Consensus Service: a modular approach for building agreement protocols in distributed systems *

Rachid Guerraoui André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract

This paper describes a consensus service and suggests its use for the construction of fault-tolerant agreement protocols. We show how to build agreement protocols, using a classical client-server interaction, where (1) the clients are the processes that must solve the agreement problem, and (2) the servers implement the consensus service. Using a generic notion, called consensus filter*, we illustrate our approach on non-blocking atomic commitment and on view synchronous multicast. The approach can trivially be used for total order broadcast. In addition of its modularity, our approach enables efficient implementations of the protocols, and precise characterization of their liveness.*

1 Introduction

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 non-blocking atomic commitment protocols, total order broadcast/multicast protocols. Usually, these protocols are considered separately, and do not rely on a common infrastructure.

A notable exception is the *group membership service* [14], used to implement total order multicast protocols [2, 12, 5, 6]. However, the group membership problem (solved by the membership service) is just another example of an *agreement* problem that arises in distributed systems. All the agreement problems (atomic commitment, total order, membership) are related to the abstract *consensus* problem [4, 17], and thus are subject, in asynchronous systems, to the Fischer-Lynch-Paterson impossibility result [7, 3][1]. In fact, 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. This is unsatisfactory, not only from a theoretical point of view, but also from a practical point of view: the user of a fault-tolerant system should know the conditions under which liveness is guaranteed.

To summarize, not only have the various agreement protocols been considered separately, but also, most of them lack a precise liveness characterization[2]. Thanks to the recent work of Chandra and Toueg on failure detectors, we now have a formalism that allows to define conditions under which the consensus problem is solvable in asynchronous distributed systems. This is not only of key importance from a theoretical point of view[3] but also from a system builder point of view: consensus stops being a *taboo* problem, which means that it is time to define a common infrastructure to solve all agreement-type problems that arise in distributed systems. This is precisely the objective of this paper.

The paper suggests the use of a *consensus service*, implemented by a set of *consensus server* processes[4], to build agreement protocols. We introduce the generic notion of *consensus filter* to customize the consensus service for specific agreement protocols. Building an agreement protocol leads to a client-server

*Research supported by the "Fonds national suisse" under contract number 21-43196.95.

[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.

[2] An exception is the total order broadcast protocol in [4].

[3] In [10] we present a systematic way to transform several agreements problems into consensus.

[4] The number of server processes depends on the desired reliability.

interaction, where (1) the clients are the processes that have to solve an 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.

The rest of the paper is structured as follows. Section 2 defines the system model, the consensus problem, and briefly recalls the result about the consensus problem established by Chandra and Toueg. Section 3 presents the "client/consensus-server" interaction scheme. A non-blocking atomic commitment protocol is used throughout the section to illustrate the interaction scheme. The Chandra-Toueg total order algorithm can trivially be implemented using our consensus service. Section 4 illustrates the use of the consensus service on another agreement problem: view synchronous multicast. Section 5 discusses implementation issues, presents a cost analysis, and gives experimental results. Section 6 concludes the paper.

2 System model and the consensus problem

2.1 System model

We consider an asynchronous distributed system composed of a finite set of processes $\Omega = \{p_1, p_2, \ldots, p_n\}$ completely connected through a set of channels. Processes may fail by crashing. We do not consider Byzantine failures. A correct process is a process that does not crash in an infinite run. Processes communicate using the following communication primitives:

- *send*(m) to p_j: sending of message m to process p_j, over a *reliable* channel. This primitive ensures that 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 correct (i.e. do not crash)[5].
- *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.

[5]Reliable channels can be implemented by retransmitting messages. This does not exclude network partitions, assuming that partitions are eventually repaired.

- *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* introduces a stronger semantics. Implementation of the *Rmulticast* primitive, and its cost, should be ignored for the moment. These issues are discussed in Section 5.

2.2 Consensus and failure detectors

In the consensus problem, defined over a set Π of processes ($\Pi \subseteq \Omega$), every process $p_i \in \Pi$ starts with an initial value v_i, and the processes have to decide on a common value v. Formally, the consensus is defined by the following three properties [4]:

Uniform-Agreement. No two processes decide differently[6].

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

Termination. Every correct process eventually decides.

