From Total Order to Database Replication

Yair Amir and Ciprian Tutu
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218, USA
{yairamir, ciprian}@cnds.jhu.edu

Technical Report
CNDS-2001-6
http://www.cnds.jhu.edu

November 5, 2001

Abstract

This paper presents in detail an efficient and provably correct algorithm for database replication over partitionable networks. Our algorithm avoids the need for end-to-end acknowledgments for each action while supporting network partitions and merges and allowing dynamic instantiation of new replicas. One round of end-to-end acknowledgments is required only upon a membership change event such as a network partition. New actions may be introduced to the system at any point, not only while in a primary component. We show how performance can be further improved for applications that allow relaxation of consistency requirements. We provide experimental results that demonstrate the superiority of this approach.

1 Introduction

Database replication is quickly becoming a critical tool for providing high availability, survivability and high performance for database applications. However, to provide useful replication one has to solve the non-trivial problem of maintaining data consistency between all the replicas.

The state machine approach [27] to database replication ensures that replicated databases that start consistent will remain consistent as long as they apply the same deterministic actions (transactions) in the same order. Thus, the database replication problem is reduced to the problem of constructing a global persistent consistent order of actions. This is often mistakenly considered easy to achieve using the Total Order service (e.g. ABCAST, Agreed order, etc) provided by group communication systems.

Early models of group communication, such as Virtual Synchrony, did not support network partitions and merges. The only failures tolerated by these models were process crashes, without recovery. Under these circumstances, total order is sufficient to create global persistent consistent order.

Unfortunately, almost no real-world system today adheres to the requirement of never having network partitions. Even in local area networks, network partitions occur regularly due to either hardware (e.g. temporarily disconnected switches) or software (heavily loaded servers). Of course, in wide area networks, partitions can be common [5].

When network partitions are possible, total order service does not directly translate to a global persistent consistent order. Existing solutions that provide active replication either avoid dealing with network partitions [29, 24, 23] or require additional end-to-end acknowledgements for every action after it is delivered by the group communication and before it is admitted to the global consistent persistent order (and can be applied to the database) [16, 12, 28].

In this paper we present a complete and provably correct algorithm that provides global persistent consistent order in a partitionable environment without the need for end-to-end acknowledgments on a per action basis. In our approach end-to-end acknowledgements are only used once for every network connectivity change event (such as network partition or merge) and not per action. The basic concept, though never published, was first introduced as part of a PhD dissertation [2]. This paper presents our newly developed insight into the problem and goes beyond [2] by supporting online additions of completely new replicas and complete removals of existing replicas while the system executes.

Our algorithm does not require changes to existing databases to support replication. Instead it builds a generic replication engine which runs outside the database and can be seamlessly integrated with existing databases and applications. The replication engine supports various semantic models, relaxing or enforcing the consistency constraints as needed by the application. We have implemented the replication engine on top of the Spread toolkit [4, 3] and provide experimental performance results, comparing the throughput and latency of the global consistent persistent order using our algorithm, the COReL algorithm introduced in [16], and the standard two-phase commit algorithm. These results demonstrate the power of eliminating the end-to-end acknowledgments on a per-action basis.

The rest of the paper is organized as follows. The following subsection discusses related work. Section 2 describes the working model. Section 3 introduces a conceptual solution. Section 4 addresses the problems exhibited by the conceptual solution in a partitionable system and introduces the Extended Virtual Synchrony model as a tool to provide global persistent order. Section 5 describes the detailed replication algorithm and extends it to support online removals and additions to the set of participating replicas. Section 6 shows how the global persistent order guarantees of the algorithm can be used to support various relaxed consistency requirements useful for database replication. Section 7 evaluates the performance of our prototype, while Section 8 concludes the paper. Appendix A presents the complete pseudo-code of the static replication algorithm.

1.1 Related Work

Two-phase commit protocols [12] remain the main technique used to provide a consistent view in a distributed replicated database system over an unreliable network. These protocols impose a substantial communication cost on each transaction and may require the full connectivity of all replicas to recover from some fault scenarios. Three-phase-commit protocols [28, 17] overcome some of the availability problems of two-phase-commit protocols, paying the price of an additional communication round.

Some protocols optimize for specific cases: limiting the transactional model to commutative transactions [26]; giving special weight to a specific processor or transaction [30]. Explicit use of timestamps enables other protocols [6] to avoid the need to claim locks or to enforce a global total order on actions, while other solutions settle for relaxed consistency criteria [11]. Various people investigated methods to implement efficient lazy replication algorithms by using epidemic propagation [8, 14] or by exploiting application semantics [22].

Atomic Broadcast [13] in the context of Virtual Synchrony [7] emerged as a promising tool to solve the replication problem. Several algorithms were introduced [29, 24, 25, 23] to implement replication solutions based on total ordering. All these approaches, however, work only in the context of non-partitionable environments.

Keidar [16] uses the Extended Virtual Synchrony (EVS) [21] model to propose an algorithm that supports network partitions and merges. The algorithm requires that each transaction message is end-to-end acknowledged, even when failures are not present, thus increasing the latency of the protocol. In section 7 we demonstrate the impact of these end-to-end acknowledgements on performance by comparing this algorithm with ours. Fekete, Lynch and Shvartsman [9] study both [16] and [2] (which is our static algorithm) to propose an algorithm that translates View Synchrony, another specification of a partitionable group service, defined in the same work, into a global total order.

Kemme, Bartoli and Babaoglu[20] study the problem of online reconfiguration of a replicated system in the presence of network events, which is an important building block for a replication algorithm. They propose various useful solutions to performing the database transfer to a joining site and provide a high-level description of an online reconfiguration method based on Enriched Virtual Synchrony allowing new replicas to join the system if they are connected with the primary component. Our solution can leverage from any of their database transfer techniques and adds to that in its detail, accuracy and proof of correctness as well as the capability to allow new sites to join the running system without the need to be connected to the primary component.

Kemme and Alonso [19] present and prove the correctness for a family of replication protocols that support different application semantics. The protocols are introduced in a failure-free environment and then enhanced to support server crashes and recoveries. The model used does not allow network partitions, always assuming disconnected sites to be crashed. In their model, the replication protocols rely on external view-change protocols that provide uniform reliable delivery in order to provide consistency across all sites. In our work we show that the transition from the group communication uniform delivery notification to the strict database consistency is not trivial and we provide a detailed algorithm for this purpose and prove its correctness. In [18], Kemme and Alonso provide valuable experimental results for the integration of their replication methods into the Postgres database system (although they do not provide a detailed algorithm). In contrast, our algorithm is completely decoupled from the database mechanisms in order to offer seamless integration with any kind of database or application. We think that combining our engine with their techniques of database integration could outperform either method.

