}, v_i(g))$. Moreover, as the failure detector satisfies strong completeness, the *GM-L-CallInitValue* predicate eventually returns $true$, i.e., the filter is live.

GM-L-Validity property is satisfied. We first prove that a process in $v_i(g)$ that does not crash and does not want to leave is in $v_{i+1}(g)$. Let $p_k \in v_i(g)$. Consider that p_k does not crash and does not wish to leave the view $v_i(g)$. We show that, in this case, p_k is in the new view $v_{i+1}(g)$. If p_k does not crash, then, by the strong accuracy property of the failure detector, no process suspects p_k. Hence, at every correct server process, *GM-L-CallInit* waits for the message $(cid, \text{-}, clients)$ from p_k. If p_k does not wish to leave $v_i(g)$, then p_k sends the message $(cid, p_k\text{-}yes, clients)$. By the *GM-L-InitValue* function, every consensus server that starts consensus has an initial value that includes p_k. By the validity property of consensus, the new view $v_{i+1}(g)$ includes p_k.

We now prove that $v_{i+1}(g) \subset v_i(g)$. Because no processes are added, it is sufficient to prove that $v_{i+1}(g) \neq v_i(g)$. There can be more than one initiator for a view change. Let p_k be the first one. Process p_k sends the $(cid, \textit{view-change}, v_i(g))$ message only 1) if it wishes to leave or 2) if it suspects some process $p_{k'}$. In Case 1, it is easy to show that no server that starts consensus has an imitial value that includes p_k. So, the new view $v_{i+1}(g)$ does not include p_k. In Case 2, because of the perfect detector, no server that starts consensus has an initial value that includes $p_{k'}$. So, the new view $v_{i+1}(g)$ does not include p_k.

6.3 Variations on the Consensus Filter

In the following, we show how to handle, first, process *joins* and, second, unreliable failure detection, using our consensus-based transformation. In each case, we discuss the impact of the new problem specification on the consensus filter.

6.3.1 Group Membership with Process Joins

As pointed out above, the GM-L problem is only concerned with process *removals* and does not include process *joins*. We define a "full" GM problem that handles process *joins* by extending the GM-L-Termination and GM-L-Validity properties as follows (the GM-L-Agreement property remains unchanged):

GM-Termination. If a process $p_k \in v_i(g)$ either 1) wishes to leave $v_i(g)$, 2) crashes, or 3) wishes to add some process

p_x ($p_x \notin v_i(g)$), then every correct process in $v_i(g)$ eventually installs $v_{i+1}(g)$.[8]

GM-Agreement. If a process $p_k \in v_i(g)$ installs $v_{i+1}(g)$ and a process $p_{k'} \in v_i(g)$ installs $v'_{i+1}(g)$, then $v_{i+1}(g) = v'_{i+1}(g)$.

GM-Validity. If a process installs $v_{i+1}(g)$ then the following can be noted: 1) $v_{i+1}(g) \neq v_i(g)$, 2) for every process $p_k \in v_i(g) \setminus v_{i+1}(g)$, either p_k has crashed or p_k wishes to leave $v_i(g)$, and 3) for every process $p_k \in v_{i+1}(g) \setminus v_i(g)$, there is at least one process in $v_i(g)$ that has wished to add process p_k.

The GM-L consensus-based solution can easily be adapted to the full GM problem. The initiator p_k multicasts the same message $(cid, \textit{view-change}, v_i(g))$ as above if 1) it suspects some process in $v_i(g)$, 2) it wishes to leave $v_i(g)$, or 3) it wishes to add some process p_x. A client process p_k, upon reception of the message sent by the initiator, multisends the message $(cid, p_k\text{-}yes, addSet_k, v_i(g))$ or $(cid, p_k\text{-}no, addSet_k, v_i(g))$, where $addSet_k$ is the set of processes that p_k wishes to add ($addSet_k \cap v_i(g) = \emptyset$). The *CallInitValue* predicate remains the same as in the previous section. The *InitValue* function is the following:

Function $GM\text{-}InitValue(dataReceived_j)$:
return $\{p \mid ((cid,\ p_k yes,\ addSet_k,\ clients) \in dataReceived_j)$ **and** $(p = p_k$ **or** $p \in addSet_k)\}$.

The proofs that the filter is live and that the GM-Validity property is satisfied are very similar to the proofs in Section 6.2.

