An algorithm for replicated atomic objects

Roberto De Prisco* Nancy Lynch†

EXTENDED ABSTRACT - August 29, 2001

Abstract

The state machine paradigm is a successful approach for data replication. An example of algorithm that follows this approach is the PAXOS algorithm.

Agreeing on the sequence of operations to be applied to the replicated data is made difficult by the possibility of failures and network partitions. Further difficulties arise from systems that allow processes to join and leave the computation on the fly (i.e., not having a fixed set of processors). Such systems are called dynamic.

In this paper we provide an algorithm, called RAO, that implements a replicated atomic object. The RAO algorithm works in dynamic systems. It is built upon a dynamic group communication service, called DLC, and can be seen as a dynamic version of the PAXOS algorithm. The PAXOS algorithm itself can be implemented on top of DLC, resulting in a simpler code and inheriting the dynamic properties of DLC.

CONTACT AUTHOR: Roberto De Prisco, `robdep@theory.lcs.mit.edu`

*Akamai Technologies, 500 Technology Square, Cambridge, MA 02139.

†MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

1 Introduction

In a distributed system a well known technique for improving performance and availability is that of replication. Objects are replicated at several or all sites so that access is more efficient. Consistency over all replicas must be maintained so that the replicated object behaves as if there was only one copy in the eyes of the external users. In practice this is made difficult by the presence of failures and network partitions. Extensive work has been done in the area of data replication (see for example [3, 7, 8, 9, 10, 18]).

One way of implementing replication is that of using the state machine approach: given that all replicas are identical deterministic state machines starting the computation from the same initial state, it is enough that each replica apply the exact same sequence of operations to the replicated object in order to maintain consistency.

So the problem is reduced to that of totally ordering operations among all the replicas. A great help is given by group communication services which provide tools to broadcast messages within group of processors (see for example [1, 12]). The COReL algorithm [12] is built upon the Horus and Transis group communication systems. In COReL each update to the replicated object can be initiated by any replica which sends a message and the algorithm guarantees that all messages will eventually reach all replicas in the same order.

The PAXOS algorithm [13] also follows the state machine approach but it does not use group communication services. In PAXOS the computation is coordinated by leaders which may change over time as a result of failures. The basic PAXOS algorithm is a consensus algorithm used to agree on a single operation. To agree on the sequence of operations to be applied to the replicated object we need to run a sequence of instances of the basic PAXOS algorithm. The basic PAXOS algorithm uses two full rounds of communication in order to achieve consensus given that the system is stable, that is, there is a leader that does not crash for long enough. When running a sequence of instances of PAXOS it is possible to optimize the communication requiring only one full round of communication per each operation, while two full rounds are required only when there is a change in the leadership.

Dealing with dynamic systems in which processes can join and leave the computation on the fly makes the problem harder. To cope with partitions progress is allowed only in a distinguished *primary* component. In dynamic systems a *dynamic primary* component is necessary. Several dynamic voting schemes have been developed to define primary components adaptively, e.g. [2, 7, 11, 15, 16, 17, 19]. View-oriented group communication services that provide dynamic primary views are provided in [4, 5].

The PAXOS algorithm is quite fault tolerant: process crash, network partitions, message loss, duplication and reordering are allowed and progress is guaranteed as long as there is a primary component. However it requires a fixed set of processors.

In this paper we provide an algorithm, called RAO, that implements a replicated atomic object. The RAO algorithm can be seen as a dynamic version of the PAXOS algorithm. As we will see later, the RAO algorithm has the same key steps as the PAXOS algorithm.

The RAO algorithm is built upon a group communication service called DLC, dynamic leader configuration. The DLC service that we use as a building block is a variation of the DC group communication service provided in [5]. The advantage of implementing the RAO algorithm using a dynamic group communication service is that the algorithm becomes dynamic.

2 Notation and data types

$\mathcal{P}$ denotes the universe of all processors,[1] and $\mathcal{M}$ the universe of all messages. $\mathcal{G}$ is a totally ordered set of configuration identifiers, with a distinguished least element g_0.

A *group* is a set of processors. A *view* is a group membership with a unique identifier. A *configuration* is a view augmented with a quorum system defined on the set of processors members of the group and a leader process. More formally, a configuration is a quadruple, $c = \langle g, P, \mathcal{Q}, p\rangle$, where $g \in \mathcal{G}$ is a unique identifier, $P \subseteq \mathcal{P}$ is a nonempty set of processors, $\mathcal{Q}$ is a quorum system for P, and p is a distinguished element of P, called the leader of the configuration c. Each element of $\mathcal{Q}$ is called a *quorum* of the configuration c. We denote with $\mathcal{C}$ the set of all configurations. Given a configuration $c = \langle g, P, \mathcal{Q}, p\rangle$, selectors for the components are $c.id = g$, $c.set = P$, $c.qrms = \mathcal{Q}$ and $c.ldr = p$.

We distinguish an initial configuration $c_0 = \langle g_0, P_0, \mathcal{Q}_0, p_0\rangle$.

The following data types pertain to the DLC service.

Let $\mathcal{I}$ be a set of operation identifiers, partitioned into sets $\mathcal{I}_p$, $p \in \mathcal{P}$. We denote by $\mathcal{M}_c \subseteq \mathcal{M}$ the set of messages that clients of the DLC service may use for communication.