2 System Model

The system consists of a set of nodes (servers) S=$\{S_1, S_2, ..., S_n\}$, each holding a copy of the entire database. Initially we assume that the set S is fixed and known in advance. Later, in Section 5.1, we will show how to deal with online changes to the set of potential replicas[1].

2.1 Failure and Communication Model

The nodes communicate by exchanging messages. The messages can be lost, servers may crash and network partitions may occur. We assume no message corruption and no Byzantine faults.

A server that crashes may subsequently recover. Upon recovery, a server retains its old identifier and stable storage. Each node executes several processes: a database server, a replication engine and

[1]Note that these are changes to the system setup, not membership changes caused by temporary network events.

a group communication layer. The crash of any of the components running on a node will be detected by the other components and treated as a global node crash.

The network may partition into a finite number of disconnected components. Nodes situated in different components cannot exchange messages, while those situated in the same component can continue communicating. Two or more components may subsequently merge to form a larger component.

We employ the services of a *group communication layer* which provides reliable multicast messaging with ordering guarantees (FIFO, causal, total order). The group communication system also provides a membership notification service, informing the replication engine about the nodes that can be reached in the current component. The notification occurs each time a connectivity change, a server crash or recovery, or a voluntary join/leave occurs. The set of participants that can be reached by a server at a given moment in time is called a *view*. The replication layer handles the server crashes and network partitions using the notifications provided by the group communication. The basic property provided by the group communication system is called Virtual Synchrony [7] and it guarantees that processes moving together from one view to another deliver the same (ordered) set of messages in the former view. (We will see in Section 4 that Virtual Synchrony alone is not sufficient for our purposes.)

2.2 Service Model

A *Database* is a collection of organized, related data that can be accessed and manipulated through a *database management system*. Clients access the data by submitting *transactions*. A transaction consists of a set of commands and has to follow the ACID properties.

A replication service maintains a replicated database in a distributed system. Each server from the server set maintains a private copy of the database. The initial state of the database is identical at all servers. Several models of consistency can be defined for a replicated database, the strictest of which is *one-copy serializability*. One-copy serializability requires that the concurrent execution of transactions on a replicated data set is equivalent to a serial execution on a non-replicated data set. We are focusing on enforcing the strict consistency model, but we also support weaker models (see Section 6).

An *action* defines a transition from the current state of the database to the next state; the next state is completely determined by the current state and the action. We view actions as having a *query* part and an *update* part, either of which can be missing. Client transactions will translate into actions that are applied to the database. The basic model best fits one-operation transactions, but as we show in Section 6, active actions and interactive actions can be supported as well.

3 Replication Algorithm

In the presence of network partitions, the replication layer identifies at most a single component of the server group as a *primary component*; the other components of a partitioned group are *non-primary components*. A change in the membership of a component is reflected in the delivery of a view-change message by the group communication layer to each server in that component. The replication layer implements a symmetric distributed algorithm to determine the order of actions to be applied to the database. Each server builds its own knowledge about the order of actions in the system. We use the coloring model defined in [1] to indicate the knowledge level associated with each action. Each server marks the actions delivered to it with one of the following colors:

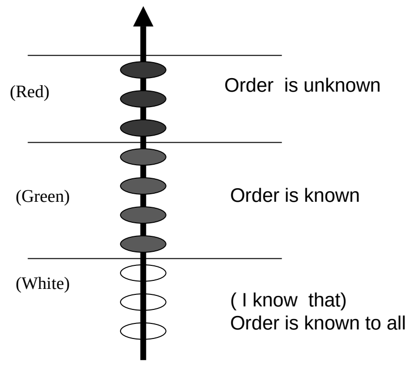


Figure 1: Action coloring

Red Action An action that has been ordered within the local component by the group communication layer, but for which the server cannot, as yet, determine the global order.

Green Action An action for which the server has determined the global order.

White Action An action for which the server knows that all of the servers have already marked it as *green*. These actions can be discarded since no other server will need them subsequently.

At each server, the *white* actions precede the *green* actions which, in turn, precede the *red* ones. An action can be marked differently at different servers; however, no action can be marked *white* by one server while it is missing or is marked *red* at another server.

The actions delivered to the replication layer in a primary component are marked green. Green actions can be applied to the database immediately while maintaining the strictest consistency requirements. In contrast, the actions delivered in a non-primary component are marked red. The global order of these actions cannot be determined yet, so, under the strong consistency requirements, these actions cannot be applied to the database at this stage.

3.1 Conceptual Algorithm

The algorithm presented in this section should, intuitively, provide an adequate solution to the replication problem. While this is not actually the case, as the algorithm is not able to deal with some of the more subtle issues that can arise in a partitionable system, we feel that presenting this simplified solution provides a better insight into some of the problems the complete solution needs to cope with and also introduces the key properties of the algorithm.

Figure 2 presents the state machine associated with the conceptual algorithm. A replica can be in one of the following four states:

- **Prim State.** The server belongs to the primary component. When a client submits a request, it is multicast using the group communication to all the servers in the component. When a message is delivered by the group communication system to the replication layer, the action is immediately marked green and is applied to the database.

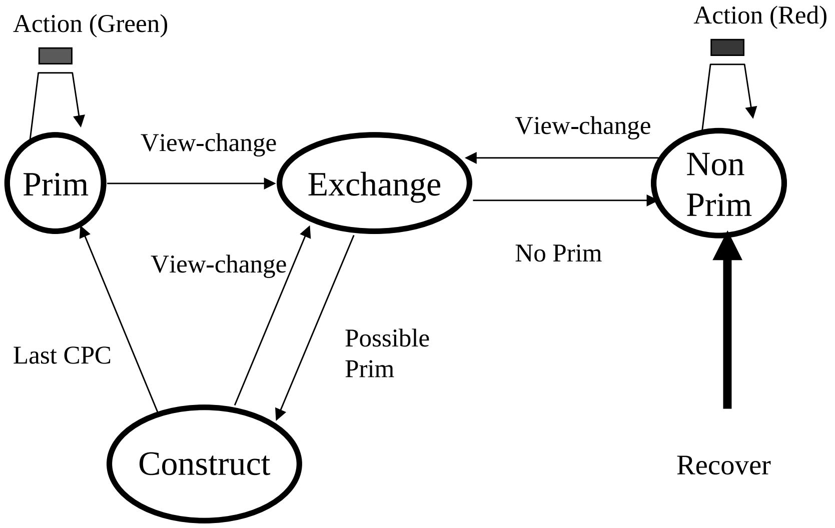


Figure 2: Conceptual Replication Algorithm

- **NonPrim State.** The server belongs to a non-primary component. Client actions are ordered within the component using the group communication system. When a message containing an action is delivered by the group communication system, it is immediately marked red.