6.3.2 Group Membership with Weak Validity

Given the GM-Validity (resp. GM-L-Validity) property, a process can be excluded only if it wishes to leave the view or if it crashes. Combined with the GM-Termination (resp. GM-L Termination) property, this requires perfect failure detection (both strong completeness and strong accuracy). We may also consider weaker specifications of the group membership problem where the accuracy property of the failure detectors is not necessary (for the consensus filter). In this specification, a process could be excluded if it is suspected (even without having actually crashed). To prevent the case where too many processes are excluded (because the failure detector makes too many mistakes), we may consider a validity property that requires a majority of processes in view $v_{i+1}(g)$, namely:

GM-WeakValidity. If a process installs $v_{i+1}(g)$, then 1) $v_{i+1}(g) \neq v_i(g)$, 2) $v_{i+1}(g)$ contains a majority of processes from $v_i(g)$, 3) for every process $p_k \in v_i(g) \setminus v_{i+1}(g)$, either p_k is suspected[9] or p_k wishes to leave $v_i(g)$, and 4) for every process $p_k \in v_{i+1}(g) \setminus v_i(g)$, there is at least one process that has wished to add p_k.

The GM filter can be easily adapted to such a property and proven live under the assumptions that a majority of processes in view $v_i(g)$ are correct and the failure detector satisfies strong completeness (every process that crashes is eventually suspected by every correct process).[10] Note that GM-WeakValidity does not ensure the presence of a majority of correct processes in $v_{i+1}(g)$.

7 View Synchrony

Now, we illustrate the generic construction of agreement protocols on an extension of group membership known as *view synchrony* or, more accurately, *view synchronous communication*.

7.1 Background

View synchronous communication (in the context of the primary partition model) has been introduced by the Isis system [7], [8] and later formalized in [32]. It augments a group membership protocol with an additional group broadcast primitive that orders messages with respect to the installation of views. We use here the term *view synchronous broadcast* and we denote the view synchronous broadcast primitive in this context by *VScast*.

For simplicity of presentation, we restrict ourselves to the case where processes can only be excluded from a view: Process *joins* can be handled as in Section 6.3. Furthermore, we consider a strong version of the problem where a process can only be excluded from a view if it wishes to leave the view or it crashes. Our solution to the problem relies on a perfect failure detector. Similarly to group membership, one could design alternative solutions (with weaker failure detection mechanisms) that exclude processes in the case of false failure suspicions (see Section 6.3).

Let $v_i(g)$ be the current view of process p_k in g and let p_k VScast message m to processes in $v_i(g)$. We denote by $VSdeliver$ the corresponding delivery of m. Roughly speaking, the VScast primitive ensures that 1) either no new view is installed, and all the members of $v_i(g)$ eventually VSdeliver m or 2) a new view $v_{i+1}(g) \subset v_i(g)$ is installed and if any process in $v_{i+1}(g)$ has VSdelivered m before installing $v_{i+1}(g)$, then all the processes that install $v_{i+1}(g)$ VSdeliver m before installing $v_{i+1}(g)$. View synchronous multicast is adequate, for example, in the context of the primary-backup replication technique, to multicast the *update* message from the primary to the backups [20].

We specify here the view synchronous communication problem as an extension of the group membership problem (GM-L) defined in Section 6.1. Let $v_i(g)$ be the current view of $p_k \in v_i(g)$ and let p_k VScast message m:

VS-Termination. GM-L-Termination plus the following property: a correct process in $v_i(g)$ eventually either 1) VSdelivers m or 2) installs view $v_{i+1}(g)$.

VS-Agreement. Identical to GM-L-Agreement.

8. One could consider a different specification where every correct process in $v_{i+1}(g)$ (rather than in $v_i(g)$) eventually installs $v_{i+1}(g)$. This would require a small change in the interaction between the clients ($v_i(g)$) and the consensus service: The decision of the consensus would need to be sent to $v_{i+1}(g)$, rather than to the clients $v_i(g)$ only.

9. Suspected either by a client or by a server process. Notice that, in the case where the server role is played by a subset of the client processes (see Section 4.1), the distinction between "suspected" by a server and "suspected" by a client disappears. Considering only this special case is nicer from the specification point of view, but not from the framework point of view (loss of generality).

10. When assuming no accuracy property for the failure detector, we have to bring a small modification to the algorithm of the clients: The condition $v_i(g) \neq v_{i+1}(g)$ of the GM-WeakValidity property can only be ensured by having each client process install a new view iff it is different from the current one.

VS-Validity. GM-L-Validity plus the following property: If there exists one process $p_k \in v_i(g) \cap v_{i+1}(g)$ that has VSdelivered m before installing $v_{i+1}(g)$, then every process in $v_i(g) \cap v_{i+1}(g)$ has VSdelivered m before installing $v_{i+1}(g)$.