Let $\mathcal{A}$ be a set of "acknowledgment" values and let $\mathcal{R}$ be a set of "response" values. A *condenser function* ϕ is a function $\phi : (\mathcal{P} \to \mathcal{A}_\perp) \to \mathcal{R}$. That is, a condenser function takes a function from processes to acknowledgement values and returns a response value. Let Φ be the set of all condenser functions. Let $\mathcal{S}$ be the set of all possible states of the clients of the DLC service (a state of $\mathcal{S}$ 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). The DLC service uses a condenser function also to compute the starting state of a new configuration; hence we assume that $\mathcal{S} \subseteq \mathcal{A}$ and also $\mathcal{S} \subseteq \mathcal{R}$. Given a function $f : \mathcal{P} \to D$ from the set of processes $\mathcal{P}$ to some domain D and given a subset $P \subseteq \mathcal{P}$, we write $f|P$ to denote the function $f' : P \to D$, defined as $f'(p) = f(p)$ for $p \in P$.

The following data type is used to describe operations submitted by the client to the DLC service: $\mathcal{D} = \mathcal{M} \times \Phi \times 2^{\mathcal{P}} \times (\mathcal{P} \to \mathcal{A}_\perp) \times Bool$ and we let $\mathcal{O} = \mathcal{I} \to \mathcal{D}_\perp$. Given an operation descriptor, selectors for the components are *msg*, *cnd*, *dlv*, *ack*, and *rsp*.

3 The DLC group communication service

In this section we provide the DLC service, a dynamic primary configuration group communication service that supports leaders.

DLC

Signature:

Input:	SUBMIT$(m, \phi, i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi$, $p \in \mathcal{P}$, $i \in \mathcal{I}_p$	Output:	NEWCONF$(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
	ACKDLVR$(a, i)_p$, $a \in A$, $i \in \mathcal{I}$, $p \in \mathcal{P}$		NEWSTATE$(s)_p$, $s \in \mathcal{S}$, $p \in \mathcal{P}$
	SUBMIT-STATE$(s, \phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$, $p \in \mathcal{P}$		RESPOND$(r, i)_p$, $r \in \mathcal{R}$, $i \in \mathcal{I}_p$, $p \in \mathcal{P}$
	P2P-RECV$(m)_{q,p}$, $m \in \mathcal{M}$, $q, p \in \mathcal{P}$		DELIVER$(m, i)_p$, $m \in \mathcal{M}$, $i \in \mathcal{I}$, $p \in \mathcal{P}$
Internal:	CREATECONF(c), $c \in \mathcal{C}$		P2P-SEND$(m)_{p,q}$, $m \in \mathcal{M}$, $q, p \in \mathcal{P}$,

Figure 1: The DLC specification signature

We use the IOA model (see [14]) to specify our algorithms and services. Figure 1 shows the signature of DLC. The signature provides the interface of the service with the external environment.

[1] We use "processor" and "process" interchangeably.

Since for lack of space we will not discuss in detail the DLC specification we have moved the full code to Appendix A.

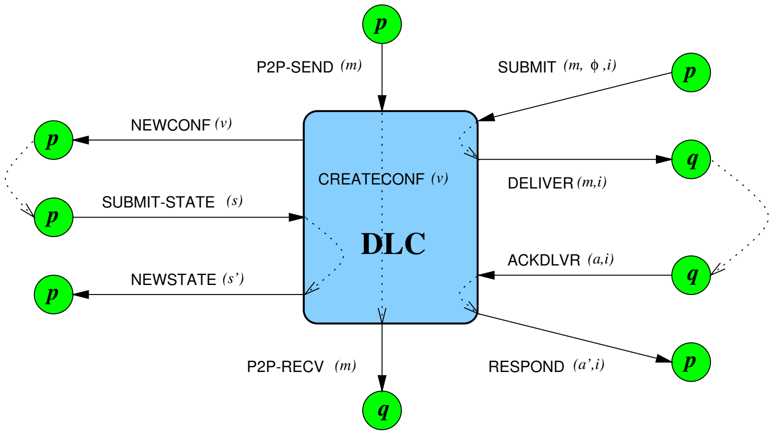

Figure 2: The interface of the DLC dlc service

Briefly, the DLC service offers the following communication tools: A process p can submit a "message" m to the service, together with a condenser function ϕ; the service tries to deliver the message to all the members of the current configuration; when at least a quorum has provided an "acknowledgement" value for m, the service uses ϕ to compute a "response" to give back to p. The DLC service also notifies changes of configuration by means of action NEWCONF(c), and use a similar communication pattern to compute the starting state for a new configuration: all processes members of the new configuration submit their current state with action SUBMIT-STATE(s), and the service computes with a predefined condenser function the new state which is then delivered with action NEWSTATE(s). Configurations are created internally with action CREATECONF(c). Point to point communication is also available. Figure 2 provides a graphical representation of these tools.

A configuration c is attempted when at least one process $p \in c.set$ has executed NEWCONF$(c)_p$. A configuration is totally attempted when all processes $p \in c.set$ have executed NEWCONF$(c)_p$. A configuration c is established when at least one process $p \in c.set$ has executed NEWSTATE$(s)_p$ for configuration c. A configuration c is totally established when all processes $p \in c.set$ have executed NEWSTATE$(s)_p$ for configuration c. 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.

The DLC service is similar to the DC service provided in [5]. There are two differences between DC and DLC.

The first difference is due to the kind of configurations considered: leader configurations instead of read-write quorum configurations (as done in DC).

The second difference is that DLC offers also point-to-point communication. That is, a process p can send a message m to another process q, provided that both p and q are in the same configuration. Actions P2P-SEND$(m)_{p,q}$ and P2P-RECV$(m)_{p,q}$ of DLC implement the point-to-point communication mechanism (this portion of the code is not present in DC).

The rest of the DLC specification is the same as the DC specification. An informal explanation of DC, as well as detailed comments on the code can be found in [5].

The internal action CREATECONF(c) has a precondition that is crucial to prove the correctness of algorithms built on top of DLC and defines the "primary configurations". 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 among c and w (i.e., the one with smaller identifier) has one quorum whose members are included in the membership set of the later configuration (i.e., the one with bigger identifier).