- **Exchange State.** A server switches to this state upon delivery of a view change message from the group communication system. All the servers in the new view will exchange information allowing them to define the set of actions that are known by some of them but not by all. These actions are subsequently exchanged and each server will apply to the database the green actions that it gained knowledge of. After this exchange is finished each server can check whether the current view has a quorum to form the next primary component. This check can be done locally, without additional exchange of messages, based on the information collected in the initial stage of this state. If the view can form the next primary component the server will move to the Construct state, otherwise it will return to the NonPrim state.

- **Construct State.** In this state, all the servers in the component have the same set of actions (they synchronized in the Exchange state) and can attempt to install the primary component. For that they will send a Create Primary Component (CPC) message. When a server has received CPC messages from all the members of the current component it will transform all its red messages into green, apply them to the database and then switch to the Prim state. If a view change occurs before receiving all CPC messages, the server returns back to the Exchange state.

For most of the execution of the algorithm, the servers will reside in either the Prim or the NonPrim state. While in these states, there is no need for end-to-end acknowledgements as the group communication layer guarantees that all the servers will receive the same set of messages, in the same order.

In a system that is subject to partitioning, we must ensure that two different components do not apply contradictory actions to the database. We use a quorum mechanism to allow the selection of a unique primary component from among the disconnected components. Only the servers in the primary component will be permitted to apply actions to the database. While several types of quorums could be used, we opted to use *dynamic linear voting* [15]. Under this system, the component that contains a (weighted) majority of the last primary component becomes the new primary component.

In many systems processes exchange information only as long as they have a direct and continuous connection. In contrast, our algorithm propagates information by means of *eventual path*. According to this concept, when a new component is formed, the servers exchange knowledge regarding the actions they have, their order and color. The method for sharing this information is efficient because the exchange process is only invoked immediately after a view change. Furthermore, all the components exhibit this behavior, whether they will form a primary or non-primary component. This allows the information to be disseminated even in non-primary components, reducing the amount of data exchange that needs to be performed once a server joins the primary component.

4 From Total Order to Database Replication

Unfortunately, due to the asynchronous nature of the system model, we cannot reach complete common knowledge about which messages were received by which servers just before a network partition occurs or a server crashes. In fact, it has been proven that reaching consensus in asynchronous environments with the possibility of even one failure is impossible [10]. Group communication primitives based on Virtual Synchrony do not provide any guarantees of message delivery that span network partitions and server crashes. In our algorithm it is important to be able to tell whether a message that was delivered to one server right before a view change, was also delivered to all its intended recipients.

A server p cannot know, for example, whether the last actions it delivered in the Prim state, before a view-change event occurred, were delivered to all the members of the primary component; Virtual Synchrony guarantees this fact only for the servers that will install the next view together with p. These messages cannot be immediately marked *green* by p, because of the possibility that a subset of the initial membership, big enough to construct the next primary component, did not receive the messages. This subset will install the new primary component and then apply other actions as green to the database, breaking consistency with the rest of the servers. This problem will manifest itself in any algorithm that tries to operate in the presence of network partitions and remerges. A solution based on Total Order cannot be correct in this setting without further enhancement. Similarly, in Construct state, if another membership change occurs, the servers must decide whether the new primary component was installed or not, which is equivalent to the consensus problem and therefore impossible. The algorithm would become incorrect if one server would decide that the primary component was installed while another will conclude the opposite.

Thus the algorithm presented in Section 3.1 is insufficient to cope with a partitionable asynchronous environment.

4.1 Extended Virtual Synchrony

In order to circumvent the inability to know who received the last messages sent before a network event occurs we use an enhanced group communication paradigm called *Extended Virtual Synchrony* (EVS) [21]. EVS splits the view-change notification into two notifications: a *transitional* configuration change message and a *regular* configuration change message. The transitional configuration message defines a reduced membership containing members of the next regular configuration coming directly from the same regular configuration. This allows the introduction of another form of message delivery, *safe delivery*, which maintains the total order property but also guarantees that every message delivered to any process that is a member of a configuration is delivered to every process that is a member of that configuration, unless that process fails. Messages that do not meet the requirements for safe delivery, but are received by the group communication system, are delivered in the transitional configuration. No messages are sent by the group communication in the transitional configuration.

The safe delivery property provides a valuable tool to deal with the incomplete knowledge in the presence of network failures or server crashes. Instead of having to decide on one of two possible values, as in the consensus problem, we have now three possible values/situations:

1. A safe message is delivered in the regular configuration. All guarantees are met and everyone in the configuration will deliver the message (either in the regular configuration or in the following transitional configuration) unless they crash.

2. A safe message is delivered in the transitional configuration. This message was received by the group communication layer just before a partition occurs. The group communication layer cannot tell whether other components that split from the previous component received and will deliver this message.

3. A safe message was sent just before a partition occurred, but it was not received by the group communication layer in some detached component. The message will, obviously, not be delivered at this component.

The power of this differentiation lies in the fact that, with respect to the same message, it is impossible for one server to be in situation 1, while another is in situation 3.

To illustrate the use of this property consider the Construct phase of our algorithm: If a server p receives all CPC messages in the regular configuration, it knows that every server in that configuration will receive all the messages before the next regular configuration is delivered, unless they crash; some servers may, however, receive some of the CPC messages in a transitional configuration. Conversely, if a server q receives a configuration change for a new regular configuration before receiving all of the CPC messages, then no server could have received a message that q did not receive as safe in the previous configuration. In particular, no server received all of the CPC messages as safe in the previous regular configuration. Thus q will know that it is in case 3 and no other server is in case 1.

Finally, if a server r received all CPC messages, but some of those were delivered in a transitional configuration, then r cannot know whether there is a server p that received all CPC messages in the regular configuration or whether there is a server q that did not receive some of the CPC messages at all; r does, however, know that there cannot exist both a p and a q as described.

5 Replication Algorithm

Based on the above observations the algorithm skeleton presented in Section 3.1 needs to be refined. We will take advantage of the Safe delivery properties and of the differentiated view change notification that EVS provides. The two vulnerable states are, as mentioned, Prim and Construct.[2]

[2]While the same problems manifest themselves in any state, it is only these two states where knowledge about the message delivery is critical, as it determines either the global total order (in Prim) or the creation of the new primary (Construct).

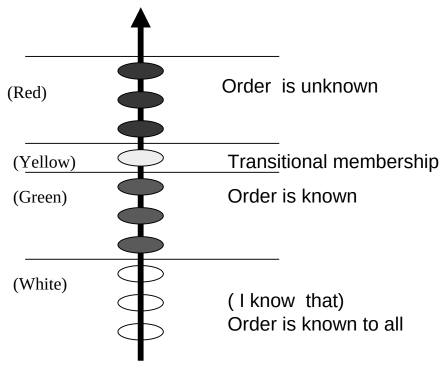


Figure 3: Updated coloring model

In the **Prim** state, only actions that are delivered as safe during the regular configuration can be applied to the database. Actions that were delivered in the transitional configuration cannot be marked as green and applied to the database before we know that the next regular configuration will be the one defining the primary component of the system. If an action a is delivered in the transitional membership and is marked directly as green and applied to the database, then it is possible that one of the detached components that did not receive this action will install the next primary component and will continue applying new actions to the database, without applying a, thus breaking the consistency of the database. To avoid this situation, the **Prim** state was split into two states: ***RegPrim*** and ***TransPrim*** and a new message color was introduced to the coloring model:

Yellow Action An action that was delivered in a transitional configuration of a primary component.