The VS-Termination property requires that every correct process either VSdelivers m or installs view $v_{i+1}(g)$. However, VS-Validity (more precisely, GM-L-Validity) specifies that view $v_{i+1}(g)$ can only be installed if a process $p_k \in v_i(g)$ crashes or wishes to leave the group. So, if no process in $v_i(g)$ crashes or wishes to leave the group, all processes in $v_i(g)$ eventually VSdeliver m. If a new view $v_{i+1}(g)$ is installed, the VS-Validity property specifies atomicity with respect to the VSdelivery of m and ordering of the VSdelivery of m before the installation of the new view $v_{i+1}(g)$.

7.2 VScast Based on a Consensus Service

Let $v_i(g)$ be the current view of process $p_k \in v_i(g)$ and let p_k VScast message m. If no process crashes and no process wishes to leave the group, then VScast can be implemented without interaction with the consensus service. This is done by having p_k send m to all the processes in $v_i(g)$.

Upon reception of m, message m is VSdelivered. Then, p_k sends an acknowledgment to all processes in $v_i(g)$. As soon as $p_{k'}$ has received the acknowledgment for m from all processes in $v_i(g)$, the predicate $stable_{k'}(m)$ holds. The predicate is used when a process crashes or wishes to leave the group. In this case, the interaction with the consensus service ensures the VScast semantics as follows:

The initiator. If 1) a process p_k suspects some process in $v_i(g)$ or 2) p_k wishes to leave $v_i(g)$, then p_k reliably multicasts the message $(cid, \textit{view-change}, v_i(g))$ to the set of clients $v_i(g)$. The consensus id "*cid*" is the pair $(gid, i+1)$, where gid is g's group id and i the current view number of process p_k.

The clients. Upon reception of the message sent by the initiator, every client process p_k multisends either 1) the message $(cid, p_k\text{-}no, v_i(g))$ to the consensus service if p_k wishes to leave $v_i(g)$ or 2) the message $(cid, (unstable_k, p_k\text{-}yes), v_i(g))$, where $unstable_k$ denotes the set of messages m such that $stable_k(m)$ does not hold. The decision computed by the consensus service is a pair $(unstable, v)$, where $unstable$ is a set of messages and v is the new view $v_{i+1}(g)$.

Upon reception of the decision $(unstable, v)$, every client process p_k first VSdelivers the messages in $unstable$ that it has not yet VSdelivered and then installs the new view v. While waiting for a consensus decision, every client process p_k discards all new messages that are received.

The VS-Agreement property follows from the CS-Agreement property of the consensus service. Consider the VS-Termination property. If no process $p_k \in v_i(g)$ crashes or wishes to leave $v_i(g)$, then, by the reliable channel assumption, every process in $v_i(g)$ eventually VSdelivers m and the VS-Termination holds. Otherwise, the VS-Termination property follows from the liveness of the VS consensus filter (see below).

The consensus filter. The VS consensus filter defines initial values for the consensus problem. The initial value for server s_j is a pair $(unstable_j, v_j)$, where $unstable_j$ is a set of unstable messages and v_j a set of processes. The decision computed by the consensus service is a pair $(unstable, v)$, where $unstable$ is the set of unstable messages to be VS-delivered before installing the new view v (i.e., $v_{i+1}(g)$). The VS consensus filter is defined as follows:

Predicate *VS-CallInitValue(cid)* : Identical to the predicate *GM-L-CallInitValue*

Function *VS-InitValue*($dataReceived_j$) :
$v_j \leftarrow \{p_k \mid (cid, (unstable_k, p_k\text{-}yes), clients) \in dataReceived_j\}$;
$unstab_j \leftarrow \{m \mid (cid, (unstable_k, yes), clients)$ *has been received* **and** $m \in unstable_k\}$;
return $(unstab_j, v_j)$

The proof of liveness of the VS consensus filter is similar to the proof of liveness of the GM-L consensus filter (Section 6.2).