4 The DPAXOS algorithm

We first briefly review the PAXOS algorithm [13]. The PAXOS algorithm, in its basic version, solves the consensus problem. The algorithm needs two full rounds of communication. In the first round the leader sends a message to all participating processes asking for information about the history of the computation; then each process sends this information back to the leader. When a quorum of processes has provided information, the leader is able to choose a value to be proposed as the consensus value. This first round is needed because past leaders might have already initiated attempts to reach consensus and in order to keep consistency the value to be proposed as the consensus value has to be carefully chosen. Then in the second round of communication the leader sends the proposed value to all participating processes and when a quorum of the processes accepts the proposed value, the value becomes the consensus value.

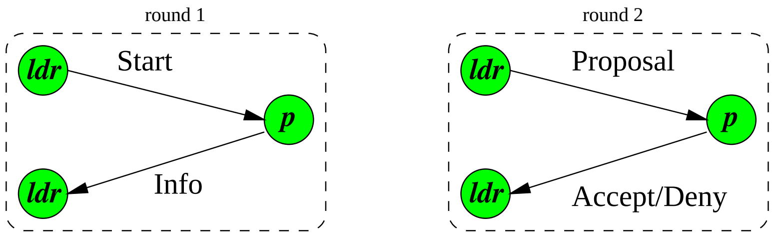

Figure 3: The rounds of communication in PAXOS.

Figure 3 summarizes the two rounds of communication needed by PAXOS.

We have provided an implementation of the basic PAXOS algorithm on top of DLC. The resulting algorithm is called DPAXOS. In DPAXOS the first round of communication is embedded into the state exchange of the DLC service while the second round is left to the implementation. For lack of space we omit the code and the proof of correctness of DPAXOS. However we provide the IOA code in Appendix B.

The basic version of the PAXOS algorithm solves the consensus problem. To implement a replicated atomic object using the state machine approach, a sequence of instances of the basic version of the algorithm is needed, so that the k^{th} instance can be used to reach consensus on the k^{th} operation to be performed on the replicated data. The resulting algorithm has been called MULTIPAXOS in [6]. The paper by Lamport [13] shows how to combine the first round of multiple instances of PAXOS so that the first round is run only once for all instances and thus only one full round of communication is needed to reach consensus in each instance, as long as there are no failures. When failures occur and the leadership changes, then MULTIPAXOS needs to go through both rounds of communication again. The MULTIPAXOS algorithm is rather complicated.

Using the DLC service most of the difficulties are hidden inside the service itself making the algorithm a lot simpler. In the next section we present the RAO algorithm which, even though is not a direct implementation of MULTIPAXOS it basically does the same thing: uses one round of communication during the state exchange for a new configuration in order to gather information about the history, and uses another round of communication to reach agreement on each of the operations to be executed on the replicated object.

The advantages of building such an algorithm on top of DLC is two-fold: on one hand the algorithm, exploiting the communication and state exchange tools of the service, becomes simpler and on the other hand it inherits the dynamic properties of the service allowing participating processes to join and leave the computation on the fly.

5 The RAO algorithm

5.1 Overview of RAO

Users of the replicated atomic object can request operations at any process. Upon reception of a request a process places the request into a local order. Then each of the received requests is sent to the leader of the current configuration. The leader is responsible for building a global order for all the client's requests. For each of these requests the leader, using the DLC service, makes sure that enough processes (at least a quorum) know the request. Then it computes an answer which is sent back to the process that accepted the request from the client. Finally the answer is given back to the client. See Figure 4 for the work-flow of such a process.

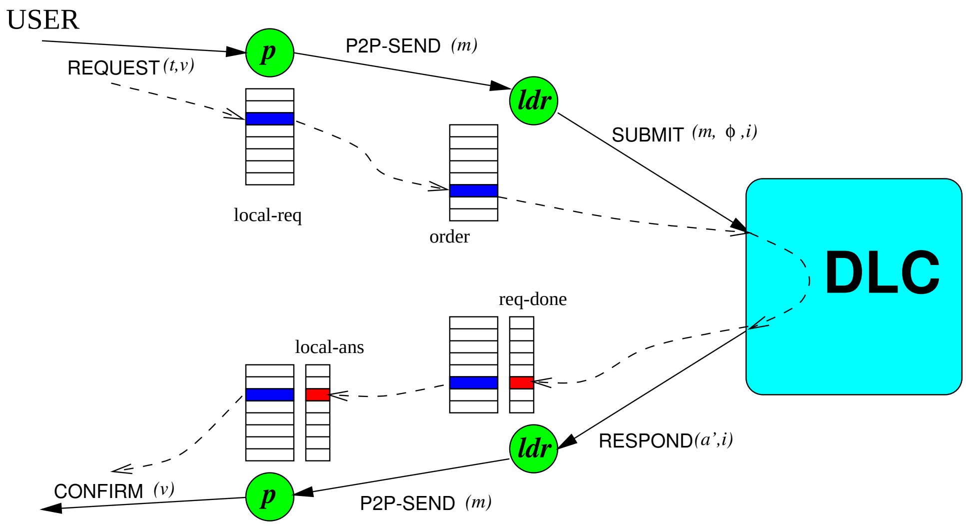

Figure 4: Work-flow for the RAO algorithm.

When a configuration change happens all the members submit their knowledge about the requests performed so far and a new common state is computed from the local knowledge of the processes. In particular, each process submits its own information about the global order of operations plus all the local requests that are still pending, that is, have been submitted to the leader but have not received an answer. The global orders submitted by each member of the new configuration are used to compute the most up to date global order, while the information about pending requests is used to locate those operations that must be resubmitted to the leader. The above is done using the DLC service.