Fischer, Lynch and Paterson have shown that there is no deterministic algorithm that solves consensus in asynchronous systems where processes are subject to even a single crash failure [7]. Later, Chandra and Toueg have shown that, by augmenting an asynchronous system with a failure detector (even unreliable, i.e. which makes false suspicions), consensus becomes solvable [4]. In this model, each process has access to a local failure detector module, which maintains a list of processes that it currently suspects to have crashed. In [4], Chandra and Toueg present a protocol (which we note $\diamond\mathcal{S}$-consensus) that solves the consensus problem given a majority of correct processes, and any failure detector of class $\diamond\mathcal{S}$. The class $\diamond\mathcal{S}$ is characterized by the following two properties: (1) *strong completeness*: eventually every process that crashes is permanently suspected by every correct process, and (2) *eventual weak accuracy*: eventually some correct process is never suspected by any correct process[7].

[6]By requiring *Uniform Agreement* (instead of *Agreement*), we consider here the *uniform consensus* problem. We have shown in [8] that consensus and uniform consensus are equivalent in asynchronous systems augmented with unreliable failure detectors.

[7]Failure suspicions can be implemented using time-outs. Time-outs ensure the strong completeness property. Eventual weak accuracy property can be ensured with high probability, if (1) the time-outs are adequately defined, and (2) possible link failures are eventually repaired.

3 The "client/consensus-service" interaction model

3.1 Overview

In this section we describe how clients interact with the consensus service. Our consensus service consists of a set of *consensus server* processes $p_{s_1}, p_{s_2}, \ldots \in \Omega$, which execute a consensus protocol, e.g. the $\diamond\mathcal{S}$-consensus protocol. Practical issues in implementing such a consensus protocol are discussed in Section 5. In order to simplify the notation, p_{s_i} will be noted s_i.

The generic "client/consensus-service" interaction uses the *Rmulticast* and the *multisend* communication primitives defined in the previous section. We distinguish 3 steps (detailed in Sections 3.3, 3.4, and 3.5) in the client-server interaction:

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

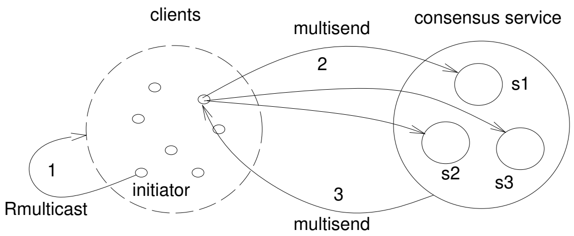

Figure 1: Invocation-reply for the point of view of a client

3.2 Illustration: non-blocking atomic commitment

Throughout the section we show how a non-blocking atomic commitment protocol can be built using our consensus service. Another example is given in Section 4.

A transaction originates at a process called the *Transaction Manager*, which issues read and write operations to *Data Manager* processes [1]. At the end of the transaction, the Transaction Manager and the Data Managers together execute a *non-blocking atomic commitment* [8] protocol (or NB-AC protocol for short) in order to decide on the *commit* or *abort* outcome of the transaction. The NB-AC protocol is initiated by the Transaction Manager, which sends a *vote-request* message to 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. Formally, the problem to be solved by a NB-AC protocol is defined by the following four properties. (i) *NB-AC-Uniform-Agreement:* no two processes decide differently. (ii) *NB-AC-Uniform-Validity:* the outcome of any process is *commit* only if all processes have voted *yes*. (iii) *NB-AC-Termination:* every correct process eventually decides. (iv) *NB-AC-Non-Triviality:* if all processes vote *yes*, and no process is ever suspected, the outcome must be *commit*[9].

3.3 The initiator

The invocation of the consensus service is started by an *initiator* process, which reliably multicasts the message $(cid, data, clients(cid))$ to the set $clients(cid)$ (Arrow 1 in Fig. 1). The parameter *cid* (*consensus id*) uniquely identifies the interaction with the consensus service, *data* contains some problem specific information illustrated below, and $clients(cid)$ is the set of clients that will invoke the consensus service:

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

Figure 2: Algorithm of the initiator

Example (initiator for NB-AC): Consider a transaction identified by an identifier *tid*, and the commitment of the transaction. Arrow 1 in Figure 1 represents the *vote-request* message sent by the Transaction Manager to the Data Managers: *cid* is the transaction identifier *tid*, the field *data* corresponds to *vote-request*, and $clients(cid)$ is the set of Data Managers accessed by the transaction (to simplify, we consider that the Transaction Manager is also a member of the set $clients(cid)$).

[8] "Non-blocking" means that correct processes must eventually decide despite failures [16].

[9] This condition actually defines the weak NB-AC problem [8]. The distinction between weak NB-AC and strong NB-AC problems is however irrelevant in the context of this paper.

