The Generic Consensus Service

Rachid Guerraoui André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
Tel: +41 +21 693 52 72
Fax: +41 +21 693 67 70
e-mail: {Rachid.Guerraoui,Andre.Schiper}@epfl.ch

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 non-blocking 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, and this leads to original solutions for problems like group membership and view synchronous communication. In addition to its modularity, our approach enables efficient implementations of agreement protocols, and precise characterization of their liveness and safety properties.

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

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 on Fault-Tolerant Computing (FTCS-26), Sendai (Japan), June 1996, pages 168-177.

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 [8, 7, 14, 25], 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 leads 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: we have here 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 [27], 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 non-blocking 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 and view synchronous communication. Section 7 considers atomic broadcast and atomic multicast protocols. Section 8 presents a cost analysis and discusses efficiency issues. Section 9 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* [23], which was used to implement various total order broadcast protocols [6, 11, 12, 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 [8, 30] and thus are subject, in asynchronous systems, to the Fischer-Lynch-Paterson impossibility result [13, 7, 9].[1] Most of the agreement protocols described in the literature usually guarantee the required *safety* property, but fail to define the conditions under which *liveness* is ensured. Thanks to the recent work of Chandra and Toueg on failure detectors, we now have a formalism that allows to define precise conditions under which the consensus problem is solvable in asynchronous distributed systems. By defining a unified consensus-based framework for solving various agreement problems, we provide a way to reuse that formalism in proving the correctness of agreement protocols.

Our work can be viewed as continuation of the work of Schneider [27], 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 leads to original solutions for problems like group membership and view synchronous communication, and leads to highlight their common characteristics with problems usually considered separately like non-blocking atomic commitment.

3 System architecture and model

Our system architecture is depicted on Figure 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 (layer 3) problems are given in Section 5, 6, and 7.

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 does never execute any action. We do not consider for instance

[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 [13].

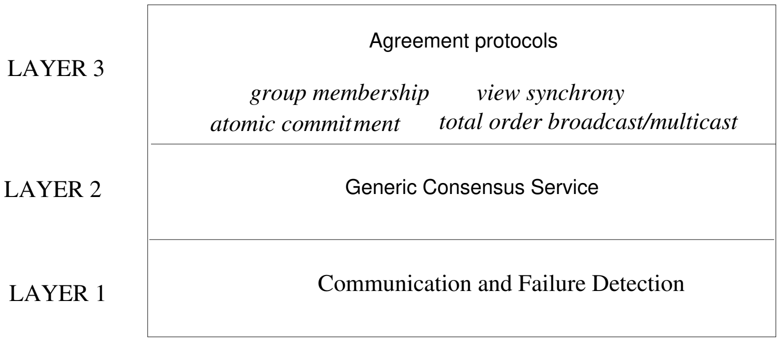

Figure 1: The architecture model

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.

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 *eventually reliable* [2]. In other words:

- 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.

Eventual reliable channels can be implemented by retransmitting messages. They do not exclude the possibility of temporary link failures (temporary partitions). An eventual reliable channel is weaker than a *reliable* channel [2] 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 eventual reliable channels can 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 eventual 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) $Rmulticast(m)$

[2]We will discuss in Section 9 the generalization of our framework to other fault models.

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 a reliable multicast. A *multisend* is implemented simply by sending multiple messages, whereas a *Rmulticast* requires message retransmission by a destination process (see [8] for more details on implementation of reliable multicast).

3.3 Failure detectors

Failure detectors have been formally introduced in [8, 7] 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.

As we consider in this paper consensus as a black box, we are not concerned with a formal characterization of failure detectors. For the general purpose of our framework, we just assume that the failure detector satisfies the so called *"strong completeness"* property: if some process p_i crashes, then every process p_j eventually suspects p_i forever. This property is easily implementable using heartbeat messages, or *Are you alive?/I am alive* message exchange. Later in the paper, and only when required, we will recall stronger properties of failure detectors.

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 [8]:

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 algorithm. The reader interested in learning more about solving consensus in an asynchronous system model augmented with failure detectors can consult [8, 25].

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 Section 5, Section 6 and Section 7.

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.