5.2 The code of DLC-TO-RAO

In this section we provide the code of algorithm DLC-TO-RAO. We first define some data types. We denote by $\mathcal{V}$ the set of values that the replicated data can assume and by $v_0 \in \mathcal{V}$ a predefined value. The set $\mathcal{T}$ is a set of types of operations. For example, if we want a read-write register, types could be $\mathcal{T} = \{\text{“}read\text{”}, \text{“}write\text{”}\}$.

The set $\mathcal{D}'$ of “operation descriptors” is defined as $\mathcal{D}' = \{\langle p, t, w, i\rangle | p \in \mathcal{P}, t \in \mathcal{T}, w \in \mathcal{V}, i \in \mathbf{N}^{>0}\}$. (Notice that this is different from the operation descriptors $\mathcal{D}$ for the DLC service. The set $\mathcal{D}'$ is used to describe operations submitted to the RAO algorithm, while the set $\mathcal{D}$ is used to describe operations that are submitted by DLC-TO-RAO to the DLC service.)

Operation descriptors are used to describe both the requests from the clients and the corresponding responses. For an element $y = \langle p, t, w, i\rangle$ of $\mathcal{D}'$ we use the following selectors to extract the single components: $y.origin = p$, $y.type = t$, $y.param = w$ and $y.local\text{-}rank = i$. Component $y.origin$ records the client at which the request has originated. Component $y.type$ specifies the type of the operation. Component $y.param$ provides possible parameters that need to be passed along with the request or with the corresponding response. Considering again the case of a read-write register, a write needs to pass the value to be written and the response to a read needs to pass the value read. For simplicity we assume that only one value needs to be passed and this value is an element of $\mathcal{V}$ (this is so in the case of a read-write register).

Next we describe the data types that DLC-TO-RAO uses for the interface with the DLC service. The notation $arrayof(\mathcal{D}')$ indicates an array whose elements are either elements of $\mathcal{D}'$ or $\perp$. The set of messages that can be sent over the point to point channels and through the DLC service is defined as $\mathcal{M}_c = \mathcal{D}' \cup (\{\text{“}req\text{”}, \text{“}ans\text{”}\} \times \mathcal{D}')$. The set of operation identifiers is $\mathcal{I} = \mathbf{N}^{>0} \times \mathcal{P}$. The set $\mathcal{S}$ is defined as $\mathcal{S} = \{\langle o, d, a, h\rangle | o \in arrayof(\mathcal{D}'), d \in arrayof(bool), a \in arrayof(bool), h \in \mathcal{G}\}$. For $x \in S$, $x = \{\langle o, d, a, h\rangle$, selectors are $x.order = o$, $x.req\text{-}done = d$, $x.req\text{-}answrd = a$ and $x.high = h$.

The set of acknowledgment values is $\mathcal{A} = \{\text{“}ack\text{”}\} \cup \mathcal{S}$. The set of response values is $\mathcal{R} = \{\text{“}done\text{”}\} \cup \{\langle o, d, a\rangle | o \in arrayof(\mathcal{D}'), d \in arrayof(bool), a \in arrayof(bool)\}$.

The DLC-TO-RAO algorithm uses two condenser functions that we define in the following:

- ϕ_{done}: This condenser function takes a set of acknowledgment values “*ack*” and returns the string “*done*”. This function is used by the leader to make sure that a quorum of processes has received information about a particular operation.

- $\phi_{raostate}$: Let c be the configuration for which this condenser function is to be used. The condenser function takes a collection $S \subseteq \mathcal{S}$ of tuples, one for each member of c, and returns a triple $\langle o, d, a\rangle \in \mathcal{R}$, defined as follows:

  - o is defined as follows: Let R be the set of tuples of S that have the maximum *high* component. For any $i \in \mathbf{N}^{>0}$ such that there exists at least one element $x \in R$ with $x.order(i) \neq \texttt{nil}$, fix any such element x and set $o(i) := x.order(i)$. For any i for which no such element exists set $o(i) := \perp$. Let j be the smallest index such that $o(j) = \perp$; for $i > j$ set $o(i) := \perp$.
  - d is defined as the “or” of the *req-done* components in R.
  - a is defined as the “or” of the *req-answrd* components in R.

The code of DLC-TO-RAO is provided in Figures 5 and 6. The code uses the following helper functions: function $apply\text{-}all(i)$, used to compute the value of the replicated atomic object after the execution of the i^{th} operation, and function *update*(*local-ans*) used to update the variables that store the answers to operations after a configuration change.

$apply\text{-}all(i)_p$ is defined as follows:
if for all $k \leq i$, $order(k)_p \neq \perp$ then
$apply\text{-}all(i)_p = \langle q, t, w, j\rangle$, where $q = order(i)_p.origin$, $t = order(i)_p.type$,
w is the value obtained by applying operations $order(1,..,i)_p$ to the initial value v_0 in order,
and $j = order(i)_p.local\text{-}rank$;
else
$apply\text{-}all(i)_p = \perp$.

$update(local\text{-}ans)_p$ is defined as follows:
for any i such that
$local\text{-}req(i) \neq \perp, local\text{-}ans(i) = \perp$, and
for all $j \leq i$, $req\text{-}done(j)_p = \texttt{true}$, do
$local\text{-}ans(i) = apply\text{-}all(i)_p$

Next we provide a description of DLC-TO-RAO.