3.4 Client-server interaction: the clients' point of view

Upon reception of the message $(cid, data, clients(cid))$ multicast by the initiator, a client process p_i computes $data'_i$, multisends the message $(cid, data'_i, clients(cid))$ to the consensus service, and waits for the decision of the consensus service:

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

Figure 3: Algorithm of a client process p_i

Example (client for NB-AC): The $data'_i$ value is the *yes/no* vote of the Data Manager p_i, and the decision awaited from the consensus service is either *commit* or *abort*.

3.5 Client-server interaction: the servers' point of view

The interaction between the clients and the consensus service is illustrated from the point of view of a server process in Figures 4 and 5. The genericity of the consensus service is obtained thanks to the notion of *"consensus filter"*, depicted in Figure 4 as a shaded ring. The consensus filter allows to tailor the consensus service to any particular agreement problem: the filter transforms the messages received by a server process s_j, into an initial value v_j for the consensus protocol.

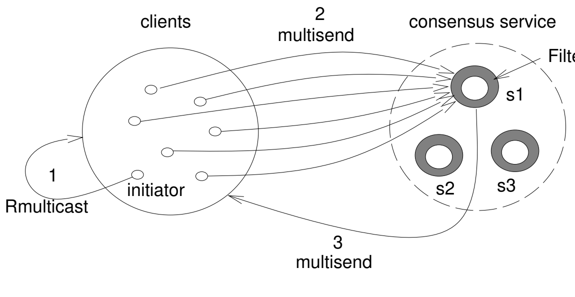

Figure 4: Invocation-reply from the point of view of server s_1 (arrows to and from s_2, s_3 have not been drawn)

A consensus filter is defined by two parameters, (1) a predicate $CallInitValue$, and (2) a function $InitValue$. The predicate $CallInitValue$ decides when the function $InitValue$ can be called. Once $CallInitValue$ is $true$, the function $InitValue$ is called, with the messages received as argument: the function returns the initial value for the consensus protocol (line 3, Fig. 5). At that point the consensus protocol is started. In Figure 5 (line 4), the consensus protocol is represented as a function $consensus$. The decision, once known, is multisent to the set $clients(cid)$.

```
1  receive messages (cid, data'_i, clients(cid)) from client p_i
   until CallInitValue ;
2  dataReceived_j ←
       {data'_i | message (cid, data'_i, clients(cid)) received} ;
3  v_j ← InitValue(dataReceived_j) ;
4  decision ← consensus(v_j) ;
5  multisend(cid, decision) to clients(cid) ;
```

Figure 5: Algorithm of a server s_j

Example (consensus filter for NB-AC): The consensus filter, given below, tailors the consensus service to build a NB-AC protocol. The filter is defined as follows. Consider a server s_j. The *NB-AC-CallInitValue* predicate follows the *NB-AC-Non-Triviality* condition (Sect. 3.2): the predicate at s_j returns $true$ as soon as, for every client process p_i, either (1) the message $(cid, vote_i, clients(cid))$ from p_i has been received, or (2) s_j suspects p_i. The function *NB-AC-InitValue* function follows the *NB-AC-Uniform-Validity* condition: the function returns *commit* if and only if a *yes* vote has been received by s_j from every process in $clients(cid)$, and *abort* otherwise[10].

```
Predicate NB-AC-CallInitValue :
    if [ for every process p_i ∈ clients(cid):
          received (cid, vote_i, clients(cid)) 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 ∈ clients(cid):
          (cid, vote_i, clients(cid)) in dataReceived_j
          and
          vote_i = yes ]
    then return commit else return abort.
```

3.6 Correctness of the interaction scheme

The generic "clients/consensus service" invocation scheme is live if it satisfies the following property: *If the initiator is correct, or if some correct*

[10] *commit/abort* are here initial values for the consensus, and not yet the decision of the consensus service.

client has received the $(cid, data, clients(cid))$ *message sent by the initiator, then every correct client in* $clients(cid)$*, eventually receives the decision message* $(cid, decision)$ *from the consensus service.* This property holds under the following three conditions:

C1. Liveness of the filter predicate ***CallInitValue*****.** If every correct process $p_i \in clients(cid)$ sends the message $(cid, data'_i, clients(cid))$ to the consensus service, then *CallInitValue* eventually returns $true$.

C2. Liveness of the consensus protocol. If every correct server s_j eventually starts the $consensus$ protocol, then the protocol eventually terminates (i.e. every correct server eventually decides).

C3. Correctness of some server. There is at least one correct server.

