A Dynamic Primary Configuration Group Communication Service

Roberto De Prisco[1], Alan Fekete[2], Nancy Lynch[3], and Alex Shvartsman[4]

[1] MIT Laboratory for Computer Science, Cambridge, MA 02139, USA and
Dip. di Informatica ed Applicazioni, University of Salerno, Italy.
robdep@theory.lcs.mit.edu
[2] Basser Dept. of Computer Science, University of Sydney, NSW 2006, Australia
fekete@theory.lcs.mit.edu
[3] MIT Laboratory for Computer Science, Cambridge, MA 02139, USA.
lynch@theory.lcs.mit.edu
[4] Dept. of Computer Science and Eng., University of Connecticut, Storrs, CT, USA,
and MIT Laboratory for Computer Science, Cambridge, MA 02139, USA.
alex@theory.lcs.mit.edu

Abstract. Quorum-based methods for managing replicated data are popular because they provide availability of both reads and writes in the presence of faulty behavior by some sites or communication links. Over a very long time, it may become necessary to alter the quorum system, perhaps because some sites have failed permanently and others have joined the system, or perhaps because users want a different trade-off between read-availability and write-availability. There are subtle issues that arise in managing the change of quorums, including how to make sure that any operation using the new quorum system is aware of all information from operations that used an old quorum system, and how to allow concurrent attempts to alter the quorum system.
In this paper we use ideas from group management services, especially those providing a dynamic notion of primary view; with this we define an abstract specification of a system that presents each user with a consistent succession of identified *configurations*, each of which has a membership set, and a quorum system for that set. The key contribution here is the intersection property, that determines how the new configurations must relate to previous ones. We demonstrate that our proposed specification is neither too strong, by showing how it can be implemented, nor too weak, by showing the correctness of a replicated data management algorithm running above it.

1 Introduction

In distributed applications involving replicated data, a well known way to enhance the availability and efficiency of the system is to use *quorums*. A quorum is a subset of the members of the system, such that any two quorums have nonempty intersection. An update can be performed with only a quorum available, unlike other replication techniques where all of the members must be available.

The intersection property of quorums guarantees consistency. Quorum systems have been extensively studied and used in applications, e.g., [1, 7, 8, 18, 23, 24, 34, 38]. The use of quorums has been proven effective also against Byzantine failures [32, 33].

Pre-defined quorum sets can yield efficient implementations in settings which are relatively static, i.e., failures are transient. However they work less well in settings where processes routinely join and leave the system, or where the system can suffer multiple partitions. These settings require the on-going modification of the choice of quorums. For example, if more sites join the system, quorums must be reconfigured to make use of the new sites. If many sites fail permanently, quorums must be reconfigured to maintain availability. The most common proposal has been to use a two-phase commit protocol which stops all application operations while all sites are notified of the new configuration. Since two-phase commit is a blocking protocol, this solution is vulnerable to a single failure during the configuration change. In a setting of database transactions, [23] showed how to integrate fault-tolerant updates of replicated information about quorum sizes (using the same quorums for both data item replicas, and for quorum information replicas).

Here we offer a different approach, based on ideas of dynamic primary views from group management systems. *View-oriented group communication services* have become important as building blocks for fault-tolerant distributed systems. Such a service enables application processes located at different nodes of a fault-prone distributed network to operate collectively as a group, using the service to multicast messages to all members of the group. Each such service is based on a *group membership service*, which provides each group member with a *view* of the group; a view includes a list of the processes that are members of the group. Messages sent by a process in one view are delivered only to processes in the membership of that view, and only when they have the same view. Within each view, the service offers guarantees about the order and reliability of message delivery. Examples of view-oriented group communication services are found in Isis [9], Transis [15], Totem [37], Newtop [20], Relacs [3], Horus [46] and Ensemble [45].

For many applications, some views must be distinguished as *primary views*. Primary views have stronger properties, which allow updates to occur consistently. Traditionally, a primary view was defined as one containing a majority of all possible sites, but other, dynamic, definitions are possible, based on intersection properties between successive primary views. Several methods have been developed to define primary views adaptively, e.g., [6, 13, 14, 17, 27, 35, 41, 43, 47]. Producing good specifications for view-oriented group communication services is difficult, because these services can be complicated, and because different such services provide different guarantees about safety, performance, and fault-tolerance. Examples of specifications for group membership services and view-oriented group communication services appear in [4, 5, 10, 12, 16, 21, 22, 25, 26, 36, 39, 42, 44]. Extending these definitions to specify dynamic primary views was the focus of [14, 47].