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 8). 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:

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

We will see throughout the paper that many agreement problems can be solved by the above three phase interaction. In most of the cases (Sect. 5 and Sect. 6), 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 7 the case of a *n-1* correspondence, where several instances of an agreement problem 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 on Fig. 2; and Fig. 3). The parameter *cid* (*consensus id*) uniquely

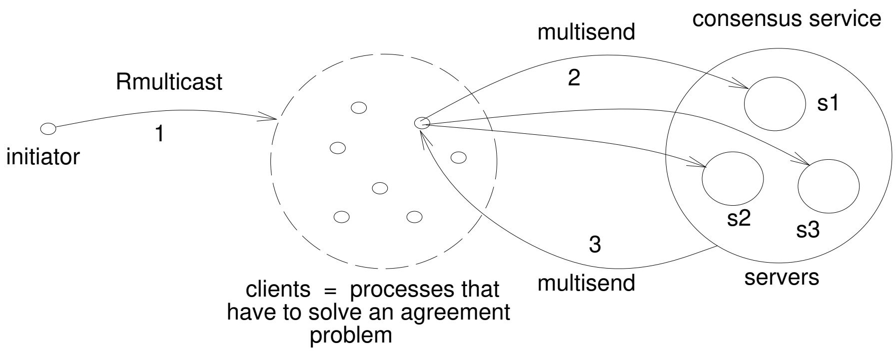

Figure 2: Interaction from a client's point of view

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.

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

Figure 3: Initiator's algorithm

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).

upon reception of $(cid, data, clients)$ by a client p_i:
  compute $data'_i$;
  $multisend(cid, data'_i, clients)$ to the members of the *consensus service* ;
  wait reception of $(cid, decision)$ from the *consensus service* ;

Figure 4: Algorithm of a client p_i

The servers. The interaction between the clients processes and the consensus service is illustrated from the point of view of a server process in Figures 6 and 5. The genericity of the consensus service is obtained thanks to the notion of *"consensus filter"*, depicted in Figure 6 as a shaded ring (arrows *to* and *from* s_2 and s_3 have not been drawn). The consensus filter allows 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 $InitValue$ (Figure 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 Figure 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).

1 wait reception of $(cid, data'_i, clients)$ from clients **until** *CallInitValue(cid)* ;
2 $v_j \leftarrow InitValue(\{data'_i \mid message\ (cid, data'_i, clients)\ received\})$;
3 $decision \leftarrow consensus(cid, v_j)$;
4 $multisend(cid, decision)$ to $clients$;

Figure 5: Algorithm of a server s_j

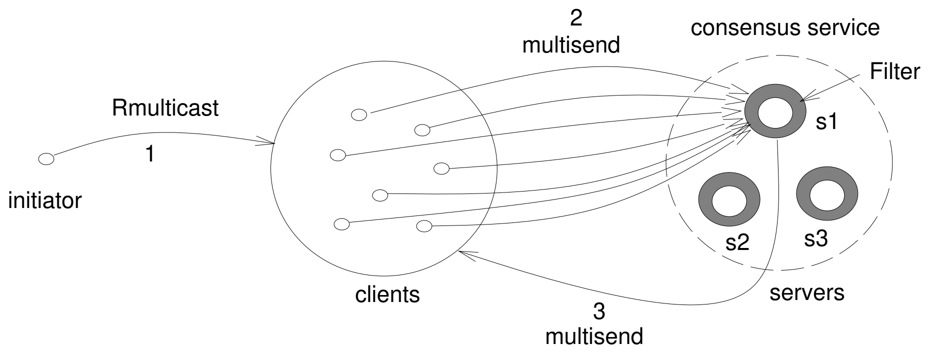

Figure 6: Invocation-reply from the point of view of server s_1

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

4.3 Correctness

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

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 (Sect. 3.4). Consider the CS-Termination (Consensus Service Termination) property. If the consensus filter is live, then all correct members of the consensus service eventually start consensus. By the termination property of consensus (Sect. 3.4), every correct server eventually decides, and sends the $(cid, decision)$ message to the clients. As we assume at least one correct server and eventual reliable channels, every correct client eventually receives the decision message $(cid, decision)$.

5 Non-Blocking Atomic Commitment

Throughout this section we show how a modular non-blocking atomic commit 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 8, we will compare the performances of our protocol with those of non-blocking atomic protocol that were proposed in the literature so far, namely Skeen's Three Phase Commit protocols [28].