The conditions C1 to C3 ensure the liveness of the interaction scheme for the following reason. Assume that the initiator of the request *cid* is correct, or that a correct client has received the message $(cid, data, clients(cid))$ sent by the *initiator*. As the message is sent using a reliable multicast, every correct process in $clients(cid)$ eventually receives the message. Thus every correct process $p_i \in clients(cid)$ eventually multisends the message $(cid, data'_i, clients(cid))$ to the consensus service. By condition C1, for every correct server s_j, $CallInitValue$ eventually returns $true$. Hence, every correct server s_j eventually calls the $InitValue$ function, and starts the $consensus$ protocol. By condition C2, every correct server s_j eventually decides. The decision is returned to the clients using a reliable multisend. By condition C3, every correct process $p_i \in clients(cid)$ eventually gets the decision.

Example (correctness of the NB-AC protocol): We show first that the conditions C1, C2, C3 are satisfied by our generic consensus service and the *NB-AC* filter: this implies that the *NB-AC-Termination* property is satisfied, given that the Transaction Manager is correct, or that a correct Data Manager has received the *vote-request* message. If we assume reliable channels and a failure detector that satisfies *strong completeness*, then the *NB-AC-CallInitValue* predicate satisfies C1. Condition C2 is satisfied by the Chandra-Toueg $\diamond\mathcal{S}$-consensus protocol, given a majority of correct server processes [4]. A majority of correct server processes also ensures condition C3.

Consider now the other properties that define the NB-AC problem (Sect. 3.2). The *NB-AC-Uniform-Agreement* property follows directly from the *Uniform-Agreement* property of the consensus problem. The *NB-AC-Uniform-Validity* property is ensured by the *NB-AC-InitValue* function of the filter (the initial value for the consensus is *commit* only if all clients have voted *yes*), together with the *Uniform-Validity* property of consensus. The *NB-AC-Non-Triviality* property is ensured by the *NB-AC-CallInitValue* predicate of the filter, together with the *NB-AC-InitValue* function of the filter: if no client is ever suspected, and all clients vote *yes*, then every server starts the consensus with the initial value *commit*. In this case, by the *Uniform-Validity* property of consensus, the decision can only be *commit*.

In summary, assuming the consensus service is implemented with the $\diamond\mathcal{S}$-consensus protocol, the NB-AC protocol is correct if (a) a majority of server processes are correct, and (b) the failure detector is of class $\diamond\mathcal{S}$.

4 Solving agreement problems using the consensus service

The "client/consensus service" interaction has been illustrated in the previous section on a non-blocking atomic commitment protocol. Chandra and Toueg have shown in [4] how to build a total order broadcast protocol using a consensus algorithm. This "reduction" can be expressed using our consensus service in a straightforward way (empty filter). We give here another example that requires a non-empty filter.

4.1 View synchronous multicast

View synchronous multicast, initially introduced by the Isis system [2], can be seen as an atomic (in the sense of *all-or-nothing*) multicast for dynamic groups of processes. This primitive is adequate, for example, in the context of the primary-backup replication technique, to multicast the *update* message from the primary to the backups [11].

Consider a dynamic group g, i.e. a group whose membership changes during the life-time of the system, for example as the result of the crash of one of its members. A crashed process p_i is removed from the group; if p_i later recovers, then it rejoins the group (usually with a new identifier). The notion of *view* is used to model the evolving membership of a group. The initial membership of a group g is noted $v_0(g)$, and $v_k(g)$ is the k^{th} membership of g [11].

Within this context, view synchronous multicast is defined as follows. Consider a group g, and let $t_i(k)$ be the local time at which a process p_i delivers view $v_k(g)$, with $p_i \in v_k(g)$. From $t_i(k)$ on, and until the delivery of the next view $v_{k+1}(g)$, all multicasts m of p_i are sent to $v_k(g)$, and are time-stamped with the current view number k. For every multicast of such a message m, view synchronous multicast ensures that either (1) no new view $v_{k+1}(g)$ is ever defined and all the members of $v_k(g)$ eventually deliver m, or (2) a new view $v_{k+1}(g)$ is defined, and all the processes in $v_k(g) \cap v_{k+1}(g)$ deliver m before delivering the next view $v_{k+1}(g)$ [12].

4.2 Implementation of view synchronous multicast

Isis has implemented view synchronous multicast, based on a membership service [14], using a flush protocol [2]. As pointed out in [15], the flush protocol might lead, in certain circumstances, to violate the view synchronous multicast definition: [15] proposes a correct implementation of view synchronous multicast, also based on a membership service.