Variable $current_p$ contains the current configuration of process p and variable $high_p$ contains the latest established configuration of process p. Variable $local\text{-}req_p$ is the sequence of requests that the client submits at process p; variable $local\text{-}ans_p$ contains the answers for these requests. Variable $next_p$ is a pointer used to insert new requests from the client into $local\text{-}req_p$. Variable $order_p$ contains the global sequence of all requests as known by process p. Variable $status_p$ contains the status of process p; it is used when a new configuration is announced; for regular computation this variable is set to *normal*. The remaining state variables are used for bookkeeping and their use should be clear from the code. Next we describe the actions, in the order in which they are executed (see also Figure 4).

A client requests an operation on the object to a process p by means of action REQUEST$(type, param)_p$. Variable *type* specifies the type of the operation (e.g., read or write) and variable *param* provides an optional parameters (e.g., the value of the write). The request is stored into the local order $local\text{-}req_p$ of process p.

Process p sends a request r to the leader of the current configuration, say process q, by means of action P2P-SEND$(\langle\text{“}req\text{”}, r\rangle)$. The leader q, upon reception of a request r, inserts the request into its global order of operations $order_q$, in the first available position.

Then the leader uses the DLC service to make sure that a quorum of processes knows about the request. First the leader q submits the request r by means of action SUBMIT$(r, \phi_{done}, \langle i, q\rangle)_q$. Parameter r contains the request being submitted, parameter ϕ_{done} is the condenser function and parameter $\langle i, q\rangle$ is the operation identifier for the DLC service. The DLC service delivers the request by means of action DELIVER$(r, \langle i, q\rangle)_{p'}$ to process p'. Then process p' acknowledges the request by means of action ACKDLVR$(\text{“}ack\text{”}, \langle i, q\rangle)_{p'}$. Once a quorum of process has acknowledged request r the DLC service can apply the condenser function ϕ_{done}. This function does nothing since its sole purpose is to make sure that a quorum of processes has acknowledged the request, which is true when the function gets applied. At this point the DLC service can notify the leader q, which has submitted the request, that a quorum of processes knows about the request. This is done by means of action RESPOND$(\text{“}done\text{”}, \langle i, q\rangle)_q$.

Now the leader q is able to compute an answer (provided that all previous requests have been completed) and sends it back to the process p that originated the request on behalf of the client. This is done with action P2P-SEND$(\langle\text{“}ans\text{”}, a\rangle)_{q,p}$, where a is the result of applying operation r to the replicated atomic object.

Process p receives the answer a with action P2P-RECV$(\langle\text{“}ans\text{”}, a\rangle)_{q,p}$ and stores it locally in the

DLC-TO-RAO

Signature:

Input: $\text{REQUEST}(type, param)_p$, $type \in \mathcal{T}$, $param \in \mathcal{V}, p \in \mathcal{P}$
$\text{P2P-RECV}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
$\text{DELIVER}(m,i)_p$, $m \in \mathcal{M}$, $i \in \mathcal{I}$, $p \in \mathcal{P}$
$\text{RESPOND}(a,i)_p$, $a \in \mathcal{A}$, $i \in \mathcal{I}_p$, $p \in \mathcal{P}$
$\text{NEWCONF}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\text{NEWSTATE}(s)_p$, $s \in \mathcal{S}$, $p \in \mathcal{P}$

Output: $\text{CONFIRM}(param)_p$, $param \in \mathcal{V}$, $p \in \mathcal{P}$
$\text{P2P-SEND}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
$\text{SUBMIT}(m, \phi, i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi, p \in \mathcal{P}$, $i \in \mathcal{I}_p$
$\text{ACKDLVR}(a,i)_p$, $a \in \mathcal{A}$, $i \in \mathcal{I}$, $p \in \mathcal{P}$
$\text{SUBMIT-STATE}(s,\phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$, $p \in \mathcal{P}$

State:

$current \in \mathcal{C}_\perp$ initially c_o if $p \in P_0$, else $\perp$
$high \in \mathcal{C}_\perp$ initially g_0 if $p \in P_0$, else $\perp$
$local\text{-}req \in arrayof(\mathcal{D}')$, initially $\perp$ everywhere
$local\text{-}ans \in arrayof(\mathcal{D}')$, initially $\perp$ everywhere
$next \in \mathbf{N}$, initially 1
$order \in arrayof(\mathcal{D}')$, initially $\perp$ everywhere
$status \in \{normal, exch\text{-}ready, exch\text{-}wait\}$, initially $normal$

$req\text{-}sent \in seqof(bool)$, initially `false` everywhere
$req\text{-}sbmttd \in seqof(bool)$, initially `false` everywhere
$req\text{-}acked \in seqof(bool)$, initially `false` everywhere
$req\text{-}leader \in seqof(\mathcal{P})$, initially $\perp$ everywhere
$req\text{-}done \in seqof(bool)$, initially `false` everywhere
$req\text{-}answrd \in seqof(bool)$, initially `false` everywhere
$req\text{-}cnfrmd \in seqof(bool)$, initially `false` everywhere

Actions:

input $\text{REQUEST}(type, param)_p$
Eff: $local\text{-}req(next) := \langle p, type, param, next\rangle$
$next := next + 1$

output $\text{CONFIRM}(param)_p$
Pre: $local\text{-}ans(i) = \langle p, type, param, i\rangle$
$\forall j \le i,\ req\text{-}cnfrmd(j) = \texttt{true}$
$req\text{-}cnfrmd(i) = \texttt{false}$
Eff: $req\text{-}cnfrmd(i) := \texttt{true}$

output $\text{P2P-SEND}(\langle \text{“}req\text{”}, r\rangle)_{p,q}$ choose j
Pre: $q = current.ldr$
$r = local\text{-}req(j)$
$r \neq \perp$
$\forall i < j, req\text{-}sent = \texttt{true}$
$req\text{-}sent(j) = \texttt{false}$
$status = normal$
Eff: $req\text{-}sent(j) := \texttt{true}$