In this paper we combine the notion of dynamic primary view with that of a quorum system, and call the result a *configuration*. We integrate this with a group communication service, resulting in a *dynamic primary configuration group communication service*. The main difficulty in combining quorum systems with the notion of dynamic primary view is the intersection property between quorums from different views, which is required to maintain consistency. With configurations the simple intersection property that works for primary views, is no longer enough. Indeed updated information might be only at a quorum and the processors in the intersection might be not in that quorum. A stronger intersection property is required. We propose one possible intersection property that allows applications to keep consistency across different primary configurations. Namely, we require that there be a quorum of the old primary configuration which is included in the membership set of the new primary configuration. This guarantees that there is at least one process in the new primary configuration that has the most up to date information. This, similarly to the intersection property of dynamic primary views, allows flow of information from the old configuration to the new one and thus permits one to preserve consistency.

The specific configurations we consider use two sets of quorums, a set of *read* quorums and a set of *write* quorums, with the property that any read quorum intersects any write quorum. (This choice is justified by the application we develop, an atomic read/write register.) With this kind of configuration the intersection property that we require for a new primary configuration is that there be one read quorum and one write quorum both of which are included in the membership set of the new primary configuration. The use of read and write quorums (as opposed to just quorums) can be more efficient in order to balance the load of the system (see for example [18]).

We provide a formal automaton specification, called DC for "dynamic configurations", for the safety guarantees made by a dynamic primary configuration group communication service. We remark that we do not address liveness properties here, but that they can be expressed as conditional performance properties, similar to those in [21], or with other techniques such as failure-detectors [11].

Clearly the DC specification provides support for primary configurations. However it also has another important feature, namely, it provides support for state-exchange. When a new configuration starts, applications generally require some pre-processing, such as an exchange of information, to prepare for ordinary computation. Typically this is needed in order to bring every member of the configuration up to date. For example, processes in a coherent database application may need to exchange information about previous updates in order to bring everyone in the new configuration up to date. We will refer to the up-to-date state of a new configuration as the *starting state* of that configuration. The starting state is the state of the computation that all members should have in order to perform regular computation. When the notification of a new configuration is given to its members, the DC specification allows these members to submit their current state. Then the service takes care of collecting all the states and computing the starting state for the new configuration and delivering it to the members.

When all members have been notified of the starting state for a configuration c, all information about the membership set and the quorums of previous configurations is not needed anymore, and the service no longer needs to ensure intersection in membership between configurations before c and any subsequent ones that are formed. This is the basis of a garbage-collection mechanism which was introduced in [47].

The DC specification offers a broadcast/convergecast communication service which works as follows: a process p submits a message to the service; the service forwards this message to the members of the current configuration and upon receiving acknowledgment values from a quorum of members it computes a response for the message sent by process p and gives the response to p. This communication mechanism has been introduced in [30], though in the setting of that paper there is no group-oriented computation.

We demonstrate the value of our DC specification by showing both how it can be implemented and how it can be used in an application. Both pieces are shown formally, with assertional proofs.

We implement DC by using a variant of the group membership algorithm of [47]. Our variant integrates communication with the membership service, provides state-exchange support at the beginning of a new configuration, and uses a static configuration-oriented service internally. We prove that this algorithm implements DC, in the sense of trace inclusion. The proof uses a simulation relation and invariant assertions.

We develop an atomic read/write shared register on top of DC. The algorithm is based on the work of Attiya, Bar-Noy and Dolev [2] and follows the approach used in [19, 30]. The application exploits the communication and state-exchange services provided by DC. The proof of correctness uses a simulation relation and invariant assertions.

2 Mathematical foundations and notation

We describe the services and algorithms using the the I/O automaton model of Lynch and Tuttle [31] (without fairness). The model and associated methodology is described in Chapter 8 of [29].

Next we provide some definitions used in the rest of the paper.

We write λ for the empty sequence. If a is a sequence, then $|a|$ denotes the length of a. If a is a sequence and $1 \leq i \leq |a|$ then $a(i)$ denotes the ith element of a.

Given a set S, $seqof(S)$ denotes the set consisting of all finite sequences of elements of S.

If s and t are sequences, the concatenation of s and t, with s coming first, is denoted by $s+t$. We say that sequence s is a *prefix* of sequence t, written as $s \leq t$, provided that there exists u such that $s+u = t$. The "head" of a sequence a is $a(1)$. A sequence can be used as a queue: the *append* operation modifies the sequence by concatenating the sequence with a new element and the *remove* operation modifies the sequence by deleting the head of the sequence.

If R is a binary relation, then we define $dom(R)$, the *domain* of R to be the set (without repetitions) of first elements of the ordered pairs comprising relation R.