Using a consensus service, view synchronous multicast can be implemented without relying on a membership service. Given a group g, we describe the implementation of the case where members are removed from the current view of a group. The algorithm for adding new members to the view is very similar.

Implementation of view synchronous multicast consists in launching multiple, independent, instances of consensus, identified by an integer k. This is similar to the Chandra-Toueg total order broadcast algorithm [4]. However, consensus number k decides here not only on a batch of messages $batch(k)$ (as in Chandra-Toueg's algorithm), but also on the membership for the next view $v_{k+1}(g)$. Each process p_i, after learning the decision of consensus number k, first delivers the messages of $batch(k)$ that it has not yet delivered, and then delivers the next view $v_{k+1}(g)$.

Consider a group g and its current view $v_k(g)$. The decision to launch consensus number k is related to the *stability* of the messages multicast within view $v_k(g)$. Let m be a message multicast to the view $v_k(g)$ and received by p_i: the local predicate $stable_i(m)$ is true if and only if process p_i knows that every process p_j in $v_k(g)$ has received m (and will thus eventually deliver m unless p_j crashes). Therefore, whenever some process $p_j \in v_k(g)$ has received a message m, and $stable_i(m)$ does not hold after some time-out period following the reception of m, then p_j becomes the initiator for consensus number k (note that there can be more than one initiator for consensus number k). The generic interaction with the consensus service described in Section 3 is instantiated as follows, to implement view synchronous multicast:

- The parameter cid is the pair (gid, k), where gid is the identifier of group g, and k the current view number. The set $clients(cid)$ is the set $v_k(g)$.
- The initiator reliably multicasts $(cid, req, clients(cid))$ to the set $clients(cid)$, where req is the request for a view change ($req = data$ in Fig. 2).
- Upon reception of $M \equiv (cid, req, clients(cid))$, a client p_i defines $data'_i$ as the set $unstable_i$ of messages received by p_i that are not stable. The client process p_i then multisends $(cid, unstable_i, clients(cid))$ to the consensus servers[13].
- The servers' VS-filter is defined as follows. The *VS-CallInitValue* predicate returns $true$ as soon as the message $(cid, unstable_i, clients(cid))$ has been received from a majority of $clients(cid)$ and from all non-suspected processes in $clients(cid)$. The majority requirement is related to the usual assumption that every view contains a majority of correct processes.

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

The function *VS-InitValue* returns a pair $(batch(k), v_{k+1}(g)$, where $batch(k)$ is the union of the sets $unstable_i$ received, and $v_{k+1}(g)$ is the set of clients p_i such that $unstable_i$ has been received. The set $v_{k+1}(g)$ is the proposal for the membership of the next view.

Function *VS-InitValue($dataReceived_j$)* :
  $batch(k) \leftarrow dataReceived_j$;
  $v_{k+1}(g) \leftarrow \{p_i \mid unstable_i \in dataReceived_j\}$;
  return $(batch(k), v_{k+1}(g))$.

[11] This unique sequence defines what has been called a "linear membership".

[12] "Extended virtual synchrony" [13] extends this property to the non "linear membership" model.

[13] When using view synchronous multicast to implement the primary-backup replication technique, the set $unstable_i$ contains at most one message [11].

The pair $(batch(k), v_{k+1}(g))$ returned by the *VS-InitValue* function is the initial value of server s_j for the consensus, and not yet the decision of the consensus service. For every server s_j, the initial value of s_j is such that $v_{k+1}(g)$ contains a majority of processes of $v_k(g)$, and all the non-suspected processes. Thus a client process p_i that has not been suspected by any server s_j is necessarily member of the next view $v_{k+1}(g)$. Moreover, if a client process p_i is member of the next view $v_{k+1}(g)$, then p_i's set $unstable_i$ is included in $batch(k)$. As every process in $v_{k+1}(g)$, before delivering $v_{k+1}(g)$, delivers the messages in $batch(k)$ that it has not yet delivered, the view synchronous multicast property is ensured.

Liveness of the implementation of view synchronous multicast is ensured by the generic conditions C1, C2, C3 (Sect. 3.6). Condition C1 is satisfied if we assume a majority of correct processes in $v_k(g)$, reliable channels, and a failure detector that satisfies *strong completeness*. Condition C2 is satisfied with the $\diamond\mathcal{S}$-consensus protocol, assuming a failure detector of class $\diamond\mathcal{S}$, and a majority of correct server processes [4]. A majority of correct server processes satisfies condition C3.