A *yellow* action becomes *green* at a server as soon as this server learns that another server marked the action *green* or when this server becomes part of the primary component. As discussed in the previous section, if an action is marked as *yellow* at some server p, then there cannot exist two servers r and s such that one marked the action as *red* and the other marked it *green*.

In the presence of consecutive network changes, the process of installing a new primary component can be interrupted by another configuration change. If a transitional configuration is received by a server p while in the **Construct** state, before receiving all the CPC messages, the server will switch to a new state: ***No***. In this state, as far as p knows, no other server has installed the primary component by receiving all the CPC messages in the **Construct** state, although this situation is possible. Therefore, p basically expects the delivery of the new regular configuration which will trigger the initiation of a new exchange round. However, if p receives all the rest of the CPC messages in **No** (in the transitional configuration), it means that it is possible that some server q has received all CPC messages in **Construct** and has moved to **Prim**.

To account for this possibility, p will switch to another new state: ***Un*** (undecided). If an action message is received in this state then p will know for sure that there was a server q that switched to **RegPrim** and even managed to generate new actions before noticing the network failure that caused the cascaded membership change. Server *p,* in this situation (1b), has to act as if installing the primary component in order to be consistent, mark its old yellow/red actions as green, mark the received action as yellow and switch to **TransPrim**, “joining” q who will come from **RegPrim** as it will also eventually notice the new configuration change. If the regular configuration message is

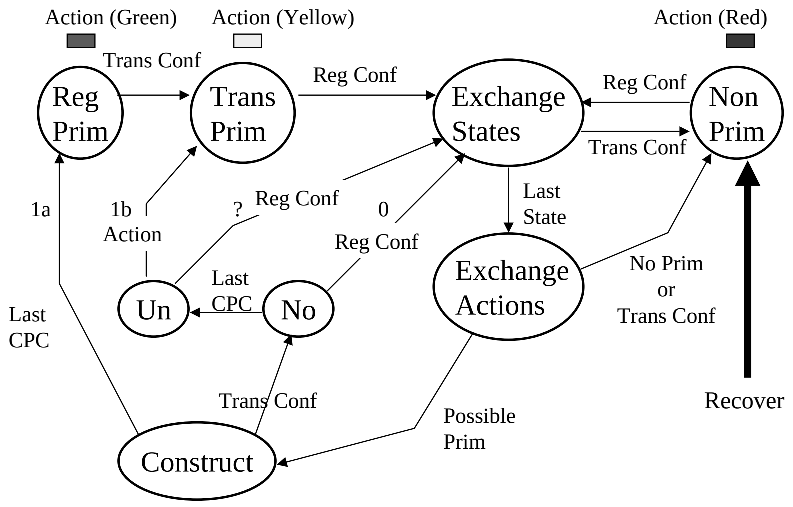


Figure 4: Replication Algorithm

delivered without any message being received in the **Un** state (transition marked ? in Figure 4), p remains uncertain whether there was a server that installed the primary component. Until this dilemma is cleared through future exchange of information, p will remain *vulnerable,* signifying that it was possibly part of a primary component but it did not perform the installment procedure.

The *vulnerable* flag plays a very important role for the correctness of the algorithm. A server that agrees to the forming of a new primary component (by generating a CPC message) will mark itself *vulnerable* on its stable storage. This signifies that the server does not know how the creation of the primary component ended or, in case the primary component was created, what messages were delivered in that primary component. If this server crashed while vulnerable, there is a risk that safe messages were delivered in the primary component, but this server crashed before processing them and therefore it has no recollection of these messages on its persistent storage. Therefore the server should not present itself as a "knowledgeable" member of that primary component upon recovery. The server ceases to be vulnerable when it has on persistent storage the complete knowledge regarding the primary component he was vulnerable to. If all the servers in the primary component crash (before any of them processes a configuration change), then they all need to exchange information with each other before continuing, in order to guarantee consistency. This closes the gap between the group communication notification and the information maintained on persistent storage that will survive crashes.

Figure 4 shows the updated state machine. Aside from the changes already mentioned, the Exchange state was also split into **ExchangeStates** and **ExchangeActions**, mainly for clarity reasons. From a procedural point of view, once a view change is delivered, the members of each view will try to establish a maximal common state that can be reached by combining the information and actions held by each server. After the future common state is determined, the participants proceed to exchange the relevant actions. Obviously, if the new membership is a subset of the old one, there is no need for action exchange, as the states are already synchronized.

The complete pseudo-code of the algorithm is attached in Appendix A.

5.1 Dynamic Replica Instantiation and Deactivation

As mentioned in the description of the model, the algorithm that we presented so far works under the limitation of a fixed set of potential replicas. It is of great value, however, to allow for the dynamic instantiation of new replicas as well as for their deactivation. Furthermore, if the system does not support permanent removal of replicas, it is susceptible to blocking in case of a permanent failure or disconnection of a majority of nodes in the primary component.

However, dynamically changing the set of servers is not straightforward: the set change needs to be synchronized over all the participating servers in order to avoid confusion and incorrect decisions such as two distinct components deciding they are the primary, one being the rightful one in the old configuration, the other being entitled to this in the new configuration. Since this is basically a consensus problem, it cannot be solved in a traditional fashion. We circumvent the problem with the help of the persistent global total order that the algorithm provides.

CodeSegment 5.1 Online reconfiguration in the replication algorithm

```
MarkGreen (Action)
  MarkRed(Action)
  if (Action not green)
      place Action just on top of the last green action
      greenLines[ serverId ] = Action.action_id
      if (Action.type == PERSISTENT_JOIN && Action.join_id not in local structures)
          extend greenLines, redCut to include new server id
          greenLines[Action.join_id] = Action.action_id
          if (Action.action_id == serverId)
              start database transfer to joining site
      elsif (Action.type == PERSISTENT_LEAVE & & Action.leave_id is in local structures)
          reduce greenLines, redCut to exclude Action.leave_id
          if (Action.leave_id == serverId) exit
      else
          ApplyGreen( Action )
When new server initiates connection
  if (state == Prim) or (state == NonPrim)
      if (new server not in local data structures)
          create PERSISTENT_JOIN action
          generate action
      else
          continue database transfer to joining site
When replica wants to leave the system
  if (state == Prim) or (state == NonPrim)
      create PERSISTENT_LEAVE action
      generate action
```

Algorithm 5.1 shows the modifications that need to be added to the replication engine described in 5 to support online reconfiguration. The pseudo-code is presented in the format used in Appendix A where we show the complete code of the algorithm and we describe the meaning of the variables used throughout. Algorithm 5.2 shows the actions that need to be performed by the joining site before it can join the replicated system and start executing the replication algorithm.