We denote by $\mathcal{P}$ the universe of all processors[1] and we assume that $\mathcal{P}$ is totally ordered. We denote by $\mathcal{M}$ the universe of all possible messages. We denote by $\mathcal{G}$ a totally ordered set of identifiers used to distinguish configurations. Given a set S, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$. If a set S is totally ordered, we extend the ordering of S to the set $S_\perp$ by letting $\perp < s$ for any $s \in S$.

A *configuration* is a quadruple, $c = \langle g, P, \mathcal{R}, \mathcal{W} \rangle$, where $g \in \mathcal{G}$ is a unique identifier, $P \subseteq \mathcal{P}$ is a nonempty set of processors, and $\mathcal{R}$ and $\mathcal{W}$ are nonempty sets of nonempty subsets[2] of P, such that $R \cap W \neq \{\}$ for all $R \in \mathcal{R}$, $W \in \mathcal{W}$. Each element of $\mathcal{R}$ is called a *read quorum* of c, and each element of $\mathcal{W}$ a *write quorum*. We let $\mathcal{C}$ denote the set of all configurations.

Given a configuration $c = \langle g, P, \mathcal{R}, \mathcal{W} \rangle$, the notation $c.id$ refers to the configuration identifier g, the notation $c.set$ refers to the membership set P, while $c.rqrms$ and $c.wqrms$ refer to $\mathcal{R}$ and $\mathcal{W}$, respectively. We distinguish an initial configuration $c_0 = \langle g_0, P_0, \mathcal{R}_0, \mathcal{W}_0 \rangle$, where g_0 is a distinguished configuration identifier.

3 The DC specification

In many applications significant computation is performed only in special configurations called *primary configurations*, which satisfy certain intersection properties with previous primary configurations. In particular, we require that the membership set of a new primary configuration must include the members of at least one read quorum and one write quorum of the previous primary configuration. The DC specification provides to the client only configurations satisfying this property. This is similar to what the DVS service of [14] does for ordinary views.

An important feature of the DC specification is that it allows for state-exchange at the beginning of a new primary configuration. State-exchange at the beginning of a new configuration is required by most applications. When a new configuration is issued each member of the configuration is supposed to submit its current state to the service which, once obtained the state from all the members of the configuration computes the most up to date state over all the members, called the *starting state*, and delivers this state to each member. This way, each member begins regular computation in the new configuration knowing the starting state. We remark that this is different from the approach used by the DVS service of [14] which lets the members of the configuration compute

[1] In the rest of the paper we will use *processor* as synonymous of *process*. The differences between the two terms are immaterial in our setting.

[2] Expressing each quorum as a set of subsets is a generalisation of the common technique where the quorums are based on integers nr and nw such that $nr + nw \geq |P|$; the two approaches are related by defining the set of read quorums as consisting of those subsets of P with cardinality at least nr, and the set of write quorums as consisting of those subsets of P of cardinality at least nw.

the starting state. Some existing group communication services also integrate state-exchange within the service [43].

Finally, the DC specification offers a broadcast/convergecast communication mechanism. This mechanism involves all the members of a quorum, and uses a condenser function to process the information gathered from the quorum. More specifically, a client that wants to send a message (request) to the members of its current configuration submits the message together with a *condenser* function to the service; then the DC service broadcasts the message to all the members of the configuration and waits for a response from a quorum (the type of the quorum, read or write, is also specified by the client); once answers are received from a quorum, the DC service applies the condenser function to these answers in order to compute a response to give back to the client that sent the message.

We remark that this kind of communication is different from those of the VS service [21] and the DVS service [14]. Instead, it is as the one used in [30]. We integrate it into DC because we want to develop a particular application that benefits from this particular communication service (a read/write register as is done in [30]).

Prior to providing the code for the DC specification, we need some notation and definitions, which we introduce in the following while giving an informal description of the code.

Each operation requested by the client of the service is tagged with a unique identifier. Let OID be the set of operation identifiers, partitioned into sets OID_p, $p \in \mathcal{P}$. Let $\mathcal{A}$ be a set of "acknowledgment" values and let $\mathcal{R}$ be a set of "response" values. A *value condenser function* is a function from $(\mathcal{A}_\perp)^n$ to $\mathcal{R}$. Let Φ be the set of all value condenser functions. Let $\mathcal{S}$ be the set of states of the client (this does not need to be the entire client's state, but it may contain only the relevant information in order for the application to work). A *state condenser function* is a function from $(\mathcal{S}_\perp)^n$ to $\mathcal{S}$. Let Ψ be the set of all state condenser functions. Given a function $f : \mathcal{P} \to D$ from the set of processors $\mathcal{P}$ to some domain value D and given a subset $P \subseteq \mathcal{P}$ of processors we write $f|P$ to denote the function f' defined as follows: $f'(p) = f(p)$ if $p \in P$ and $f'(p) = \perp$ otherwise.