The VS-Validity property consists of two subproperties: the GM-L-Validity property plus the additional *message-view ordering* property. The correctness proof of the GM-L-Validity property was discussed in Section 6.2. The additional property is satisfied for the following reason: Consider process $p_k \in v_i(g)$, $p_k \in v_{i+1}(g)$, and assume that p_k has VSdelivered some message m before installing $v_{i+1}(g)$. As $p_k \in v_{i+1}(g)$, process p_k has multisent message $(cid, (unstable_k, p_k\text{-}yes), clients)$ to the consensus service. Since p_k does not VSdeliver any message while waiting for the consensus decision, then if p_k has VSdelivered m before receiving $(unstable, v_{i+1}(g))$ and installing $v_{i+1}(g)$, either 1) $stable_k$ holds or 2) $m \in unstable$. In Case 1, every process in $v_i(g)$ has received and VSdelivered m before installing $v_{i+1}(g)$. In Case 2, by the CS-Agreement property, every process that installs $v_{i+1}(g)$ first VSdelivers m. So, the additional *message-view ordering* property holds.

8 Total Order Multicast/Broadcast

8.1 Total Order Multicast vs. Total Order Broadcast

The difference between total order multicast and broadcast has to do with message destination sets. Let $Dst(m)$ denote the destination set of message m and let m_1 and m_2 be any two messages. With total order broadcast, if $p \in Dst(m_1)$ and $p \in Dst(m_2)$, then $Dst(m_1) = Dst(m_2)$. In other words, total order broadcast forbids overlapping destinations. This restriction does not apply to total order multicast which allows issuing messages to overlapping destinations (we discuss these differences in detail in [19]).

Most total order algorithms that were proposed in the literature are total order broadcast algorithms and many of them rely on a membership service or a view synchronous communication primitive (e.g., [8], [1], [12]). These algorithms operate in two modes: 1) a *normal* mode which lasts as long as no process is suspected to have crashed and 2) a *special* mode in which a termination protocol ensures the ordering property while installing a new membership. The special mode is based on protocols that have been discussed

in the previous section (membership and view synchronous communication).

We restrict our discussion below to algorithms that do not require a membership service, i.e., to algorithms that operate in one single mode. Total order algorithms differ slightly from the agreement problems discussed in the previous sections for the following reason: Agreeing on an order requires agreeing on a value first and then inferring an order from that value.

In the total order broadcast algorithm of [10], agreement is achieved on a *set of messages* (which is a case where multiple agreement problems are solved by one instance of consensus). This algorithm can be straightforwardly expressed in our generic communication scheme with an empty consensus filter.[11] In the following, we illustrate our generic solution on a total order *multicast* algorithm.

8.2 Total Order Multicast

8.2.1 Background

The algorithm we describe here is an extension of a nonfault-tolerant total order multicast algorithm proposed by Skeen [35]. Basically, we show how to make that protocol fault-tolerant by using our consensus service.

We denote by *TO-multicast(m, Dst(m))* the primitive by which a process multicasts message m according to total order multicast semantics and *TO-deliver(m)* the corresponding delivery event. The basic idea of Skeen's algorithm consists of having the processes agree on a sequence number $sn(m)$ for every message m and *TO-deliver* the messages in the order of their sequence numbers. The sequence number is based on strictly increasing timestamps provided by the receiving processes. More precisely, when *TO-multicast(m, Dst(m))* is executed by p_i, process p_i sends the message m to all processes in $Dst(m)$ and collects the timestamps attached to m by these processes. Process p_i then defines $sn(m)$ as the maximum of these timestamps and sends back $sn(m)$ to $Dst(m)$. Skeen's algorithm does not tolerate the failure of a single process. Indeed, to compute a sequence number $sn(m)$, the sender of a message m waits for timestamps from all destination processes.

Given the execution of *TO-mutlicast(m, Dst(m))*, we consider here the problem of agreeing on the sequence number $sn(m)$ in spite of process crashes. In other words, we consider a single instance of the multicast and we focus on the problem of agreeing on a sequence number for m ($sn(m)$), based on the timestamps attached to m by the processes in $Dst(m)$. We call it the SN problem. Inferring the order from the $sn(m)$ value is not discussed here. It can be found in [29], which improves on the idea originally presented in [18]. Given a message m and the primitive *TO-multicast(m,Dst(m))*, the SN problem is defined by the following properties:

SN-Termination. If a correct process TO-multicasts m, then every correct process in $Dst(m)$ eventually decides $sn(m)$.

SN-Agreement. No two processes in $Dst(m)$ decide on two different sequence numbers for m.

SN-Validity. The sequence number $sn(m)$ is computed as the maximum of the timestamps provided by all correct processes in $Dst(m)$.

8.2.2 Computing $sn(m)$ Using a Consensus Service

Consider *TO-multicast(m, Dst(m))* executed by some process p_i and let $id(m)$ denote the id of message m. The consensus service is used as follows to compute $sn(m)$:

The initiator. Process p_i reliably multicasts $(cid, m, clients(cid))$ to the set $clients(cid)$; cid is $id(m)$ and the set $clients(cid)$ is $Dst(m)$.