5.1 Background

A transaction originates at a process called the *transaction manager*, which issues read and write operations to *data manager* processes [4]. At the end of the transaction, the transaction manager, together with the data managers, must solve an *Atomic Commit* problem in order to decide on the *commit* or *abort* outcome of the transaction. We consider here the *"Non-blocking" Atomic Commit* problem (NB-AC for short) where correct processes must eventually decide despite failures [28]. 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 more accurately by the following properties:

NB-AC-Agreement. No two processes decide differently.

NB-AC-Termination. Every correct process eventually decides.

NB-AC-Validity. The decision must be *abort* if one process votes *no*, and the decision must be *commit* if all processes vote *yes* and no process is suspected.

It is important to notice that the NB-AC-Validity property enables an *abort* decision if any process is suspected. This condition actually defines the weak NB-AC problem [14]. The distinction between weak NB-AC and strong NB-AC problems is however irrelevant in the context of this paper (see [14] for more details).

5.2 NB-AC based on a consensus service

In the following, we show how a NB-AC protocol is derived from our consensus service framework (Sect. 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 Figure 2 represents the message $(tid, vote\text{-}request, data\text{-}managers)$ 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 (Sect. 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 (Sect. 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 non-suspected processes 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*. [3]

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

[3] Notice that, depending on the failure suspicions, it might occur that one server s_j starts 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.

Liveness of the NB-AC consensus filter. If the initiator is correct or some correct process $p_i \in clients$ has received the message $(cid, vote\text{-}request, clients)$ sent by the initiator, then by the properties of the reliable multicast, every correct client receives the message $(cid, vote\text{-}request, clients)$, 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 (remember that we assume a failure detector that satisfies the strong completeness property, see Sect 3.3). 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 processes vote *yes* and no process is ever suspected, and (2) the decision must be *abort* if one process votes *no*. Consider (1). If no client is ever suspected then *CallInitValue* waits 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 (Sect. 3.4), the decision can only be *commit*. Consider now (2). If one process votes *no*, then *CallInit* ensures that every 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*.[4]

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.

[4] We do not justify such a majority condition. Our goal is just to show how such a validity condition translates into a consensus filter. Notice however that such a majority condition assumes for every logical $data_i$ a majority of correct data manager replicas.

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 for a server s_j allows to adapt our generic framework to this specific atomic commitment problem (r stands for "*replication*", 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*.

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 and view synchrony

The generic construction of agreement protocols based on a consensus service has been illustrated in the previous section on the non-blocking atomic commitment problem. In this section we illustrate our approach on the *group membership problem*, and an extension of it known as *view synchrony*, or more accurately *view synchronous communication*. We do not focus on particular specifications of these problems, but we rather show how given a specification, a simple and original solution is obtained using our reduction to consensus.

6.1 Group membership

6.1.1 Background

Roughly speaking, the *group membership* problem consists for a group of processes to agree 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* [6] 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 [3]).

As there is no agreed on definition for the group membership problem, we follow the same approach as in [9]: we consider a problem specification which, although simple, is not trivial. As we point out later in this section, our modular construction could easily be used with other specifications of the problem.

As in [9], we restrict ourselves to the case where processes can only be excluded from a view (there is no join). A process can be excluded if it *wishes* to leave the view, if it crashes, or of it is suspected to have crashed. 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-Termination. If a process $p_k \in v_i(g)$ wishes to leave $v_i(g)$, then there is at least one correct process in $v_i(g)$ that eventually installs $v_{i+1}(g)$.

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 no process is suspected and $p_k \in v_i(g)$ is the only process wishing to leave $v_i(g)$, then if a process installs $v_{i+1}(g)$ we have $v_{i+1}(g) = v_i(g) \setminus \{p_k\}$.

6.1.2 Group membership based on a consensus service

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

The initiator. If a process p_k suspects some process in $v_i(g)$ or p_k wishes to leave $v_i(g)$, then p_k reliably multicasts the message $(cid, view\text{-}change, v_i(g))$ 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 the current view number of process p_k.

The clients. Upon reception of the message sent by the initiator,[5] a client process "multisends" to the consensus servers either (1) the message $(cid, no, v_i(g))$ if it wishes to leave $v_i(g)$, or (2) the message $(cid, yes, v_i(g))$ otherwise. The decision computed by the consensus service is the new view $v_{i+1}(g)$.