CodeSegment 5.2 Joining the replicated system

```
25  while not updated
26     (re)connect to server in the system
27     transfer database
28  set greenLines[serverId] to the action_id given in the system to the PERSISTENT_JOIN action.
29  state = NonPrim
30  join replicated group and start executing replication algorithm.
```

When a replica wants to permanently leave the system, it will broadcast a PERSISTENT_LEAVE message (lines 22-24) that will be ordered together with the rest of the actions. When this message becomes *green* at a replica, the replica can update its local data structures to exclude the parting member (lines 11-12). The PERSISTENT_LEAVE message can also be administratively inserted into the system to signal the permanent removal, due to failure, of one of the replicas. The message will be issued by a site that is still in the system and will contain the server id of the dead replica.[3]

A new replica that wants to join the replicated system will first need to connect to one of the members of the system. This server will act as a representative for the new site to the existing group by creating a PERSISTENT_JOIN message to announce the new site (lines 18-19). This message will be ordered as a regular action, according to the standard algorithm. When the message becomes *green* at a server, that replica will update its data structures to include the newcomer's server id and set the green line (the last globally ordered message that the server has) for the joining member as the action corresponding to the PERSISTENT_JOIN message (lines 5-7). Basically, from this point on the servers acknowledge the existence of the new member, although it did not actually join the system by connecting to the replicated group. When the PERSISTENT_JOIN message becomes green at the peer server (the representative), the peer server will take a snapshot of the database and start transferring it to the joining member (lines 9-10). If the initial peer fails or a network partition occurs before the transfer is finished, the new server will try to establish a connection with a different member of the system and continue its update. If the new peer already ordered the PERSISTENT_JOIN message sent by the first representative, it will know about the new server (line 17) and the state that the new server has to reach before joining the system and will be able to resume the transfer procedure (line 21). If the new peer has not yet ordered the PERSISTENT_JOIN message it will issue another PERSISTENT_JOIN message for the new site. PERSISTENT_JOIN messages for members that are already present in the local data structures are ignored by the existing servers, therefore only the first ordered PERSISTENT_JOIN will define the entry point of the new site into the system. Finally, when the transfer is complete, the new site will start executing the replication algorithm by joining the replica group and becoming part of the system.

Another method for performing online reconfiguration is described in [20]. This method requires the joining site to be permanently connected to the primary component while being updated. We maintain the flexibility of the engine and we allow joining replicas to be connected to non-primary components during their update stage. It can even be the case that a new site is accepted into the system without **ever** being connected to the primary component, due to the eventual path propagation method. The insertion of a new replica into the system, even in a non-primary component, can be useful to certain applications as is shown in Section 6.

[3]Securing this mechanism to avoid malicious use of the PERSISTENT_LEAVE message is outside the scope of this paper.

5.2 Proof of Correctness

The algorithm in its static form was proven correct in [2]. The correctness properties that were guaranteed were liveness, FIFO order and Total global order. Here, we prove that the enhanced dynamic version of the algorithm still preserves the same guarantees.

Lemma 1 (Global Total Order (static))
If both servers s and r performed their ith actions, then these actions are identical.

Lemma 2 (Global FIFO Order (static))
If server r performed an action a generated by server s, then r already performed every action that s generated prior to a.

These are the two properties that define the **Safety** criterion in [2]. These specifications need to be refined to encompass the removal of servers or the addition of new servers to the system.

Theorem 1 (Global Total Order (dynamic))
If both servers s and r performed their ith action, then these actions are identical.

Proof: Consider the system in its start-up configuration set. Any server in this configuration will trivially maintain this property according to Lemma 1. Consider a server s that joins the system. The safety properties of the static algorithm guarantee that after ordering the same set of actions, all servers will have the same consistent database. This is the case when a PERSISTENT_JOIN action is ordered. According to the algorithm s will set its global action counter to the one assigned by the system to the PERSISTENT_JOIN action (line 4 in algorithm 5.2). From this point on the behavior of s is indistinguishable from a server in the original configuration and the claim is maintained as per Lemma 1. $\square$

Theorem 2 (Global FIFO Order (dynamic))
If server r performed an action a generated by server s, then r already performed every action that s generated prior to a, or it inherited a database state which incorporated the effect of these actions.

Proof: According to Lemma 2, the theorem holds true from the initial starting point until a new member is added to the system. Consider r, a member who joins the system. According to the algorithm, the joining member transfers the state of the database as defined by the action ordered immediately before the PERSISTENT_JOIN message. All actions generated by s and ordered before the PERSISTENT_JOIN will be incorporated in the database that r received. From Theorem 1, the PERSISTENT_JOIN message is ordered at the same place at all servers. All actions generated by s and ordered after the PERSISTENT_JOIN message will be ordered similarly at every server, including r, according to Theorem 1. Since Lemma 2 holds for any other member, this is sufficient to guarantee that r will order all other actions generated by s prior to a, and ordered after r joined the system. $\square$

Lemma 3 (Liveness (static))
If server s orders action a and there exists a set of servers containing s and r, and a time from which on that set does not face any communication or process failures, then server r eventually orders action a.

This is the liveness property defined in [2] and proven to be satisfied by the static replication algorithm. This specification needs to be refined to include the notion of servers permanently leaving the system.

Theorem 3 (Liveness (dynamic))
If server s orders action a in a configuration that contains r and there exists a set of servers containing s and r, and a time from which on that set does not face any communication or process failures, then server r eventually orders action a.

Proof: The theorem is a direct extension of Lemma 3, which acknowledges the potential existence of different server-set configurations. An action that is ordered by a server in one configuration will be ordered by all servers in the same configuration as a direct consequence of Theorem 1. Servers that leave the system or crash do not meet the requirements for the liveness property, while servers that join the system will order the actions generated in any configuration that includes them, unless they crash. □

6 Supporting Various Application Semantics

The presented algorithm was designed to provide strict consistency semantics by applying actions to the database only when they are marked green. Thus, the actions delivered to the replication layer in a primary component can be applied to the database immediately. In contrast, the actions delivered in a non-primary component are marked red. The global order of these actions cannot be determined yet, so, if we require strong consistency, these actions cannot be applied to the database at this stage. Under this model, even queries issued while in a non-primary component cannot be answered until the connectivity with the primary component is restored.