The clients. Upon reception of $(cid, m, clients(cid))$, a client p_i defines $data'_i$ as the timestamp ts_i of the $receive$ event, according to Lamport's clock, and multisends $(c_i, ts_i, clients(cid))$ to the consensus servers.

The SN consensus filter. The filter of server s_j is defined as follows:

Predicate *SN-CallInitValue*:
if for every process $p_i \in clients$:
[received $(cid, -, clients)$ from p_i **or** s_j suspects p_i]
then return $true$ **else return** $false$.

Function *SN-InitValue(dataReceived$_j$)* :
$sn(m) \leftarrow \max\{ts_i(m) \mid (cid, ts_i(m), clients(cid)) \in dataReceived_j\}$
return $sn(m)$

It is easy to show that the filter ensures the SN-Termination, SN-Agreement, and SN-Validity properties defined above. Basically, the *SN-CallInitValue* predicate returns $true$ as soon as the message $(cid, ts_i(m), clients(cid))$ has been received from all nonsuspected processes in $Dst(m)$. The function *SN-InitValue* returns the maximum of the timestamps $ts_i(m)$ received. More details on this protocol are given in [29]. It is worthwhile to point out here that the protocol is correct under the assumption of a perfect failure detector [10]. Given this assumption and the definition of the consensus filter, we ensure that $sn(m)$ is always computed as the maximum of the timestamps from all correct processes in $Dst(m)$. We show in [19] that, in order to tolerate even a single crash failure, any *genuine* total order multicast protocol requires a perfect failure detector. Overcoming that requirement in specific models is discussed in [18] and in [19].

9 Cost Evaluation

We describe below the overall cost of a general interaction with the consensus service in terms of the number of messages and communication steps. This cost is the same for all agreement protocols presented in the previous sections. We will use this cost to compare the efficiency of agreement protocols built following our modular approach with the efficiency of specialized agreement protocols. As we will show, the generality of the consensus service approach does not imply a loss of efficiency. On the contrary, our modular architecture enables interesting optimizations.

11. This actually leads to a degenerated version of our generic communication scheme.

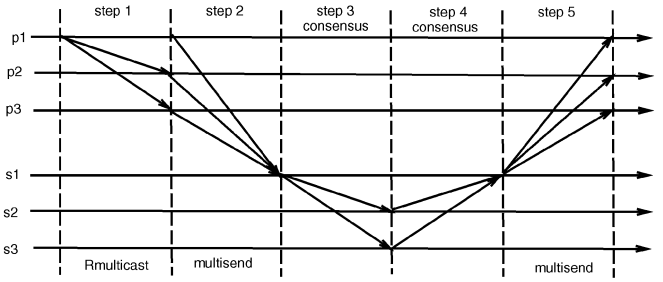

Fig. 7. Centralized scheme (in good runs): p_1 is the initiator; p_1, p_2, p_3 are the clients; s_1, s_2, s_3 implement the consensus service.

Up to now, we have considered consensus as a black box. In the following and to discuss efficiency issues, we consider consensus implementations. We distinguish two approaches: a centralized one where the consensus decision is taken through one coordinator and a decentralized one where there is no coordinator. For both approaches, we first point out some optimizations and then we present implementation costs in terms of messages and communication steps.

We reasonably assume that runs with no failure and no failure suspicion are the most frequent ones and implementations should be optimized for these runs. We call a "good run" a run in which no failure occurs and no failure suspicion is generated.

9.1 Centralized Algorithm

We consider the (centralized) consensus algorithm presented by Chandra and Toueg [10], noted here $\diamond\mathcal{S}$-consensus. This algorithm requires a majority of correct processes and a failure detector of class $\diamond\mathcal{S}$.

In the original description of the $\diamond\mathcal{S}$-consensus protocol, every process p_j starts with an initial value v_j. In fact, it is sufficient for one correct process to start with an initial value. In other words, when invoking the consensus service, it is sufficient that one correct member of the consensus service has an initial value. We thus assume an implementation of the *multisend* primitive such that, in runs with no failures, client processes need not send their messages to all consensus servers: It is sufficient that they send their messages to one server, e.g., to s_1. The client processes should send their messages to the other servers only upon suspecting s_1 (we assume a failure detector with strong completeness). This ensures that at least one correct consensus server gets an initial value. In the following, we consider our protocol with this optimization.