GM-Agreement and GM-Termination follow from the consensus service (CS-Agreement and CS-Termination properties) and from the liveness of the GM consensus filter (see below).

The consensus filter. The GM-Validity property is ensured by the following GM consensus filter (filter of server s_j):

Predicate *GM-CallInitValue(cid)* :
    if received $(cid, -, clients)$ from one process in *clients*
      and 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 *GM-InitValue(dataReceived$_j$)* :
    return $\{p_k \mid (cid, yes, clients)\ from\ p_k\ \in dataReceived_j\}$.

We show now that the GM consensus filter is live and ensures the GM-Validity property.

Liveness of the GM consensus filter. The filter is live if there is at least one correct process in $v_i(g)$. Let p_k be a process that wishes to leave view $v_i(g)$. (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 $(cid, -, v_i(g))$ to the consensus servers. (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 (i.e., (1) and (2)), every correct consensus server receives at least one message $(cid, -, v_i(g))$. As we assume a failure detector that satisfies strong completeness and at least one process is correct in $v_i(g)$, then the consensus filter is live.

GM-Validity property is satisfied. Assume that no process is suspected. Then *GM-CallInitValue* waits for the message $(cid, -, clients)$ from every process in $v_i(g)$. If p_k is the only process wishing to leave $v_i(g)$, then p_k is the only process to send $(cid, no, clients)$. In this case, the function *GM-InitValue* ensures that every server starts consensus with the initial value $v_i(g) \setminus \{p_k\}$. By the Validity property of consensus, the new view $v_{i+1}(g)$ can only be $v_i(g) \setminus \{p_k\}$.
As pointed out earlier, alternative definitions of the group membership problem could be considered. The GM-Validity property above does not for instance prevent excluding all processes from a view (this can happen if these processes suspect each others). We

[5] 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.

could consider a variation of the GM-Validity property which always requires preserving a majority of processes within a view: the filter could be easily adapted to such a property and it will be live under the assumption that there always is a majority of correct processes within a view.

6.2 View synchronous communication

6.2.1 Background

View synchronous communication (in the context of the primary partition model) has been introduced by the Isis system [6], and later formalized in [26]. 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*, denoted by *VScast*.

As in Section 6.1, we restrict ourselves to the case where processes can only be excluded from a view. Let $v_i(g)$ be the current view of process p_k in g, and let p_k VScast message m inside group 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 $v_{i+1}(g)$ 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 [19].

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

VS-Termination. GM-Termination plus the following "message-view ordering" property: every process in $v_i(g)$ eventually VSdelivers m, or there is at least one correct process in $v_i(g)$ that installs $v_{i+1}(g)$.

VS-Agreement. Identical to GM-Agreement.

VS-Validity. GM-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 $v_i(g) \cap v_{i+1}(g)$ must VSdeliver m before installing $v_{i+1}(g)$.

The VS-Termination property requires a new view to be defined whenever VSdelivery of m by all processes in $v_i(g)$ cannot be ensured. The VS-Agreement property requires agreement on the views. The VS-Validity property, in addition to the GM-Validity property, requires the VSdelivery of messages with respect to the installation of new views.

6.2.2 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: message m is sent to all processes in $v_i(g)$. Upon reception of m, message m is VSdelivered. However, this is not enough to ensure the semantics of view synchronous communication. The implementation of VScast requires the notion of message *stability*. The predicate $stable_{k'}(m)$ holds at process $p_{k'} \in v_i(g)$ when process $p_{k'}$ knows that all the processes in $v_i(g)$ have received m.[6] Based on the notion of message stability, and assuming the current view $v_i(g)$, 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 $v_i(g)$, or (3) p_k has VSdelivered a message m and $stable_k(m)$ still does not hold at p_k after some time-out period following the VSdelivery of m by p_k, then p_k reliably multicasts the message $(cid, view\text{-}change, v_i(g))$ 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 $(cid, no, v_i(g))$ to the consensus service if p_k wishes to leave $v_i(g)$, or (2) the message $(cid, (unstable_k, yes), v_i(g))$, where $unstable_k$ denotes the set of messages unstable at p_k. The decision computed by the consensus service is a pair $(unstab, v)$, where $unstab$ is a set of messages and v a view. Upon reception of the decision $(unstab, v)$, every client process p_k first VSdelivers the messages in $unstab$ that it has not yet VSdelivered, and then installs the new view v.