5 Implementation issues

This section discusses implementation issues related to the invocation scheme of Section 3. Our aim is to show that the generality of the consensus-server approach does not imply a loss of efficiency.

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

5.1 Reducing the number of messages

We start be showing that a simple optimization of the reliable multicast and multisend primitives, leads to reduce the number of messages in good runs.

The implementation assumes a failure detector that satisfies *strong completeness*: every crashed process is eventually suspected forever[14].

5.1.1 Reliable multicast primitive

The reliable multicast primitive *Rmulticast*(m) to *Dst*(m) is usually implemented as follows (see for example [4]):

- **for each** $p_i \in Dst(m)$: **if** p_i receives m **then**
  [**for each** $p_j \in Dst(m)$: $send(m)$ to p_j]

where "$send(m)$ to p_j" sends m to p_j over a reliable channel. In this implementation, every $p_i \in Dst(m)$ relays m. This obviously costs $O(n^2)$ messages, where $n = |Dst(m)|$. However, if the process executing "*Rmulticast*(m) to *Dst*(m)" is correct, there is no need to relay m. This leads to the following optimized implementation, that costs only $O(n)$ messages in good runs:

- **for each** $p_i \in Dst(m)$:
  if (p_i has received m from p_k) **and** (p_i suspects p_k)
  then [**for each** $p_j \in Dst(m)$: $send(m)$ to p_j]

To be complete, this specification requires some notion of *termination*:

- **for each** $p_i \in Dst(m)$:
  if (p_i has received m from p_k) **and** (p_i suspects p_k)
  and (Rmulticast of m is not terminated)
  then [**for each** $p_j \in Dst(m)$: $send(m)$ to p_j]

A discussion of *termination* is out of the scope of this paper.

5.1.2 Multisend primitive

The multisend primitive, used (1) by the client processes to invoke the consensus service, and (2) by the consensus service to send the decision back to the clients, has been introduced as a convenient notation. In case (2) it is really needed that every client receives the decision from the consensus service. Case (1) however may be interestingly optimized.

While in the original description of the $\diamond\mathcal{S}$-consensus protocol [4], every process p_j starts with an initial value v_j, there is in fact no need for all processes to start with an initial value: it is sufficient if one correct process starts with an initial value[15]. In the following, we consider the $\diamond\mathcal{S}$-consensus protocol with this optimization. In other words, when invoking the consensus service, it is sufficient that one correct member of the consensus service has an initial value. This leads us to introduce a *multisend-to-all* and a *multisend-to-one* primitive:

- *multisend-to-all* is the *multisend* introduced so far, and used by the consensus service to send back the decision to the clients. If $n = |Dst(m)|$, the primitive "*multisend-to-all*(m) to *Dst*(m)" clearly costs $O(n)$ messages;

[14] Using time-outs to generate suspicions leads to satisfy Strong completeness (Sect. 2).

[15] In every round of the $\diamond\mathcal{S}$-consensus protocol, the coordinator defines its estimate according to the estimates received from the participants. This can be omitted in the first round: the coordinator can define its estimate to be its initial value. We assume this trivial optimization here.

- *multisend-to-one* sends the message m only to one process p_j in $Dst(m)$. Only when p_j is suspected, then m is sent to the other processes. The primitive "*multisend-to-one*(m) to $Dst(m)$" can be used by the clients to invoke the consensus service. This primitive only costs one message in good runs. Consequently, using *multisend-to-one* instead of *multisend-to-all* when invoking the consensus service, reduces the number of messages in good runs.

We introduce the following notation to formally define *multisend-to-one*. Assume that processes in the system are ordered by their *id*. Given $Dst(m)$, we define $first(Dst(m))$ to be the process in $Dst(m)$ with the smallest *id*, and $rest(Dst(m))$ to be $Dst(m) \setminus \{first(Dst(m))\}$. The primitive "*multisend-to-one*(m) to $Dst(m)$", executed by a process p_i, is defined as follows:

- *send*(m) to $first(Dst(m))$;
  if p_i suspects $first(Dst(m))$
  then *multisend-to-one*(m) to $rest(Dst(m))$.

To be complete, this specification also requires some notion of *termination*, not discussed here. The optimized *multisend-to-one* primitive obviously costs only one message in good runs.