input $\text{P2P-RECV}(\langle \text{“}req\text{”}, r\rangle)_{q,p}$
Eff: Let i be such that
$\forall k < i,\ order(k) \neq \perp$
$order(i) = \perp$
$order(i) := r$

output $\text{SUBMIT}(r, \phi_{done}, \langle i, p\rangle)_p$
Pre: $p = current.ldr$
$order(i) = r$
$r \neq \perp$
$req\text{-}sbmttd(i) = \texttt{false}$
$status = normal$
Eff: $req\text{-}sbmttd(i) := \texttt{true}$

input $\text{DELIVER}(r, \langle i, q\rangle)_p$
Eff: $order(i) := r$
$req\text{-}acked(i) := \texttt{false}$
$req\text{-}leader(i) = q$

output $\text{ACKDLVR}(\text{“}ack\text{”}, \langle i, q\rangle)_p$
Pre: $q = current.ldr$
$order(i) \neq \perp$
$req\text{-}leader(i) = q$
$req\text{-}acked(i) = \texttt{false}$
$status = normal$
Eff: $req\text{-}acked(i) := \texttt{true}$

input $\text{RESPOND}(\text{“}done\text{”}, \langle i, p\rangle)_p$
Eff: $req\text{-}done(i) := \texttt{true}$

Figure 5: The DLC-TO-RAO code.

DLC-TO-RAO

output P2P-SEND$(\langle$"*ans*"$, a\rangle)_{p,q}$ choose i
Pre: $p = current.ldr$
$req\text{-}answrd(i) = \texttt{false}$
$\forall j \le i,\ req\text{-}done(j) = \texttt{true}$
$q = order(i).origin$
$a = apply\text{-}all(i)$
$status = normal$
Eff: $req\text{-}answrd(i) := \texttt{true}$

input P2P-RECV$(\langle$"*ans*"$, a\rangle)_{q,p}$
Eff: $j := a.local\text{-}rank$
$local\text{-}ans(j) := a$
$local\text{-}ack(j) := \texttt{false}$

output P2P-SEND$(\langle$"*ack*"$, r\rangle)_{q,p}$ choose i
Pre: $p = current.ldr$
$local\text{-}req(i) = r$
$r \neq \perp$
$local\text{-}ack(i) = \texttt{false}$
$status = normal$
Eff: $req\text{-}answrd(i) := \texttt{true}$

input P2P-RECV$(\langle$"*ack*"$, r\rangle)_{q,p}$
Eff: Let j be such that $local\text{-}req(j) = r$
$req\text{-}answrd(j) := \texttt{true}$

input NEWCONF$(c)_p$
Eff: $current := c$
$status := exch\text{-}ready$
$\forall i,\ order(i) \neq \perp$ and $req\text{-}acked(i) = \texttt{false}$
$req\text{-}acked(i) := \texttt{true}$

output SUBMIT-STATE$(\langle o, d, a, g\rangle, \phi_{raostate})_p$
Pre: $status = exch\text{-}ready$
$o = order$
$d = req\text{-}done$
$a = req\text{-}answrd$
$g = high.id$
Eff: $status := exch\text{-}wait$

input NEWSTATE$(\langle o, d, a\rangle)_p$
Eff: $status := normal$
$high := current$
$order := o$
$req\text{-}done := d$
$req\text{-}answrd := a$
$update(local\text{-}ans)$
$\forall j, local\text{-}req(j) \neq \perp$
if $\nexists i$ such that
$order(i).origin = p$ and
$order(i).local\text{-}rank = j$ then
$req\text{-}sent(j) := \texttt{false}$
$\forall i, order(i) \neq \perp$
if $req\text{-}done = \texttt{false}$ then
$req\text{-}sbmttd(i) := \texttt{false}$
else
$req\text{-}sbmttd(i) := \texttt{true}$

Figure 6: The DLC-TO-RAO code (cont'd).

variable $local\text{-}ans_p$. Process p also sends back an acknowledgement to the leader q with action P2P-SEND$(\langle$"*ack*"$, r\rangle)_{p,q}$ for request r. Such an acknowledgement is received by the leader q with action P2P-RECV$(\langle$"*ack*"$, r\rangle)_{p,q}$.

Once process p gets the answer for a request r it can relay the answer to the client that requested the operation. This is done by means of action CONFIRM$(param)_p$.

The actions that we have described above are those actions that get executed when there is no configuration change, that is when the mode of the system is "normal" (see variable *status*). When there is a configuration change, a state exchange must be executed in order to bring every member of the new configuration up to date.

A configuration change starts with a notification from the DLC service about the new configuration. This happens by means of action NEWCONF$(c)_p$. Process p sets its current configuration to c and goes into reconfiguration mode by setting $status_p$ to *exch-ready*. For any request not yet completed, process p has to re-send the request to the new leader, because it cannot know whether the request has been completed or not by the old leader. This is accomplished by resetting variables *req-sent*. It also sets $req\text{-}acked_p$ to `true` for all those operations that have pending acknowledgments to be sent; since the old configuration is not current anymore, such an acknowledgment must no longer be sent and setting the flag $\mathit{req\text{-}acked}_p$ to `true` has this effect. Indeed in the new configuration process p has not yet received any message from the leader so it is incorrect to acknowledge a message.

Then process p submits to the DLC service its order_p, $\mathit{req\text{-}done}_p$, $\mathit{req\text{-}answrd}_p$ and high_p, which constitute the relevant part of the state that has to be exchanged.

When all the processes have submitted their states, the DLC service is able to compute the starting state of the new configuration by using the $\phi_{raostate}$ condenser function. Then it gives this state to process p by means of action $\text{NEWSTATE}(\langle o, d, a\rangle)_p$. When this action is executed, process p updates its order_p, $\mathit{req\text{-}done}_p$ and $\mathit{req\text{-}answrd}_p$ state components. It also adjusts the values of $\mathit{req\text{-}sent}_p$ and $\mathit{req\text{-}sbmttd}_p$ to take care of two problems that arise in establishing a new configuration, as we explain below.