We use the following data type to describe an operation: $\mathcal{D} = \mathcal{M} \times \Phi \times \{\text{"}read\text{"}, \text{"}write\text{"}\} \times 2^{\mathcal{P}} \times (\mathcal{A}_\perp)^n \times Bool$ and we let $\mathcal{O} = OID \to \mathcal{D}_\perp$. Given an operation descriptor, selectors for the components are msg, cnd, sel, dlv, ack, and rsp. Given an operation descriptor $d \in \mathcal{D}$ for an operation i, $d.msg$ is the message of operation i which is delivered to all the processes (it represents the request of the operation, like read a register or write a register), $d.cnd$ is the condenser of operation i which is used to compute a response when acknowledgment values are available from a quorum of processes, $d.sel$ is a selector that specifies whether to use a read or a write quorum, $d.dlv$ is the set of processes to which the message has been delivered, $d.ack$ contains the acknowledgment values, and, finally, $d.rsp$ is a flag indicating whether or not the client has received a response for the operation. Operation descriptors maintain information about the operations. When an operation i is submitted its descriptor $d = pending[g](i)$ is initialized to $d = (m, \phi, b, \{\}, \{\}, \texttt{false})$ where m, ϕ and b come with the opera-

tion submission (i.e., are provided by the client). Then $d.dlv, d.ack$ and $d.rsp$ are updated while the operation is being serviced. Once a response has been given back to the client and thus $d.rsp$ is set to `true`, the operation is completed.

For each process p we define the current configuration of p as the last configuration c given to p with a NEWCONF$(c)_p$ event (or a predefined configuration if no such event has happened yet). The identifier of the current configuration of process p is stored into variable $cur\text{-}cid_p$. When a configuration c has been notified to a processor p we say that processor p has "attempted" configuration c. We use the history variable *attempted* to record the set of processors that have attempted a particular configuration c. More formally $p \in attempted[c.id]$ iff processor p has attempted c.

Next we define an important notion, the one of "dead" configuration. Informally a dead configuration c is a configuration for which a member process p went on to newer configurations, that is, it executed action NEWCONF$(c')_p$ with $c'.id > c.id$, before receiving the notification, that is the NEWCONF$(c)_p$ event, for configuration c (which can no longer be notified to that processor, and thus is dead because processor p cannot participate and it is impossible to compute the starting state). More formally we define $dead \in 2^{\mathcal{C}}$ as $dead = \{c \in \mathcal{C} | \exists p \in c.set : cur\text{-}cid_p > c.id \text{ and } p \notin attempted[c.id]\}$.

DC (Signature and state)

Signature:

Input:
SUBMIT$(m, \phi, b, i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi$,
  $b \in \{$"*read*", "*write*"$\}$, $p \in \mathcal{P}$,$i \in OID_p$
ACKDLVR$(a, i)_p$, $a \in A$, $i \in OID$, $p \in \mathcal{P}$
SUBMIT-STATE$(s, \psi)_p$, $s \in \mathcal{S}$, $\psi \in \psi$

Internal: CREATECONF(c), $c \in \mathcal{C}$
Output:
NEWCONF$(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
NEWSTATE$(s)_p$, $s \in \mathcal{S}$
RESPOND$(a, i)_p$, $a \in A$, $i \in OID_p$, $p \in \mathcal{P}$
DELIVER$(m, i)_p$, $m \in \mathcal{M}$, $i \in OID$, $p \in \mathcal{P}$

State:

$created \in 2^{\mathcal{C}}$, init $\{c_0\}$
for each $p \in \mathcal{P}$: $cur\text{-}cid[p] \in \mathcal{G}_\perp$,
  init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$: $attempted[g] \in 2^{\mathcal{P}}$,
  init P_0 if $g = g_0$, $\{\}$ else

for each $g \in \mathcal{G}$:
  $got\text{-}state[g] = \mathcal{P} \to \mathcal{S}_\perp$, init everywhere $\perp$
  $condenser[g] = \mathcal{P} \to \psi_\perp$, init everywhere $\perp$
  $state\text{-}dlv[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else
  $pending[g] \in \mathcal{O}$, init everywhere $\perp$

Fig. 1. The DC signature and state

We say that a configuration c is *totally attempted* in a state s of DC if $c.set \subseteq attempted[c.id]$. We denote by $\mathcal{T}ot\mathcal{A}tt$ the set of totally attempted configurations. Informally a totally attempted configuration is a configuration for which all members have received notification of the new configuration. Similarly, we say that a configuration c is *attempted* in a state s of DC if $attempted[c.id] \neq \{\}$. We denote by $\mathcal{A}tt$ the set of attempted configurations. Clearly $\mathcal{A}tt \subseteq \mathcal{T}ot\mathcal{A}tt$.