We also assume an optimized implementation of reliable multicast (used by the initiator to send its message to the clients). If the initiator process p_i executing "*Rmulticast(m)*" to the clients is correct, no client needs to relay m. A client process relays m only when it suspects p_i. This optimized implementation costs only one communication step and $O(n)$ messages in good runs.

Let n_c be the number of clients, and n_s the number of servers. Fig. 7 illustrates the five communication steps and $3n_c + 2n_s - 3$ messages needed before the clients receive the decision of the consensus (i.e., the solution of the agreement problem):

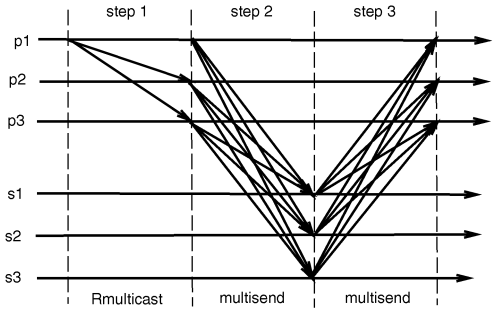

Fig. 8. Decentralized scheme (in good runs): p_1 is the initiator; p_1, p_2, p_3 are the clients; s_1, s_2, s_3 implement the consensus service.

Step 1, the reliable multicast from the initiator to the set of clients, costs $n_c - 1$ messages.

Step 2, the multisend from the clients to one of the servers (say s_1), costs n_c messages.

Steps 3 and 4 correspond to messages sent within the $\diamond\mathcal{S}$-consensus protocol. In good runs, s_1 knows the decision at the end of Step 4. Steps 3 and 4 each cost $n_s - 1$ messages (see Fig. 7).

Step 5, the multisend initiated by the server s_1 to the clients, costs n_c messages.

9.2 Decentralized Scheme

This implementation takes advantage of the validity property of consensus: If each member of the consensus service starts the consensus with the same initial value v ($\forall s_i, s_j$, we have $v_i = v_j = v$), then the decision is v. We exploit this property through the following interaction scheme (see Fig. 8):

Step 1, as before, is the (optimized) reliable multicast from the initiator to the set of clients and it costs $n_c - 1$ messages.

In Step 2, the clients multisend their messages to all the server processes and every member of the consensus service gets an initial value. This costs $n_c * n_s$ messages.

In Step 3, the consensus servers simply send their initial value to the clients. This costs $n_s * n_c$ messages. A client receiving the same initial value v from every member of the consensus service knows that v is the decision. If this is not the case, the $\diamond\mathcal{S}$-consensus is used as a termination protocol. This case is not depicted in Fig. 8 (we give a detailed description in [17] for the case of atomic commitment).

Despite the fact that, whenever $n_s \geq 2$, the number of messages is higher in Fig. 8 than in Fig. 7, reducing the number of communication steps from five to three reduces the latency. Moreover, with a network that provides broadcast capabilities, the decentralized scheme can be far more efficient than the centralized one because the cost of sending a message to n processes is the same as the cost of sending a message to one process.

9.3 Comparison with Three Phase Commit

We compare below the performance of a Nonblocking Commit Protocol built following our approach with those of Skeen's well-known Three Phase Commit protocols (3PC) [34]. Alternative nonblocking atomic commitment protocol

that we know of (e.g., [22]) increase the resilience of 3PC (tolerating network partitions), but, compared with our protocol, do not lead to better performance in good runs.[12]

Assume that the consensus service is implemented by the clients themselves and consider only runs where no process crashes or is suspected to have crashed. The communication scheme of our NB-AC protocol using the centralized implementation above is similar to the communication scheme of the 3PC protocol [34]. Furthermore, if we consider the decentralized implementation, the communication scheme of our NB-AC protocol is similar to the communication scheme of the D3PC protocol (Decentralized 3PC of Skeen) [34].

However, based on a consensus service, our solution is more modular and, in both cases (centralized or decentralized), allows us to trade the number of messages exchanged against resilience. If we denote the number of clients and the number of servers by n_c and n_s, respectively, then if n_s decreases, the resilience of the consensus server decreases, but the number of messages also decreases. In the case $n_c > n_s$, our centralized solution requires fewer messages than 3PC and our decentralized solution requires fewer messages than D3PC. For instance, our centralized solution requires $3n_c + 2n_s - 3$ messages, whereas the 3PC requires $5n_c - 5$ messages. In practice, $n_s = 3$ achieves a sensible resilience. In this case, $3n_c + 2n_s - 3 < 5n_c - 3$ is true already for $n_c = 4$ (a transaction on three objects, i.e., one transaction manager and three data managers, leads to $n_c = 4$). In [17], we present experimental results confirming that an optimized consensus-based NB-AC protocol is more efficient that a 3PC protocol.