The VS-Agreement property follows from the CS-Agreement property of the consensus service. For the VS-Termination property, there are two cases two consider. Case 1: no process $p_k \in v_i(g)$ wishes to leave $v_i(g)$ and every process in $v_i(g)$ VSdelivers m. Case 2: the other cases. In case 1, the VS-Termination trivially holds. In case 2, 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 $(unstab_j, v_j)$, where $unstab_j$ is a set of unstable messages, and v_j a set of processes. The decision computed by the consensus service is a pair $(unstab, v)$, where $unst$ 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 as follows:

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

[6] This can be implemented by having each process, upon reception of m, sending acknowledgements to all processes in $v_i(g)$.

Function *VS-InitValue(dataReceived$_j$)* :

$v_j \leftarrow \{p_k \mid (cid, (unstable_k, yes), clients) \in dataReceived_j\}$;

$unstab_j \leftarrow \{m \mid (cid, (unstable_k, yes), clients)$ *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 consensus filter (Sect. 6.1.2).

The VS-Validity property consists of two sub-properties: the GM-Validity property plus the additional "message-view ordering" property. For the proof of the GM-Validity property, refer to Section 6.1.2. 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 $(cid, (unstable_k, yes), clients)$ to the consensus service. There are two cases to consider: (1) $m \in unstab_k$, or (2) $m \notin unstab_k$. In case (1), $stable_k(m)$ holds, i.e., every process in $v_i(g)$ has received and VSdelivered m (before installing $v_{i+1}(g)$). So the additional "message-view ordering" property holds. In case (2), $m \in unstab$, where $(unstab, v)$ is the decision of the consensus service. Because every client process VSdelivers the messages in *unstab* that it has not yet VSdelivered before installing the new view, the additional "message-view ordering" property also holds.

7 Total order multicast/broadcast

7.1 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 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 for total order multicast, which allows issuing messages to overlapping destinations (we discuss these differences in detail in [18]).

Most of total order algorithms that were proposed in the literature are total order broadcast algorithms, and many of them rely on a membership service or view synchronous communication (e.g. [6, 1, 11]). 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. Such an algorithm for total order broadcast has been described in [8], where agreement is on a *set of messages*. This algorithm can be expressed in our generic communication scheme with an empty consensus filter. [7] In the following, we illustrate our generic solution on a total order *multicast* algorithm.

7.2 Total Order Multicast

7.2.1 Background

The algorithm we describe is an extension of a non-fault-tolerant total order multicast algorithm proposed by Skeen [29]. Basically, we show how to make that protocol fault-tolerant by using our consensus service.

We denote by $TO\text{-}multicast(m, Dst(m))$ the event by which a process multicasts message m according to total order multicast semantics, and $TO\text{-}deliver(m)$ the corresponding delivery event. The basic idea of Skeen's algorithm consists in 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 Lamport's logical clocks [20]. More precisely, when $TO\text{-}multicast(m, Dst(m))$ is executed by p_i, process p_i sends the message m to all processes in $Dst(m)$ and collects the timestamps of the $receive(m)$ events from 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.

We consider here the problem of agreeing on the sequence number $sn(m)$ in spite of process crashes. Inferring the order from that value is not discussed here: it can be found in [17] and [8]. Given a message m, and $TO\text{-}multicast(m, Dst(m))$, the computation of $sn(m)$ is defined by the following properties:

SN-Termination. If a correct process TO-multicasts a message 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.

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

7.2.2 Computing $sn(m)$ using a consensus service

Let $TO\text{-}multicast(m, Dst(m))$ 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 $Dst(m)$.

[7] This actually leads to a degenerated version of our generic communication scheme.

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 non-suspected 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 [17]. It is worthwhile to point out here that the protocol is correct under the assumption of a *perfect* failure detector: besides the strong completeness property (every process that crashes is eventually suspected by every correct process), the failure detector also needs to satisfy the *strong accuracy property*, i.e., no process is suspected before it crashes [8]. 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 [18] that in order to tolerate even a single crash failure, any genuine total order multicast protocol requires a perfect failure detector. Overcoming the need for a perfect failure detector in specific models is discussed in [17] and in [18].

8 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 lead to a loss of efficiency. On the contrary, our modular architecture enables interesting optimizations.