In the real world, however, where incomplete knowledge is unavoidable, many applications would rather have an immediate answer, than incur a long latency to obtain a complete and consistent answer. Therefore, we provide additional service types for clients in a non-primary component.

The result of a *weak query* is obtained from a consistent, but possibly obsolete state of the database, as reflected by the green actions known to the server at the time of the query. The *weak consistency* service, when requested by an application, will allow the replication engine to reply to a query delivered while in a non-primary component. Updates, however, will not be allowed (will be delayed) until the server joins a primary component. This may result in a client requesting some updates to the database, then querying the database and getting an old result which does not reflect the updates it just made. Still, this is acceptable for some applications.

Other applications would rather get an immediate reply based on the latest information available. In the primary component this information is reflected in the state of the database and is always consistent. In a non-primary component, however, red actions must be taken into account in order to provide the latest, though not consistent, information. We call this type of query a *dirty query*. To provide this service, a *dirty* version of the database is maintained while the replicas are not in the primary component.

Different semantics can be supported also with respect to updates. Two examples would be the *timestamp* update semantics and the *commutative* update semantics. In the timestamp case, all updates are timestamped and the application only wants the information with the highest timestamp. Therefore the actions don't need to be ordered. Location tracking is a good example of an application that would employ such semantics. Similarly, in the commutative case, the order is irrelevant as long as all actions are eventually applied to the database. Consider an inventory model (where temporary negative stock is allowed); all operations on the stock would be commutative. For both semantics, the one-copy serializability property is not maintained in the presence of network partitions. However, after the network is repaired and the partitioned components merge, the databases states converge.

Regardless of the semantics involved, the algorithm can be optimized if it has the ability to distinguish a query-only action from an action that contains updates. A query issued at one server can be answered as soon as all previous actions generated by this server were applied to the database, without the need to generate and order an action message.

Modern database applications exploit the ability to execute a procedure specified by a transaction. These are called *active* transactions and they are supported by our algorithm, provided that the invoked procedure is deterministic and depends solely on the current database state. The key is that the procedure will be invoked at the time the action is ordered, rather than before the creation of the update.

Finally, we mentioned that our model best fits one-operation transactions. Some applications need to use interactive transactions which, within the same transaction, read data and then perform updates based on a user decision, rather than a deterministic procedure. Such behavior, cannot be modeled using one action, but can be mimicked with the aid of two actions. The first action will read the necessary data, while the second one will be an active action as described above. This active action will encapsulate the update dictated by the user, but will first check whether the values of the data read by the first action are still valid. If not, the update will not be applied, as if the transaction was aborted in the traditional sense. Note that if one server "aborts", all of the servers will abort that (trans)action, since they apply an identical deterministic rule to an identical state of the database.

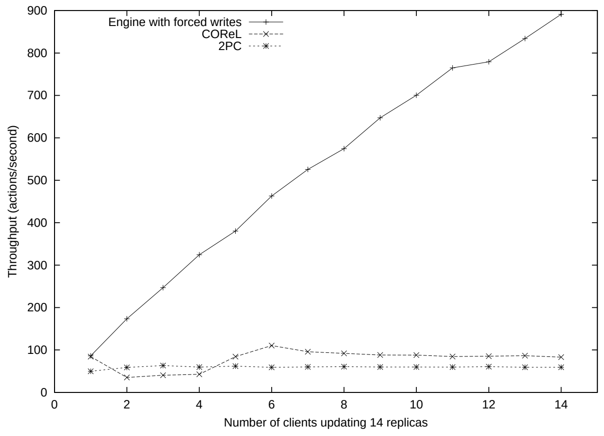
7 Performance Analysis

In this section we provide a practical evaluation of the replication engine and compare its performance to that of two existing solutions. All our tests were conducted with 14 replicas, each of which ran on a dual processor Pentium III-667 computer running Linux connected by a 100Mbits/second local area network. Each action is contained in 200 bytes (e.g. an SQL statement).

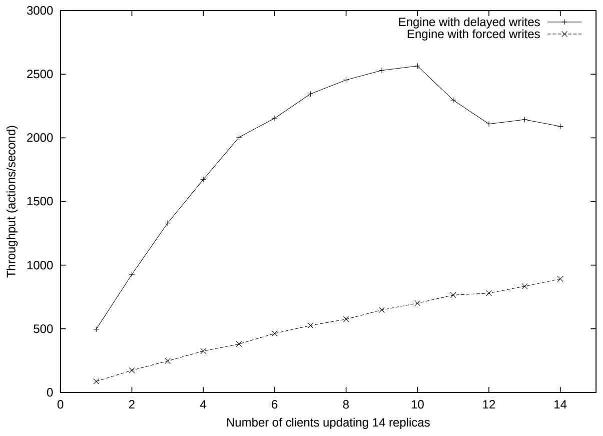
Two-phase commit is adopted by most replicated systems that require strict consistency. This algorithm however pays the price for its simplicity by requiring two forced disk writes and 2n unicast messages per action. Keidar [16] designed a consistent object replication algorithm (COReL) that exploits some of the group communication benefits to improve on the performance of traditional two-phase commit algorithms. In this algorithm only one forced disk write and n multicast messages per action are necessary. Our algorithm only requires one forced disk write and one multicast message per action.

We have implemented all three algorithms and we compared their performance while running in normal configuration when no failures occur. Since we were interested in the intrinsic performance of the replication engines, clients receive responses to their actions when the actions are globally ordered, without any interaction with a database.

Figure 5(a) presents a comparison of the throughput that a system of 14 replicas is able to sustain while running the three algorithms. We vary the number of clients that simultaneously submit requests into the system between 1 and 14, in which case each computer has both a replica and a client. The clients are constantly injecting actions into the system, the next action from a client being introduced immediately after the previous action from that client is completed and its result reported to the client. This allows us to increase the amount of actions that need to be served by the system. We notice that, compared to our algorithm, two-phase commit and COReL pay the price for extra communication and disk writes. The extra disk write creates the difference between two-phase commit and COReL under these settings; however, it is expected that on wide area network,



(a) Throughput comparison



(b) Impact of forced disk writes

Figure 5: Throughput Comparison

where network latency becomes a more important factor, COReL will further outperform two-phase commit. Our algorithm was able to sustain increasingly more throughput and has not reached its processing limit under this test. In order to assess this limit and to determine the impact of forced disk writes in a local area environment we ran our algorithm allowing for asynchronous disk writes instead of forced writes. The comparison is shown in Figure 5(b). Our algorithm tops at processing 2500 actions/second. This also shows the potential performance that the engine can sustain in a high-performance environment equipped with fast stable storage medium.