5.1.3 Cost analysis

Given the former optimizations, we now analyze the overall cost of the consensus service invocation scheme in good runs. When counting the number of messages, we only consider the messages needed in the protocol for the clients to receive the decision from the consensus service. This allows us to compare the consensus server invocation scheme with other known protocols, such as commit protocols [16, 1]. Specifically, this means that we do not count the messages needed to implement failure suspicions (these messages are not counted either in the cost analysis of commit protocols).

Let n_c be the number of clients, and n_s the number of servers. Messages are shown on Figure 6, which illustrates the 5 communication steps needed for the clients to receive the decision:

- step 1, the reliable multicast from the initiator to the set of clients, costs $n_c - 1$ messages;
- step 2, the *multisend-to-one* initiated by each of the clients to one of the consensus servers, costs n_c messages;
- steps 3 and 4 correspond to messages sent by 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. 6);
- step 5, the *multisend-to-all* initiated by the server s_1 to the clients, costs n_c messages.

This gives a total of $3n_c + 2n_s - 3$ messages.

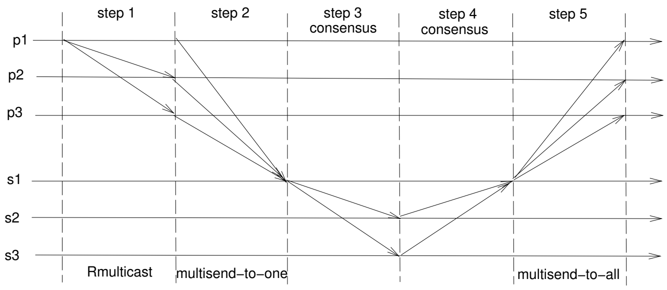

Figure 6: Invocation scheme: messages sent in good runs (p_1 is the initiator; p_1, p_2, p_3 are the clients; s_1, s_2, s_3 implement the consensus service).

5.2 Reducing the number of communication steps

In the preceding section we were concerned by the implementation of the multicast and multisend communication primitives, in order to reduce the number of messages sent during good runs. Other optimizations, aiming at reducing the latency of the overall invocation scheme, are also possible. We present one of these optimizations.

The optimization takes advantage of the following property of the consensus problem. 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 call "trivial consensus" this specific configuration. Consider for example non-blocking atomic commitment. In most of the cases, all the Data Managers vote *yes*. In those cases, and assuming that no failure occur, and no failure suspicion is generated during the run, the NB-AC-filter (see Sect. 3.5) transforms the non-blocking atomic commitment problem into a trivial consensus. A trivial consensus can also be obtained in the case of view synchronous multicast (Section 4).

A trivial consensus can be solved by the following interaction scheme (see Figure 7):

- step 1, as before, is the reliable multicast from the initiator to the set of clients;

- in step 2, the clients use the *multisend-to-all* primitive instead of the *multisend-to-one* primitive. Consequently, every member of the consensus service gets an initial value.
- in step 3, the consensus servers do not launch the consensus protocol, but simply multisend (*multisend-to-all*) their initial value to the clients. 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 consensus problem is not yet solved. This case in not depicted in Figure 7. The complete protocol to handle this case is given in [9].

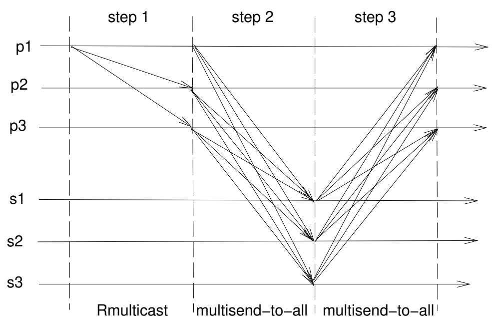

Figure 7: Trivial consensus: less communication steps (p_1 is the initiator; p_1, p_2, p_3 are the clients; s_1, s_2, s_3 implement the consensus service).

The above interaction scheme reduces the number of communication steps from 5 (Figure 6) to 3 (Figure 7).

5.3 Comparing the protocols

Despite the fact that the number of messages is higher in Figure 7 than in Figure 6, reducing the number of communication steps from 5 to 3 reduces the latency. This is shown in Figure 8, which gives the performances obtained on FDDI (100Mb/s) with SparcStations 20 (running Solaris 2.3), using the UDP transport protocol. The latency is given for $n_s = 3$, different values of n_c, and for both point-to-point and broadcast communication (broadcast network). In the latter case, sending a message to n processes costs only one message.