10 Concluding Remarks

This paper advocates the idea that consensus is a central abstraction for building fault-tolerant agreement protocols in a modular way: The paper presents a unified framework from which one can derive, simply by customizing a generic *consensus filter*, protocols that are usually considered and implemented separately. The same framework allows us to express protocols for atomic commitment, group membership, view synchrony, and total order multicast.

Our framework can be viewed as a first step toward building practical systems that provide support for various paradigms, mixing, for instance, transactions and view synchronous communication. In this context, consensus would not only be a useful theoretical concept [33], [36], but also a useful service for the clean development of reliable distributed systems. Apart from the agreement problems considered in the paper, one could of course consider other agreement problems like election [30] or terminating reliable broadcast [10].

Our framework was designed in the context of asynchronous distributed systems with process crash failures and failure detectors. That is, the framework needs "no assumption" on process communication delays and process relative speeds. One could apply the same framework in systems with stronger assumptions (e.g., a synchronous model) or different failure models. This might require modification of the implementations of our framework basic building blocks, i.e., communication primitives, failure detectors, and consensus. For instance, if a crash-recovery semantics is assumed, one could use the consensus protocol of [27], [21], [26], [3]. However, the generic interaction and the consensus filter would remain the same. An interesting question in this context is to what extent the assumptions on the underlying system model impacts the performance.

12. Note that our assumption of quasi-reliable channels is also compatible with temporary link failures and network partitions.

Acknowledgments

The authors would like to thank Vassos Hadzilacos and the anonymous reviewers for their useful comments on earlier versions of the paper. This paper is an extended and revised version of a paper that appeared, under the title "Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems," in the *Proceedings of the 26th IEEE International Symposium Fault-Tolerant Computing* (FTCS-26), Sendai, Japan, June 1996, pages 168-177.

References

[1] Y. Amir, L. Moser, P. Melliar Smith, D. Agarwal, and P. Ciarfella, "The Totem Single-Ring Ordering and Membership Protocol," *ACM Trans. Computer Systems,* vol. 13, no. 4, pp. 311-342, Nov. 1995.

[2] M.K. Aguilera, W. Chen, and S. Toueg, "Heartbeat: A Time-Out Free Failure Detector for Quiescent Reliable Communication," *Proc. Distributed Algorithms (WDAG '97),* Sept. 1997.

[3] M.K. Aguilera, W. Chen, and S. Toueg, "Failure Detection and Consensus in the Crash-Recovery Model," *Distributed Computing,* vol. 13, pp. 99-125, 2000.

[4] O. Babaoglu, R. Davoli, and A. Montresor, "Failure Detectors, GroupMembership and View-Synchronous Communication in Partitionable Systems," technical report, Computer Science Dept., Univ. of Bologna, Nov. 1995.

[5] A. Basu, B. Charron-Bost, S. Toueg, "Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes," *Proc. Distributed Algorithms (WDAG '96),* pp. 105-122, Oct. 1996.

[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[7] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Computer Systems,* vol. 5, no 1, pp. 47-76, Feb. 1987.

[8] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Computer Systems,* vol. 9, no. 3, pp. 272-314, Aug. 1991.

[9] T. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *J. ACM,* vol. 43, no. 4, pp. 685-722, July 1996.

[10] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 34, no. 1, pp. 225-267, Mar. 1996.

[11] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," Technical Report 95-1548, Dept. of Computer Science, Cornell Univ., Oct. 1995.

[12] D. Dolev, S. Kramer, and D. Malkhi, "Early Delivery Totally Ordered Broadcast in Asynchronous Environments," *Proc. IEEE Symp. Fault-Tolerant Computing,* pp. 296-306, June 1993.

[13] P. Ezhilchelvan, R. Macedo, and S. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol," *Proc. IEEE Conf. Distributed Computing Systems,* pp. 296-306, May 1997.

[14] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, pp. 374-382, Apr. 1985.

[15] R. Guerraoui, "Revisiting the Relationship between Non Blocking Atomic Commitment and Consensus Problems," *Proc. Distributed Algorithms (WDAG '95),* pp. 87-100, Sept. 1995.