We also compared the latency that a client will detect when connected to a system of replicated servers. For the test we had one client connect to the system and send a set of 2000 actions, sequentially. We recorded the response time for each action and marked the average latency. Since our tests were run on local area network, the impact of network communication was offset by the latency of the disk writes. This explains the quasi-linear behavior of the two-phase commit and COReL algorithms which should otherwise exhibit a linear increase in latency. We noticed, however the impact of the extra disk-write on the two-phase commit algorithm as well as the clearly linear behavior of our algorithm, as predicted. The average latency of the two-phase commit algorithm was around 19.3ms while for the COReL and our replication engine it was around 11.4ms regardless of the number of servers. These numbers are, as we mentioned, driven by the disk-write latency.

8 Conclusions

We presented a complete algorithm for database replication over partitionable networks sophistically utilizing group communication and proved its correctness. Our avoidance of the need for end-to-end acknowledgment per action contributed to superior performance. We showed how to incorporate online instantiation of new replicas and permanent removal of existing replicas. We also demonstrated how to efficiently support various types of applications that require different semantics.

Acknowledgements

We thank Jonathan Stanton for his numerous technical ideas and support that helped us optimize the overall performance of the system. We also thank Michal Miskin-Amir and Jonathan Stanton for their insightful suggestions that considerably improved the presentation of this paper. This work was partially funded by grant F30602-00-2-0550 from the Defense Advanced Research Projects Agency (DARPA).

References

[1] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. *Lecture Notes in Computer Science*, 774:125–139, 1993.

[2] Y. Amir. *Replication Using Group Communication over a Partitioned Network.* PhD thesis, Hebrew University of Jerusalem, Jerusalem, Israel, 1995. http://www.cnds.jhu.edu/publications/yair-phd.ps.

[3] Y. Amir, C. Danilov, and J. Stanton. Loss tolerant architecture and protocol for wide area group communication, 2000.

[4] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, 1998.

[5] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Symposium on Fault-Tolerant Computing*, pages 26–35, 1996.

[6] P.A. Bernstein, D.W. Shipman, and J.B. Rothnie. Concurrency control in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 5(1):18–51, March 1980.

[7] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on OS Principles*, pages 123–138, Austin, TX, USA, November 1987. ACM SIGOPS, ACM.

[8] A. Demers et al. Epidemic algorithms for replicated database maintenance. In Fred B. Schneider, editor, *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, Canada, August 1987. ACM Press.

[9] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[10] M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[11] R. Golding. *Weak-Consistency Group Communication and Membership.* PhD thesis, UC Santa Cruz, 1992.

[12] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques.* Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.

[14] J. Holliday, D. Agrawal, and A. El Abbadi. Database replication using epidemic update. Technical Report TRCS00-01, University of California Santa-Barbara, 19, 2000.

[15] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[16] 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.

[17] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems*, pages 245–254, 1995.

[18] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[19] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333 – 379, 2000.

[20] B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks (DSN2001)*, Göteborg, Sweden, June 2001.

[21] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *International Conference on Distributed Computing Systems*, pages 56–65, 1994.

[22] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of 14th International Symposium on DIStributed Computing (DISC'2000)*, 2000.

[23] F. Pedone. *The Database State Machine and Group Communication Issues.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[24] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[25] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. Technical Report SSC/1999/008, École Polytechnique Fédérale de Lausanne, Switzerland, March 1999.

[26] C. Pu and A. Leff. Replica control in distributed systems: an asynchronous approach. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):377–386, 1991.

[27] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[28] D. Skeen. A quorum-based commit protocol. Berkley Workshop on Distributed Data Management and Computer Networks, February 1982.

[29] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, Amsterdam, The Netherlands, May 1998. IEEE.

[30] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, May 1979.

A Appendix A: Static Replication Algorithm

This appendix contains the full pseudocode for the replication algorithm as well as the description of the variables used in the program.

Data Structure The structure *Action_id* contains two fields: server_id the creating server identifier, and action_index, the index of the action created at that server.

The following local variables reside at each of the replication servers:

- *serverId* - a unique identifier of this server in the servers group.
- *actionIndex* - the index of the next action created at this server. Each created action is stamped with the *actionIndex* after it is incremented.
- *conf* - the current configuration of servers delivered by the group communication layer. Contains the following fields:
  conf_id - identifier of the configuration.
  set - the membership of the current connected servers.
- *attemptIndex* - the index of the last attempt to form a primary component.
- *primComponent* - the last primary component known to this server. It contains the following fields:
  prim_index - the index of the last primary component installed.
  attempt_index- the index of attempt by which the last primary component was installed.
  servers - identifiers of participating servers in the last primary component.
- *State* - the state of the algorithm. One of {RegPrim, TransPrim, ExchangeStates, ExchangeActions, Construct, No, Un, NonPrim}.
- *actionsQueue* - ordered list of all the red, yellow and green actions. White actions can be discarded and, therefore, in a practical implementation, are not in the *actionsQueue*. For the sake of easy proofs this thesis does not extract actions from the *actionsQueue*. Refer to [AAD93] for details concerning message discarding.
- *ongoingQueue* - list of actions generated at the local server. Such actions that were delivered and written to disk can be discarded. This queue protects the server from loosing its own actions due to crashes (power failures).
- *redCut* - array[1..n] - the index of the last action server i has sent and that this server has.
- *greenLines* - array[1..n] - identifier of the last action server i has marked green as far as this server knows. *greenLines*[serverId] represents this server's green line.
- *stateMessages* - a list of State messages delivered for this configuration.
- *vulnerable* - a record used to determine the status of the last installation attempt known to this server. It contains the following fields:
  status - one of {Invalid, Valid}.
  prim_index - index of the last primary component installed before this attempt was made.
  attemp_index - index of this attempt to install a new primary component.

set - array of server_ids trying to install this new primary component.
bits - array of bits, each of {Unset, Set}.

- *yellow* - a record used to determine the yellow actions set. It contains the following fields:
  status - one of {Invalid, Valid}
  set - an ordered set of action identifiers that are marked yellow.

Message Structure Three types of messages are created by the replication server:

- **Action_message** - a regular action message contains the following fields:
  type - type of the message. i.e. Action
  action_id - the identifier of this action.
  green_line - the identifier of the last action marked green at the creating server at the time of creation.
  client - the identifier of the client requesting this action.
  query - the query part of the action.
  update - the update part of the action.

- **State_message** - contains the following fields:
  type - type of the message. i.e. State
  Server_id, Conf_id, Red_cut, Green_line - the corresponding data structures at the creating server.
  Attempt_index, Prim_component, Vulnerable, Yellow - the corresponding data structures at the creating server.

- **CPC_message** - contains the following fields:
  type - type of the message.
  Server_id, Conf_id - the corresponding data structures at the creating server.