DC (Transitions)

Actions:

internal CREATECONF(c)
  Pre: For all $w \in created : c.id \neq w.id$
    if $c \notin dead$ then
      For all $w \in created, w.id < c.id$:
        $w \in dead$ or
        $(\exists x \in \mathcal{TOTEST}: w.id < x.id < c.id) \vee$
        $(\exists R \in w.rqrms, \exists W \in w.wqrms:$
          $R \cup W \subseteq c.set)$
      For all $w \in created, w.id > c.id$
        $w \in dead$ or
        $(\exists x \in \mathcal{TOTEST}: c.id < x.id < w.id) \vee$
        $(\exists R \in c.rqrms, \exists W \in c.wqrms:$
          $R \cup W \subseteq w.set)$
  Eff: $created := created \cup \{c\}$

output NEWCONF$(c)_p$, $p \in c.set$
  Pre: $c \in created$
    $c.id > cur\text{-}cid[p]$
  Eff: $cur\text{-}cid[p] := c.id$
    $attempted[c.id]$
      $:= attempted[c.id] \cup \{p\}$

input SUBMIT-STATE$(s, \psi)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ and
      $got\text{-}state[cur\text{-}cid[p]](p) = \perp$ then
      $got\text{-}state[cur\text{-}cid[p]](p) := s$
      $condenser[cur\text{-}cid[p]](p) := \psi$

output NEWSTATE$(s)_p$ *choose* c
  Pre: $c.id = cur\text{-}cid[p]$
    $c \in created$
    $\forall q \in c.set: got\text{-}state[c.id](q) \neq \perp$
    $s = condenser[c.id](p)(got\text{-}state[c.id])$
    $p \notin state\text{-}dlv[c.id]$
  Eff: $state\text{-}dlv[c.id]$
      $:= state\text{-}dlv[c.id] \cup \{p\}$

input SUBMIT$(m, \phi, b, i)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ then
      $pending[cur\text{-}cid[p]](i)$
        $:= (m, \phi, b, \{\}, \{\}, \texttt{false})$

output DELIVER$(m, i)_p$ *choose* g
  Pre: $g = cur\text{-}cid[p]$
    $p \notin pending[g](i).dlv$
    $pending[g](i).msg = m$
  Eff: $pending[g](i).dlv$
      $:= pending[g](i).dlv \cup \{p\}$

input ACKDLVR$(a, i)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ and
    $pending[cur\text{-}cid[p]](i).ack(p) \neq \perp$
    then
      $pending[cur\text{-}cid[p]](i).ack(p)$
        $:= a$

output RESPOND$(r, i)_p$ *choose* c, Q
  Pre: $c.id = cur\text{-}cid[p]$
    $c \in created$
    $i \in OID_p$
    $pending[c.id](i).rsp = \texttt{false}$
    if $pending[c.id].sel =$ "*read*"
      then $Q \in c.rqrms$
    if $pending[c.id].sel =$ "*write*"
      then $Q \in c.wqrms$
    let $f = pending[c.id](i).ack$
    $\forall q \in Q : f(q) \neq \perp$
    $r = pending[c.id](i).cnd(f|Q)$
  Eff: $pending[c.id](i).rsp := \texttt{true}$

Fig. 2. The DC transitions

After a processor p has attempted a new configuration, it submits its state by means of action SUBMIT-STATE$(s, \psi)_p$. Variable *got-state*$[g](p)$ records the state s submitted by processor p for the current configuration of p whose identifier is g. Similarly, the state condenser function submitted by p is recorded into variable *condenser*$[g](p)$. After all processors members of a configuration c have submitted their state, the starting state for c can be computed, by using the appropriate condenser function, and can be given to the members of c. Note that the state condensor is used when all members have submitted a state, in contrast to message convergecast which applies the value condensor once a quorum of values are known. Variable *state-dlv*$[g]$ records the set of processors to which the starting state for the configuration with identifier g has been delivered.

When the starting state for a configuration c has been delivered to processor p we say that c is *established* (at p). A configuration is totally established when it is established at all processors members of the configuration. More formally a configuration c is *totally established* in a state s of DC if, in state s, we have $c.set \subseteq$ *state-dlv*$[c.id]$. We denote by $\mathcal{TotEst}$ the set of totally established configurations. When a configuration c becomes totally established, information about the membership set and quorums of configurations previous to c can be discarded, because the intersection property will be guaranteed between c and later configurations.

The code of the DC specification is given in Figure 1, 2.