Up to know, we have considered consensus as a black box. To discuss 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.

8.1 Centralized algorithm

We consider the (centralized) consensus algorithm presented by Chandra and Toueg [8], 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. Hence, client processes need not send their messages to all consensus servers. It is sufficient that they send their messages to one server, unless they suspect this server to have crashed. In the following, we consider the $\diamond\mathcal{S}$-consensus 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 1 communication step and $O(n)$ messages in good runs.

Let n_c be the number of clients, and n_s the number of servers. Figure 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):

- 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 costs $n_s - 1$ messages (see Fig. 7);
- step 5, the multisend initiated by the server s_1 to the clients costs n_c messages.

8.2 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 Figure 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;

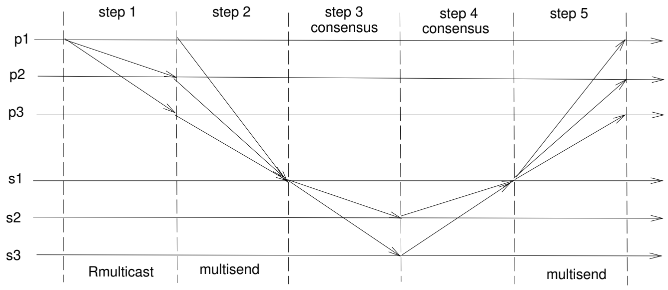

Figure 7: Centralized scheme (in good runs): p_1 is the initiator; p_1, p_2, p_3 are the clients; s_1, s_2, s_3 implement the consensus service.

- 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 in not depicted in Figure 8 (we give a detailed description in [16] for the case of atomic commit).

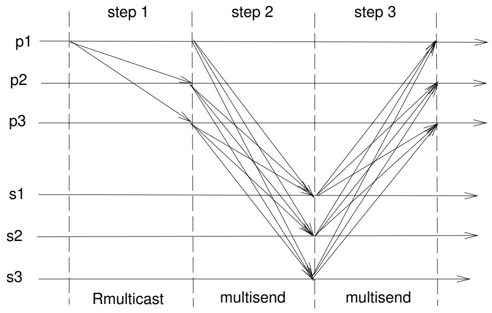

Figure 8: Decentralized scheme (in good runs):p_1 is the initiator; p_1, p_2, p_3 are the clients; s_1, s_2, s_3 implement the consensus service.

Despite the fact that, whenever $n_s \geq 2$, the number of messages is higher in Figure 8 than in Figure 7, reducing the number of communication steps from 5 to 3 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.

8.3 Comparison with Three Phase Commit

We compare below the performances of a Non-Blocking Commit Protocol built following our approach with those of Skeen's well known Three Phase Commit protocols (3PC) [28] (these are the only non-blocking atomic commit protocols we know about).

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 [28]. 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) [28]

Our solution based on a consensus service is however more modular, and in both cases (centralized or decentralized) allows to trade the number of messages against resilience. If we denote by n_c the number of clients and n_s the number of servers, 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 less messages than 3PC, and our decentralized solution requires less messages than D3PC. For instance, our centralized solution requires $3n_c + 2n_s - 3$ messages, whereas the 3PC requires $5n_c - 3$ 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 [16], we present experimental results confirming that an optimized consensus-based NB-AC protocol is more efficient that a 3PC protocol.

9 Concluding remarks

The paper advocates the idea that consensus is a central abstraction for building fault-tolerant agreement protocols in a modular way: The paper has presented 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.

To the best of our knowledge, this is the first time that solutions to the "group membership" and "virtual synchrony" problems are given based on a reduction to consensus, with well defined conditions under which liveness is ensured. Additionally, our framework suggests that, in the context of these two problems, the real open issue is the specification, rather than the implementation. Our framework can thus be viewed as a first step towards 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 [27, 30], 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 [24] or terminating reliable broadcast [8].

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 [22]. However, the generic interaction and the consensus filter, would remain the same. An interesting question in this context is to which extend the assumptions on the underlying system model impacts the performances. It is not clear for example whether assuming a completely synchronous model would lead to better performances.

References

[1] Y. Amir, L. Moser, P. Melliar Smith, D. Agarwal and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. ACM Transactions on Computer Systems, 13(4), pages 311-342, November 1995.