The reader might notice that (in good runs), if the consensus service is implemented by the clients themselves (i.e. $n_c = n_s$ and $\forall i,\ p_i \equiv s_i$), the communication scheme depicted in Figure 6 is similar to the communication scheme of the 3PC protocol [16], and the communication scheme depicted in Figure 7, is similar to the communication scheme of the D3PC protocol (Distributed 3PC) [16]. Our generic consensus server solution however incorporates well defined liveness guarantees, and our generic invocation scheme, together with the $\Diamond\mathcal{S}$-consensus protocol, also applies to other agreement problems.

Moreover, our solution based on a consensus service is more modular, and in both cases (i.e. Figures 6 and 7) allows to trade off the number of messages against resilience: 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 generic solution in Figure 6 requires less messages than 3PC, and our solution illustrated by Figure 7 requires less messages than D3PC. For instance, the solution illustrated by Figure 6 requires $3n_c + 2n_s - 3$ messages, whereas the 3PC requires $5n_c - 3$ messages. In practice $n_s = 3$ probably achieves the desired 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 $n_c = 4$).

Protocol	n_c=3	n_c=6	n_c=9
Fig. 6 **n_s=3**	3.2 (1) 3.1 (2)	3.8 (1) 3.3 (2)	4.3 (1) 3.5 (2)
Fig. 7 **n_s=3**	2.3 (1) 2.4 (2)	2.9 (1) 2.8 (2)	3.5 (1) 3.1 (2)
2PC	1.8 (1) 1.9 (2)	2.4 (1) 2.2 (2)	3.4 (1) 2.4 (2)
3PC	3.1 (1) 3.1 (2)	4.3 (1) 3.7 (2)	6.1 (1) 4.1 (2)

Figure 8: Performances in msec (1 = point-to-point communication; 2 = broadcast communication)

Figure 8 also gives the cost of the classical 2PC protocol (two phase commit), and of the 3PC protocol, among n_c processes. The comparison is interesting, as the 2PC is known to be more efficient than the 3PC protocol, but has the drawback of being blocking. Figure 8 shows that the "trivial consensus" interaction scheme allows to solve the *non-blocking atomic commitment* problem significantly faster than the 3PC protocol.

6 Conclusion

This paper can be viewed as a first step towards practical applications of recent work about comparing and solving agreement problems in asynchronous systems, augmented with unreliable failure detectors [4, 10]. The paper advocates the idea that *consen-*

sus is not only an interesting theoretical problem, but can also be viewed as a basic block for building fault-tolerant agreement protocols. Thanks to the notion of consensus filter, the generic consensus service can be tailored to build various agreement protocols. This consensus service approach is attractive, because it leads to a generic construction of agreement protocols. Moreover, by using the recent results on failure detectors for solving the consensus problem, the approach leads to a rigorous characterization of the liveness of the agreement protocols. Finally, the modularity of the consensus service interaction scheme allows interesting optimizations, at the level of the communication primitives, as well as at the level of the consensus protocol. The overall conclusion is that, in the context of agreement protocols, (1) genericity, (2) precise liveness properties, and (3) efficient implementation, are not incompatible.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison-Welsey, 1987.

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.

[3] T.D. 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.

[4] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 95-1535, Department of Computer Science, Cornell University, August 1995. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[5] D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *IEEE 23rd Int Symp on Fault-Tolerant Computing (FTCS-23)*, pages 544–553, June 1993.

[6] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In *IEEE 15th Intl. Conf. Distributed Computing Systems*, pages 296–306, May 1995.

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

[8] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *9th Intl. Workshop on Distributed Algorithms (WDAG-9)*, pages 87–100. Springer Verlag, LNCS 972, September 1995.

[9] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for Non-Blocking in Atomic Commitment. In *IEEE 16th Intl. Conf. Distributed Computing Systems*, May 1996. To appear.

[10] R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony Model: bridging the gap. In *Theory and Practice in Distributed Systems*, pages 121–132. Springer Verlag, LNCS 938, 1995.

[11] R. Guerraoui and A. Schiper. Fault-Tolerance by Replication in Distributed Systems. In *Proc Conference on Reliable Software Technologies (invited paper)*. Springer Verlag, June 1996. To appear.

[12] P. M. Melliar-Smith, L. E. Moser, and D. A. Agarwal. Ring-based ordering protocols. In *Proceedings of the Int. Conference on Information Engineering*, pages 882–891, December 1991.

[13] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended Virtual Synchrony. In *IEEE 14th Intl. Conf. Distributed Computing Systems*, pages 56–67, June 1994.

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

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

[16] D. Skeen. Nonblocking Commit Protocols. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 133–142, 1981.

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