The second precondition of CREATECONF(c) is the key to our specification. It states that when a configuration c is created it must either be already dead or for any other configuration w such that there are no intervening totally established configurations, the earlier configuration (i.e., the one with smaller identifier) has one read quorum and one write quorum whose members are included in the membership set of the later configuration (i.e., the one with bigger identifier). The above precondition is formalized in the following key invariant:

Invariant 1 *Let $c_1, c_2 \in created \setminus dead$, with $c_1.id < c_2.id$. Then either exists $w \in \mathcal{TotEst}, c_1.id < w.id < c_2.id$, or else exist R, W quorums of c_1 such that $R \cup W \subseteq c_2.set$*

The property stated by this invariant is used to prove correct the application that we build on top of DC. We remark that dead configurations are excluded, that is, the intersection property may not hold for dead configurations. However, in a dead configuration it is not possible to make progress because for such a configuration there is at least one process that will not participate and thus the configuration will never become established.

The need for considering dead configurations comes from the implementation of the specification that we provide. It is possible to give a stronger version of DC by requiring that the intersection property in the precondition of action CREATECONF holds also for dead configurations, however this stronger version might not be implementable. Moreover, as we have said above, there is no loss of generality since no progress is made anyway in dead configurations.

4 An implementation of DC

The DC specification can be implemented, in the sense of trace inclusion, with an algorithm similar to that used in [14] to implement the DVS service. Hence it thus uses ideas from [47]. This implementation consists of an automaton DC-CODE$_p$ for each $p \in \mathcal{P}$. Due to space constraints we omit the code and the proof of correctness and provide only an overall description.

4.1 The implementation

The automaton DC-CODE$_p$ uses special messages, tagged either with "*info*", used to send information about the active and ambiguous configurations, or with "*got-state*", used to send the state submitted by a process to all the members of the configuration. The former information is needed to check the intersection property that new primary configurations have to satisfy according to the DC specification. The latter information is needed in order to compute the starting state for a new configuration.

The major problem is that the DC specification requires a global intersection property (i.e., a property that can be checked only by someone that knows the entire system state), while each single process has a local knowledge of the system. So, in order to guarantee that a new configuration satisfies the requirement of DC, each single process needs information from other processes members of the configuration.

Informally, the filtering of configurations works as follows. Each process keeps track of the latest totally established configuration, called the "active" configuration, recorded into variable act, and a set of "ambiguous" configurations, recorded into variable amb, which are those configurations that were notified after the active configuration but did not become established yet. We define $use = act \cup amb$. When a new configuration is detected, process p sends out an "*info*" message containing its current act_p and amb_p values to all other processors in the new configuration, using an underlying broadcast communication mechanism, and waits to receive the corresponding "*info*" messages for configuration c from all the other members of c. After receiving this information (and updating its own act_p and amb_p accordingly), process p checks whether c has the required intersection property with each view in the use_p set. If so, configuration c is given in output to the client at p by means of action NEWCONF$(c)_p$.

When a new primary configuration c has been given in output to processor p by means of action NEWCONF$(c)_p$, the client at p submits its current state together with a condenser function to be used to compute the starting state when all other members have submitted their state (such a condenser function depends on the application). Clearly the state of p is needed by other processors in the configuration while p needs the state of the other processors. Hence when a SUBMIT-STATE$(s, \psi)_p$ is executed at p, the state s submitted by processor p is sent out with a *"got-state"* message to all other members of the configuration. Upon receiving the state of all other processors, DC-CODE$_p$ uses the state condenser function ψ provided by the client at p in order to compute the starting state to be output, by means of action NEWSTATE$(s)_p$, to the client at p.

Finally, the broadcast/convergecast communication mechanism of DC is simulated by using the underlying broadcast communication mechanism (this simulation is quite straightforward).

4.2 Proof

The proof that DC-IMPL implements DC in the sense of trace inclusion is done by using invariants and a simulation relation. The proof is similar to the one in [14] used to prove that DVS-IMPL implements DVS. There is a key difference in the implementation which provides new insights for the DVS specification and implementation, as we explain below.

The DVS specification requires a global intersection property which is the following: given two primary views w and v with no intervening totally established view, we must have that $w.set \cup v.set \neq \{\}$. The DVS implementation, when delivering a new view v, checks *a stronger property* locally to the processors, which requires that $|v.set \cup w.set| \geq |w.set|/2$ for all the views w, $w.id < v.id$, known by the processor performing the check.

The DC specification requires a global intersection property which is the following: given two primary configurations, both of which are not dead, with no intervening totally established configuration, then it must be that there exists a read and write quorum of the configuration with a smaller identifier which are included in the membership set of the configuration with bigger identifier. The DC implementation checks *the same property* locally to each processor. The intuitive reason why by checking locally the same property we can prove it also globally is that we exclude dead configurations. This suggest that also for DVS we can prove the stronger intersection property (the one checked locally) or we can use a weaker local check (the intersection required globally) if we do exclude dead views.