The first problem is that any process p has to check whether all of its local requests are in the global order o returned by $\text{NEWSTATE}(\langle o, d, a\rangle)_p$; for any local request not included in the order o, process p has to send that local request to the leader of the new configuration because the leader does not know about such a request. This is done by setting $\mathit{req\text{-}sent}(j)$ to `false` for those local operations that the leader does not know about.

The other problem regards operations that are included in the order o returned by $\text{NEWSTATE}(\langle o, d, a\rangle)_p$, but for which *req-done* is still `false`. For such operations the leader cannot be sure that a quorum of processes have them in their global order and thus cannot provide an answer for such operations. The leader needs to resubmit such operations to the DLC service in order to make sure that a quorum of processes learn about them, before giving an answer to the originator of the request. Hence for any i such that $\mathit{order}(i)_p \neq \perp$ and $\mathit{req\text{-}done}(i) = \texttt{false}$, that is, for any operation in *order* for which the leader cannot be sure that a quorum of processes know about that operation, the flag $\mathit{req\text{-}sbmttd}(i)$ is set to `false` so that the operation will be submitted to the DLC service.

5.3 The overall system

The RAO algorithm is given by the composition of DLC and DLC-TO-RAO.

Claim 5.1 *The system* RAO *implements an atomic object.*

For lack of space, we leave the proof of correctness to the full version of this paper.

References

[1] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1994.

[2] 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.

[3] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 3(17):341–370, September 1985.

[4] 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, Puerto Vallarta, Mexico, 1998.

[5] R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman. A dynamic primary configuration group communication service. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing (DISC)*, Bratislava, Slovak Republic, 1999.

[6] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(-12):35–91, July 2000.

[7] A. El Abbadi and S. Dani. A dynamic accessibility protocol for replicated databases. *Data and knowledge engineering*, 6:319–332, 1991.

[8] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[9] D. Gifford. Weighted voting for replicated data. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

[10] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[11] 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.

[12] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also Institute of Computer Science, The Hebrew University of Jerusalem Technical Report CS95-5, and available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[13] 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.

[14] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[15] 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).

[16] 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.

[17] 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, August 1991.

[18] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on databse Systems*, 4(2):180–209, June 1979.

[19] 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, Santa Barbara, CA, August 1997.

A IOA code for the DLC service

Figure 7 shows the state and the transitions for the DLC group communication service. The signature is shown in Figure 1 (Section 3).

B IOA code for the DPAXOS algorithm

Figure 8 contains the code of DLC-TO-PAXOS which composed with DLC gives the DPAXOS algorithm. Variables *Haccquo*, *Hfrom* and *Hvalue* are used for the correctness proof which is left to the full version of this paper.

The condenser function ϕ_{state} collects all the states submitted by processes in the current configuration and computes the value to propose in the current round. Formally it is a function that takes a set of pairs $W = \{\langle r, v\rangle | r \in \mathcal{G}, v \in \mathcal{V}\}$ and returns a pair $\langle r', v'\rangle \in W$ where r' is such that $r' \geq r$ for all $\langle r, v\rangle \in W$.

The condenser function ϕ_{begin} is a function that takes a set of acknowledgment values $\{\langle \text{"}accept\text{"}, i\rangle_q | q \in Q\}$ for some set of processes $Q \subseteq \mathcal{P}$ and returns $\langle \text{"}begin\text{"}, Q\rangle$.

DLC

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 $p, q \in \mathcal{P}, g \in \mathcal{G}$:
  $p2p\text{-}msgs[p,g,q] seqof(\mathcal{M})$, initially $\emptyset$
  $p2p\text{-}next[p,g,q] \in \mathbf{N}^{>0}$, initially 1

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

Derived variables:

$\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\{c \in created | attempted[c.id] \neq \emptyset\}$
$\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\{c \in created | state\text{-}dlv[c.id] \neq \emptyset\}$
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\{c \in created | c.set \subseteq attempted[c.id]\}$
$\mathcal{T}ot\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\{c \in created | c.set \subseteq state\text{-}dlv[c.id]\}$
$dead \in 2^{\mathcal{C}}$ defined as $dead = \{c \in \mathcal{C} | \exists p \in c.set : cur\text{-}cid_p > c.id$ and $p \notin attempted[c.id]\}$.

Actions:

internal CREATECONF(c)
  Pre: $\forall w \in created : c.id \neq w.id$
    if $c \notin dead$ then
      $\forall w \in created, w.id < c.id$:
        $w \in dead$ or
        $(\exists x \in \mathcal{T}ot\mathcal{E}st$: $w.id < x.id < c.id)$ or
        $(\exists Q \in w.qrms$: $Q \subseteq c.set)$
      $\forall w \in created, w.id > c.id$
        $w \in dead$ or
        $(\exists x \in \mathcal{T}ot\mathcal{E}st$: $c.id < x.id < w.id)$ or
        $(\exists Q \in c.qrms$: $Q \subseteq w.set)$
  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$(s, \phi)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ and
      $got\text{-}state[cur\text{-}cid[p]](p) = \perp$ then
        $got\text{-}state[cur\text{-}cid[p]](p) := s$
        $condenser[cur\text{-}cid[p]](p) := \phi$