Definition of Events Six types of events are handled by the replication server:

- **Action** - an action message was delivered by the group communication layer.
- **Reg_conf** - a regular configuration was delivered by the group communication layer.
- **Trans_conf** - a transitional configuration was delivered by the group communication layer.
- **State_mess** - a state message was delivered by the group communication layer.
- **CPC_mess** - a Create Primary Component message was delivered by the group communication layer.
- **Client_req** - a client request was received from a client.

CodeSegment A.1 Code executed in the NonPrim State

```
case event is

Action: MarkRed( Action )

Reg_conf:
    set conf according to Reg_conf
    Shift_to_exchange_states()

Trans_conf, State_mess: Ignore

Client_req: actionIndex++
    create action and write to ongoingQueue
    ** sync to disk
    generate Action

CPC_mess: Not possible
```

CodeSegment A.2 Code executed in the RegPrim State

```
case event is

Action: MarkGreen( Action )     ( OR-1.1 )
    greenLines[ Action.server_id ] = Action.green_line

Trans_conf: State = TransPrim

Client_req: actionIndex++
    create action and write to ongoingQueue
    ** sync to disk
    generate Action

Reg_conf, State_mess, CPC_mess: Not possible
```

CodeSegment A.3 Code executed in the TransPrim State

```
case event is

Action: MarkYellow( Action )

Reg_conf: set Conf according to Reg_conf
    Vulnerable.status = Invalid
    Yellow.status = Valid
    Shift_to_exchange_states()

Client_req: buffer request

Trans_conf, State_mess, CPC_mess: Not possible
```

CodeSegment A.4 Code executed in the ExchangeStates state

```
case event is

Trans_conf: State = NonPrim

State_mess:
        If (State_mess.conf_id = Conf.conf_id )
        add State_mess to State messages
        if ( all state messages were delivered )
        if ( most updated server ) Retrans()
        Shift_to_Exchange_actions()

Action: MarkRed( Action )

CPC_mess: Ignore

Client_req: buffer request

Reg_conf: Not possible
```

CodeSegment A.5 Code for the Shift_to_exchange_states, Shift_to_exchange_actions, and End_of_retrans Procedures

```
Shift_to_exchange_states()
** sync to disk
clear State_messages
Generate State_mess
State = ExchangeStates
Shift_to_exchange_actions()
State = ExchangeActions
if ( end of retransmission ) End_of_retrans()
End_of_retrans()
Incorporate all green_line from State messages to greenLines
ComputeKnowledge()
if ( IsQuorum() )
attemptIndex++
vulnerable.status = Valid
vulnerable.prim_index = primComponent.prim_index
vulnerable.attempt_index = attemptIndex
vulnerable.set = conf.set
vulnerable.bits = all Unset
** sync to disk
generate CPC message
State = Construct
else
** sync to disk
Handle_buff_requests()
State = NonPrim
```

CodeSegment A.6 Code executed in the ExchangeActions State

```
case event is

Action:
    Mark action according to State messages   ( OR-3 )
    if ( turn to retransmit ) Retrans()
    if ( end of retransmission ) End_of_retrans()

Trans_conf: State = NonPrim

Client_req: buffer request

Reg_conf, State_mess, CPC_mess: Not possible
```

CodeSegment A.7 ComputeKnowledge

```
1.  primComponent = primComponent in all State messages with the maximal (primIndex, attemptIndex)
    updatedGroup = the servers that sent primComponent in their State message
    validGroup = the servers in updatedGroup that sent Valid yellow.status
    attemptIndex = max attemptIndex sent by a server in updatedGroup in their State message

2.  if validGroup is not empty
      yellow.status = Valid
      yellow.set = intersection of yellow.set sent by validGroup
    else
      yellow.status = Invalid

3.  for each server with Valid in vulnerable.status
        if ( serverId not in primComponent.set or
one of its vulnerable.set does not have identical vulnerable.status or vulnerable.prim_index or vulnerable.attempt_index )
then Invalid its vulnerable.status
4.  for each server with Valid in vulnerable.status
        set its vulnerable.bits to union of vulnerable.bits of all servers with Valid in vulnerable.status
        if all bits in its vulnerable.bits are set then its vulnerable.status = Invalid
```

CodeSegment A.8 Code of the IsQuorum and Handle_buff_requests Procedures

```
IsQuorum()
    if there exists a server in conf with vulnerable.status = Valid return False
    if conf does not contain a majority of primComponent.set return False
    return True

Handle_buff_requests()
    for all buffered requests
        actionIndex++
        create action and write to ongoingQueue
    ** sync to disk
    for all buffered requests
        generate Action
    clear buffered requests
```

CodeSegment A.9 Code executed in the Construct state

```
case event is

Trans_conf: State = No

CPC_mess:
    if ( all CPC messages were delivered )
    for each server s in conf.set
    set greenLines[s] to greenLines[ serverId ]
    Install()
    State = RegPrim
    Handle_buff_requests()

Client_req: buffer request

Action, Reg_conf, State_mess: Not possible
```

CodeSegment A.10 Install procedure

```
if ( yellow.status = Valid )
    for all actions in yellow.set
        MarkGreen(Action)            ( OR-1.2 )
yellow.status = Invalid
yellow.set = empty
primComponent.prim_index++
primComponent.attempt_index = attemptIndex
primComponent.servers = vulnerable.set
attemptIndex = 0
for all red actions ordered by Action.action_id
    MarkGreen(Action)                ( OR-2 )
** sync to disk
```

CodeSegment A.11 Code executed in the No state

```
case event is

Reg_conf:
    set Conf according to Reg_conf
    vulnerable.status = Invalid
    Shift_to_exchange_states()

CPC_mess: if ( all CPC messages were delivered ) State = Un

Client_req: buffer request

Action, Trans_conf, State_mess: Not_possible
```

CodeSegment A.12 Code executed in the Un state

```
case event is

Reg_conf:
    set Conf according to Reg_conf
    Shift_to_exchange_states()

Action:
    Install()
    MarkYellow( Action )
    State = TransPrim

Client_req: buffer request

Trans_conf, State_mess, CPC_mess: Not possible
```

CodeSegment A.13 Recover procedure

```
State = Non_prim
for each action in ongoingQueue
  if ( redCut[ serverId ] ¡ Action.action_id.action_index )
    MarkRed( Action )
** sync to disk
```

CodeSegment A.14 Marking procedures

```
MarkRed( Action )

if ( redCut[ Action.server_id ] = Action.action_id.index - 1 )
    redCut[ Action.server_id ]++
    Insert Action at top of actionList
    if ( Action.type = Action ) ApplyRed( Action )
    if ( Action.action_id.server_id = serverId ) delete action from ongoingQueue

MarkYellow( Action )

MarkRed( Action )
yellow.set = yellow.set + Action

Mark_green( Action )

MarkRed( Action )
if ( Action not green )
   place action just on top of the last green action
   greenLines[ serverId ] = Action.action_id
   ApplyGreen( Action )
```