[2] A. Basu, B. Charron-Bost and S. Toueg. Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. Distributed Algorithms (WDAG'96), pages 105-122, LNCS 1151, Springer Verlag, October 1996.

[3] O. Babaoğlu, R. Davoli and A. Montresor. Failure Detectors, GroupMembership and View-Synchronous Communication in Partitionable Systems. *Technical Report, University of Bologna, Computer Science Department.* November 1995.

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

[5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. ACM Transactions on Computer Systems, 5(1), pages 47-76, February 1987.

[6] K. Birman, A. Schiper and P. Stephenson. Lightweight Causal and Atomic Group Multicast. ACM Transactions on Computer Systems, 9(3), pages 272-314, August 1991.

[7] T. Chandra, V. Hadzilacos and S. Toueg. The weakest failure detector for solving consensus. Journal of the ACM, 43(4), pages 685-722, July 1996. A preliminary version appeared in ACM Symposium on Principles of Distributed Computing, pages 147-159. ACM Press. August 1992.

[8] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 34(1), pages 225-267, March 1996. A preliminary version appeared in ACM Symposium on Principles of Distributed Computing, pages 325-340. ACM Press. August 1991.

[9] T. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost. On the Impossibility of group membership. Technical-Report 95-1548. Department of Computer Science. Cornell University, October 1995.

[10] D. Davcec and A. Burkhard. Consistency and Recovery Control for Replicated Files. ACM Symposium on Operating Systems Principles, pages 87-96, 1985.

[11] D. Dolev, S. Kramer and D. Malkhi. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. IEEE Symposium on Fault-Tolerant Computing, pages 296-306, June 1993.

[12] P. Ezhilchelvan, R. Macedo and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. IEEE Conference on Distributed Computing Systems, pages 296-306, May 1997.

[13] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, (32), pages 374-382, April 1985.

[14] R. Guerraoui. Revisiting the relationship between Non Blocking Atomic Commitment and Consensus problems. Distributed Algorithms (WDAG'95), pages 87-100, LNCS 972, Springer Verlag, September 1995.

[15] R. Guerraoui and A. Schiper. *Transactional model vs Virtual Synchrony model: bridging the gap.* In Theory and Practice in Distributed Systems, Springer Verlag (LNCS 938), 1995, 121-132.

[16] R. Guerraoui, M. Larrea and A. Schiper. Reducing the cost for Non-Blocking in Atomic Commitment. IEEE Conference on Distributed Computing Systems, pages 692-697, May 1996.

[17] R. Guerraoui and A. Schiper. Total Order Multicast to Multiple Groups. Proceedings of the 17th IEEE Int. Conf. on Distributed Computing Systems, pages 578-585, May 1997.

[18] R. Guerraoui and A. Schiper. Genuine Atomic Multicast. Distributed Algorithms (WDAG'97), Springer Verlag (LNCS 1320): pp 141-154, September 1997.

[19] R. Guerraoui and A. Schiper. Software-Based Replication for Fault-Tolerance. IEEE Computer, 30(4), pages 68-74, April 97.

[20] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), pages 558-565, July 1978.

[21] S. Mullender (editor). Distributed Systems, ACM Press Publisher, 1993.

[22] R. Oliveira, R. Guerraoui and A. Schiper. Consensus in the Crash-Recovery Model. Technical Report 97/239, EPFL, August 1997.

[23] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. ACM Symposium on Principles of Distributed Computing, pages 341-352, August 1991.

[24] L. Sabel and K. Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical Report TR95-1488, Cornell Univ, 1995.

[25] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. Distributed Computing, 10(3), pages 149-157, March/April 1997.

[26] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. IEEE Conference on Distributed Computing Systems, pages 561-568, May 1993.

[27] F. Schneider. Paradigms for distributed programs. Distributed Systems-Methods and Tools for Specification, Lecture Notes in Computer Science, Vol. 190, Springer Verlag, 1985, 343-430.

[28] D. Skeen. NonBlocking Commit Protocols. ACM SIGMOD International Conference on Management of Data, pages 133-142. ACM Press, 1981.

[29] D. Skeen. Unpublished communication. February 1985. Referenced in [5].

[30] J. Turek and D. Shasha. The Many Faces of Consensus in Distributed Systems. IEEE Computer, 25(6), pages 8-17, June 1992.