5 Atomic Read/Write Shared Memory Algorithm

In this section we show how to use DC to implement an atomic multi-writer multi-reader shared register. The algorithm is an extension of the single-writer multi-reader atomic register of Attiya, Bar-Noy and Dolev [2]. A similar extension was provided in [30]. The overall algorithm is called ABD-SYS and consists of an automaton ABD-CODE$_p$ for each $p \in \mathcal{P}$, and DC. Due to space constraint the code of automaton ABD-CODE$_P$ is omitted from this extended abstract.

5.1 The algorithm

Each processor keeps a copy of the shared register, in variable *val* paired with a tag, in variable *tag*. Tags are used to establish the time when values are written: a value paired with a bigger tag has been written after a value paired with a smaller tag. Tags consists of pairs $\langle j, p \rangle$ where j is a sequence number (an integer) and p is a processor identifier. Tags are ordered according to their sequence numbers with processors identifiers breaking ties. Given a tag $\langle j, p \rangle$ the notation $t.seq$ denotes the sequence number j.

The algorithm has two modes of operation: a *normal mode* and a *reconfiguration mode*. The latter is used to establish a new configuration. It is entered when a new configuration is announced (action NEWCONF) and is left when the configuration becomes established (action NEWSTATE). The former is the mode where read and write operations are performed and it is entered when a configuration is established and is left when a new configuration is announced. During the reconfiguration mode pending operations are delayed until the normal mode is restored.

Clients of the service can request read and write operations by means of actions READ_p and $\text{WRITE}(x)_p$. We assume that each client does not invoke a new operation request before receiving the response for the previous request. Both type of requests (read and write) are handled in a similar way: there is a *query* phase and a subsequent *propagate* phase. During the query phase the server receiving the request "queries" a read-quorum in order to get the value of the shared register and the corresponding tag for each of the members of the read-quorum. From these it selects the value x corresponding to the max tag t. This concludes the query phase. In the propagation phase the server sends a new value and a new tag (which are (t, x) for the case of a READ_p operation and $(\langle t.seq + 1, p\rangle, y)$ for a $\text{WRITE}(y)_p$ operation) to the members of a write quorum. These processors update their own copy of the register if the tag received is greater than their current tag; then they send back an acknowledgment to the server p. When p gets the acknowledgment message from the members of a write quorum, the propagate phase is completed. At this point the server can respond to the client that issued the operation with either the value read, in the case of a read operation, or with just a confirmation, in the case of a write operation.

We remark that when a configuration change happens during the execution of a requested operation, the completion of the operation is delayed until the normal mode is restored. However if the query phase has already been completed it is not necessary to repeat it in the new configuration.

5.2 Proof

The proof that ABD-SYS implements an atomic read/write shared register is omitted from this extended abstract. The proof uses an approach similar to that used in [14] and in [21] to prove the correctness of applications built on top of DVS and VS, respectively.

We remark that the intersection property of DC, namely that there exist a read quorum R and a write quorum W of a previous primary configuration both belonging to the next primary configuration comes from this particular application. For other applications one might have different (maybe weaker) intersection properties. For example, one might require that the new primary configuration contains a read quorum of the previous one (and not a write one). In our case, we must require both a read quorum and a write quorum in the new primary because we want to implement an atomic register and if, for example we only require a read quorum to be in the new configuration, it is possible that other read quorums of the old configuration will be able to read old values making the register not atomic anymore.

6 Conclusions

In this paper we have combined the notion of dynamic primary views with that of quorum systems, to identify a service that provides *configurations*. Our key contribution in solving the problem of making quorums dynamic, that is, adaptable to the set of processors currently connected, is to identify a suitable intersection property which can be used to maintain consistency across different configurations. An interesting direction of research is to identify which properties have to be satisfied in order to transform a "static" service or application into a "dynamic" one. For example, some data replication algorithms are based on views with a distinguished leader (e.g., [28, 40]) and these applications tolerate transient failures, i.e., they work well in a static setting. We think that it is possible to follow an approach similar to the one used in this paper to transform these applications into ones that adapt better to dynamic settings, where processes can leave the system forever and new members can join the system.