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

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

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: $c.id = cur\text{-}cid[p]$
    $c \in created$
    $i \in \mathcal{I}_p$
    $pending[c.id](i).rsp = \texttt{false}$
    $Q \in c.qrms$
    let $f = pending[c.id](i).ack$
    $\forall q \in Q : f(q) \neq \perp$
    $r = pending[c.id](i).cnd(f|Q)$
  Eff: $pending[c.id](i).rsp := \texttt{true}$

input P2P-SEND$(m)_{p,q}$
  Eff: if $cur\text{-}cid[p] \neq \perp$ then
      append m to $p2p\text{-}msgs[p, cur\text{-}cid[p], q]$

output P2P-RECV$(m)_{p,q}$, choose g
  Pre: $g = cur\text{-}cid[q]$
    $mp2p\text{-}msgs[p,g,q](p2p\text{-}next[p,g,q])$
  Eff: $p2p\text{-}next[p,g,q] := p2p\text{-}next[p,g,q] + 1$

Figure 7: The DLC specification - state and transitions

DLC-TO-PAXOS

Signature:

Input: $\text{NEWCONF}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\text{NEWSTATE}(s)_p$, $s \in \mathcal{S}$, $p \in \mathcal{P}$
$\text{DELIVER}(m, i)_p$, $m \in \mathcal{M}$, $i \in \mathcal{I}$, $p \in \mathcal{P}$
$\text{RESPOND}(a, i)_p$, $a \in \mathcal{A}$, $i \in \mathcal{I}_p$, $p \in \mathcal{P}$
$\text{P2P-RECV}(m)_{q,p}$, $m \in \mathcal{M}$, $q, p \in \mathcal{P}$

Internal: NEWROUND_p, $p \in \mathcal{P}$
Output: $\text{SUBMIT}(m, \phi, i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi$, $p \in \mathcal{P}$, $i \in \mathcal{I}_p$
$\text{ACKDLVR}(a, i)_p$, $a \in A$, $i \in \mathcal{I}$, $p \in \mathcal{P}$
$\text{SUBMIT-STATE}(s, \phi)_p$, $s \in S$, $\phi \in \Phi$, $p \in \mathcal{P}$
$\text{P2P-SEND}(m)_{p,q}$, $m \in \mathcal{M}$, $q, p \in \mathcal{P}$

State:

$current \in \mathcal{C}_\perp$ init c_o if $p \in P_0$, else $\perp$
$rnd\text{-}val \in \mathcal{V}$, initially v_p
$decision \in \mathcal{V}$, initially `nil`
$used\text{-}ids \subseteq \mathcal{I}$, initially $\emptyset$
$ack\text{-}q$, $seqof(A \times \mathcal{I})$, init λ
$ack\text{-}d$, either "*ack*" or `nil`, init `nil`

$last\text{-}rnd \in \mathcal{G}$, initially g_0 if $p \in P_0$, else $\perp$
$last\text{-}val \in \mathcal{V}$, initially v_p
$status \in \{normal, exch\text{-}ready, exch\text{-}wait\}$, init *normal*
$mode \in \{wait, newround, collect, begincast, decided\}$,
init *newround* for $p = c_0.ldr$, *wait* else
$ack\text{-}success(q)$, a boolean, initially `false` for all $q \in \mathcal{P}$

Actions:

input $\text{NEWCONF}(c)_p$
Eff: $current := c$
$status := exch\text{-}ready$
$ack\text{-}q := \lambda$
if $p = c.ldr$ and $mode \neq decided$ then
$mode := newround$

output $\text{SUBMIT-STATE}(\langle r, v\rangle, \phi_{state})_p$
Pre: $status = exch\text{-}ready$
$r = last\text{-}rnd$
$v = last\text{-}val$
Eff: $status := exch\text{-}wait$

input $\text{NEWSTATE}(\langle r, v\rangle)_p$
Eff: $status := normal$
$last\text{-}rnd := current.id$
$last\text{-}val := v$
$rnd\text{-}value := v$
$Hfrom(current.id) = r$
$Hvalue(current.id) = v$

internal NEWROUND_p
Pre: $mode = newround$
$status = normal$
Eff: $mode := begincast$

output $\text{SUBMIT}(\langle \text{"}begin\text{"}, v\rangle, \phi_{begin}, i)_p$
Pre: $current \neq \perp$
$i \notin used\text{-}ids$
$status = normal$
$v = rnd\text{-}value$
$mode = begincast$
Eff: $used\text{-}ids := used\text{-}ids \cup \{i\}$
$mode := wait$

input $\text{DELIVER}(\langle \text{"}begin\text{"}, v\rangle, i)_p$
Eff: append $(\langle \text{"}accept\text{"}\rangle, i)$ to $ack\text{-}q$
$last\text{-}rnd := current.id$
$last\text{-}val := v$

output $\text{ACKDLVR}(a, i)_p$
Pre: $head(ack\text{-}q) = (a, i)$
Eff: $ack\text{-}q := tail(ack\text{-}q)$

input $\text{RESPOND}(\langle \text{"}begin\text{"}, Q\rangle, i)_p$
Eff: $decision := rnd\text{-}value$
$mode := decided$
$Haccquo(current.id) := Q$

output $\text{P2P-SEND}(v)_{p,q}$
Pre: $mode = decided$
$v = decision$
$ack\text{-}success(q) =$ `false`
Eff: none

input $\text{P2P-RECV}(v)_{q,p}$
Eff: $decision := v$
$mode := decided$
$ack\text{-}d := \text{"}ack\text{"}$

output $\text{P2P-SEND}(\text{"}ack\text{"})_{p,q}$
Pre: $ack\text{-}d = \text{"}ack\text{"}$
Eff: $ack\text{-}d :=$ `nil`

input $\text{P2P-RECV}(\text{"}ack\text{"})_{q,p}$
Eff: $ack\text{-}success(q) :=$ `true`

Figure 8: The DLC-TO-PAXOS code.