[16] R. Guerraoui and A. Schiper, "Transactional Model vs. Virtual Synchrony Model: Bridging the Gap," *Theory and Practice in Distributed Systems,* pp. 121-132, 1995.
[17] R. Guerraoui, M. Larrea, and A. Schiper, "Reducing the Cost for Non-Blocking in Atomic Commitment," *Proc. IEEE Conf. Distributed Computing Systems,* pp. 692-697, May 1996.
[18] R. Guerraoui and A. Schiper, "Total Order Multicast to Multiple Groups," *Proc. 17th IEEE Int'l Conf. Distributed Computing Systems,* pp. 578-585, May 1997.
[19] R. Guerraoui and A. Schiper, "Genuine Atomic Multicast," *Proc. Distributed Algorithms (WDAG '97),* pp. 141-154, Sept. 1997.
[20] R. Guerraoui and A. Schiper, "Software-Based Replication for Fault-Tolerance," *Computer,* vol. 30, no. 4, pp. 68-74, Apr. 1997.
[21] M. Hurfin, A. Mostefaoui, and M. Raynal, "Consensus in Asynchronous Systems where Processes Can Crash and Recover," *Proc. 17th IEEE Symp. Reliable Distributed Systems,* pp. 280-286, Oct. 1998.
[22] I. Keidar and D. Dolev, "Increasing the Resilience of Atomic Commit at No Additional Cost," Technical Report CS94-18, Inst. of Computer Science, The Hebrew Univ. of Jerusalem, 1994.
[23] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 382-401, July 1982.
[24] C. Malloth, "Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks," PhD Thesis 1557, Swiss Federal Inst. of Technology (EPFL), 1996.
[25] C. Malloth and A. Schiper, "View Synchronous Communication in Large Scale Networks," ESPRIT Basic Research BROADCAST, Third Year Report, vol. 4, July 1995.
[26] R. Oliveira, "Solving Consensus: From Fair-Lossy Channels to Crash-Recovery of Processes," PhD Thesis 2139, Swiss Federal Inst. of Technology (EPFL), 2000.
[27] R. Oliveira, R. Guerraoui, and A. Schiper, "Consensus in the Crash-Recovery Model," Technical Report 97/239, Swiss Federal Inst. of Technology (EPFL), Aug. 1997.
[28] A. Ricciardi and K. Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments," *Proc. ACM Symp. Principles of Distributed Computing,* pp. 341-352, Aug. 1991.
[29] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable Atomic Multicast," *Proc. Seventh Int'l Conf. Computer Communications and Networks (ICCCN '98),* pp. 840-847, Oct. 1998.
[30] L. Sabel and K. Marzullo, "Election vs. Consensus in Asynchronous Systems," Technical Report TR95-1488, Cornell Univ., 1995.
[31] A. Schiper, "Early Consensus in an Asynchronous System with a Weak Failure Detector," *Distributed Computing,* vol. 10, no. 3, pp. 149-157, Mar./Apr. 1997.
[32] A. Schiper and A. Sandoz, "Uniform Reliable Multicast in a Virtually Synchronous Environment," *Proc. IEEE Conf. Distributed Computing Systems,* pp. 561-568, May 1993.
[33] F. Schneider, "Paradigms for Distributed Programs," *Proc. Distributed Systems—Methods and Tools for Specification,* pp. 343-430, 1985.
[34] D. Skeen, "NonBlocking Commit Protocols," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 133-142, 1981.
[35] D. Skeen unpublished communication, Feb. 1985, Referenced in [7].
[36] J. Turek and D. Shasha, "The Many Faces of Consensus in Distributed Systems," *Computer,* vol. 25, no. 6, pp. 8-17, June 1992.

Rachid Guerraoui is an assistant professor of communication systems at the Swiss Federal Institute of Technology in Lausanne since 1999. He has also worked with the Centre de Recherches de l'Ecole des Mines (Paris), the French Atomic Energy Commission (Saclay), and Hewlett-Packard Laboratories (Palo Alto, California). He teaches object-oriented programming and distributed systems and leads a research group in distributed programming. His research interests span distributed algorithms, transactional systems, and programming languages. He is a member of the IEEE.

André Schiper has been a professor of computer science at the Swiss Federal Institute of Technology in Lausanne (EPFL) since 1985, leading the Operating Systems Laboratory. During the academic year 1992-1993, he was on sabbatical leave at Cornell University, Ithaca New York. He was the program chair of the 1993 International Workshop on Distributed Algorithms (WDAG-7) and the general chair of the 1999 International Symposium on Reliable Distributed Systems (SRDS-18). His research interests are in the areas of fault-tolerant distributed systems, group communication, middleware, and security. He is a member of the IEEE.