References

1. D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, 1991.
2. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Communications of the ACM*, 42(1):124–142, 1996.
3. Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of Hawaii International Conference on Computer and System Science*, 1995, vol II, pp 612–621.
4. Ö. Babaoğlu, R. Davoli, L. Giachini, and P. Sabattini. The inherent cost of strong-partial view synchronous communication. In *Proceedings of Workshop on Distributed Algorithms on Graphs*, pages 72–86, 1995.
5. Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specifications and Algorithms. TR UBLCS99-01, Department of Computer Science, University of Bologna, 1998.
6. A. Bartoli and Ö. Babaoğlu, Selecting a "Primary Partition" in Partitionable Asynchronous Distributed Systems, In *Proceedings of the 16th Symposium on Reliable Distributed Systems* pages 138–145, 1997.
7. M. Bearden and R. Bianchini Jr. The synchronization cost of on-line quorum adaptation. In *10th (ISCA) International Conference on Parallel and Distributed Computing Systems (PDCS'97)*, pages 598–605, 1997.
8. M. Bearden and R. Bianchini Jr. A fault-tolerant algorithm for decentralized on-line quorum adaptation. In *Proceedings of the* 28^{th} *Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1998.
9. K.P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.
10. T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 322–330, 1996.
11. T. Chandra, and S. Toueg, Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

12. F. Cristian. Group, majority and strict agreement in timed asynchronous distributed systems. In *Proceedings of the 26th Conference on Fault-Tolerant Computer Systems*, pages 178–187, 1996.
13. D. Davcev and W. Buckhard. Consistency and recovery control for replicated files. In ACM Symp. on Operating Systems Principles, volume 10, pages 87–96, 1985.
14. R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the* 17^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 227–236, 1998.
15. D. Dolev and D. Malkhi. The Transis approach to high availability cluster communications. *Communications of the ACM*, 39(4):64–70, 1996.
16. D. Dolev, D. Malkhi, and R. Strong. A framework for partitionable membership service. Technical Report TR95-4, Institute of Computer Science, Hebrew University, Jerusalem, Israel, March 1995.
17. A. El Abbadi and S. Dani. A dynamic accessibility protocol for replicated databases. *Data and knowledge engineering*, 6:319–332, 1991.
18. A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.
19. B. Englert and A.A. Shvartsman. Non-obstructive quorum reconfiguration in a robust emulation of shared memory. Manuscript.
20. P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *15th International Conference on Distributed Computing Systems (ICDCS)*, 1995.
21. A. Fekete, N. Lynch, and A.A. Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the* 16^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 53–62, 1997.
22. R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR95-1537, Department of Computer Science, Cornell University, Ithaca, NY, 1995.
23. D. Gifford. Weighted voting for replicated data. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.
24. M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.
25. M. Hiltunen and R. Schlichting. Properties of membership services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995.
26. F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 2–11, 1993.
27. S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Systems*, 15(2):230–280, 1990.
28. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. Also Research Report 49, DEC SRC, Palo Alto, CA, 1989.
29. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.
30. N. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the* 27^{th} *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 272–281, 1997.

31. N. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.
32. D. Malkhi and M.K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–13, 1998.
33. D. Malkhi, M.K. Reiter, and A. Wool. The load and availability of byzantine quorum systems. In *Proceedings of the 16^{th} ACM Symposium on Principle of Distributed Computing (PODC)*, pages 249–257, 1997.
34. D. Malkhi, M.K. Reiter, and R. Wright. Probabilistic quorum systems. In *Proceedings of the 16^{th} ACM Symposium on Principle of Distributed Computing (PODC)*, pages 267–273, 1997.
35. C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360)*, July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).
36. L. Moser, Y. Amir, P. Melliar-Smith, and D. Agrawal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, 1994. Full version appears in TR ECE93-22, Dept. of Electrical and Computer Engineering, University of California, Santa Barbara, CA.
37. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), April 1996.
38. M. Naor and A. Wool. The load, capacity and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.
39. G. Neiger. A new look at membership services. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 331–340, 1996.
40. B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.
41. J. Paris and D. Long. Efficient dynamic voting algorithms. In *Proceedings of the 13^{th} International Conference on Very Large Data Base*, pages 268–275, 1988.
42. A. Ricciardi. The group membership problem in asynchronous systems. Technical Report TR92-1313, Department of Computer Science, Cornell University, Ithaca, NY, 1992.
43. A. Ricciardi and K.P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10^{th} ACM Symposium on Principle of Distributed Computing (PODC)*, pages 341–352, 1991.
44. A. Ricciardi, A. Schiper, and K.P. Birman. Understanding partitions and the "no partitions" assumption. Technical Report TR93-1355, Department of Computer Science, Cornell University, Ithaca, NY, 1993.
45. R. van Renesse, K.P. Birman, M. Hayden, A. Vaysburd, and D. Karr, Building adaptive systems using Ensemble. *Software– Practice and Experience*, 29(9):963–979, 1998.
46. R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.
47. E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 63–71, 1997.