Robust and Efficient Replication Using Group Communication *

Y. Amir, D. Dolev, P. M. Melliar-Smith, L. E. Moser

Hebrew University of Jerusalem
University of California, Santa Barbara

Abstract

We present a new architecture and algorithm for distributed replicated database systems. The replication algorithm operates in the presence of message omission faults, processor crashes and recoveries, and network partitions and re-merges. The architecture exploits a group communication service to minimize communication costs and to eliminate forced disk writes in the critical path, while preserving complete and consistent operation. End-to-end agreement is required only after a change in the membership of the connected servers, rather than on a per action basis.

The updates are globally ordered and, if the system has partitioned, they are applied to the database when they become known to the primary component of the partitioned system. An application may, however, read data and initiate updates at any time, even in a component that is not the primary component. This approach renders replication more efficient and more scalable and, therefore, applicable to many more systems.

1 Introduction

In database systems based on the client-server model, a single server may serve many clients and the heavy load on the server may cause the response time to be adversely affected. Moreover, if the clients are spread over a large geographical area and a high proportion of accesses are queries, the communication cost may be unnecessarily high. In such circumstances, database systems with replicated data and replicated servers

*This work was supported by the United States-Israel Binational Science Foundation, Grant No. 92-00189, by the Advanced Research Project Agency, Grant No. N00174-93-K-0097, and by the National Science Foundation, Grant No. NCR-9016361.

may be able to provide better performance. Each server serves fewer clients resulting in shorter response times, and communication costs are reduced because the servers are local to the clients.

In a single-server database system, there is no protection against failures. If the server fails, then all processing must stop. Replication improves availability of information when a processor crashes or the network partitions. In the extreme, when very high availability must be provided in a system that is subject to partitions such as a mobile system, each client could have its own server.

Existing replication methods are often needlessly expensive. They may use point-to-point communication when broadcast or multicast communication is available. They typically pay the full price of end-to-end acknowledgment for all of the participants for every update, or even of several rounds of end-to-end acknowledgments. They may claim locks and, therefore, may be vulnerable to faults that can unnecessarily block the system for long periods of time.

Most replicated database designs are based on two-phase commit protocols [10, 13]. A two-phase commit protocol can provide higher performance than a three-phase commit protocol [25], but incurs the risk of blocking. Even with this compromise, however, the performance of two-phase commit protocols is not good. Consequently, replicated databases are used mostly to improve availability, and the number of replicas is limited to two or three. Replication is seldom used to boost performance.

In this paper we present a new architecture and algorithm for object replication with a read-one write-all strategy. The algorithm operates on top of a group communication service, and uses low-level acknowledgments on a per action basis. In regular operation, the approach avoids end-to-end acknowledgments, and eliminates the need for forced disk writes before sending acknowledgments, while ensuring global consistency even after failures. End-to-end acknowledgments and forced disk writes are required only after a membership change, thus amortizing the cost of these expensive operations over many transactions.

This strategy derives from a new way of looking at group communication protocols. Past group communication protocols focused on totally ordering messages at the group communication level. That service, although useful for many applications, is not enough to guarantee complete consistency at the application level without additional end-to-end acknowledgments, as has been noted by Cheriton and Skeen [8]. We have focused on providing an additional level of knowledge within the group communication protocol, which allows us to reach agreement on a consistent transition from one configuration to another with minimal overhead. We show how our group communication techniques for Transis and Totem [2, 3, 18] can be utilized for database replication with high efficiency and strictly serializable semantics.

2 The Model

2.1 The Environment

A *replication service* maintains an object in a distributed system. This object is replicated to improve performance and availability. The replication service is provided by a known finite set of processes, called the *server group*. The individual processes within the server group are called *servers*, each of which has a unique identifier. Typically, each server within a server group runs on a different processor. Processes to which the service is provided are called *clients*.

2.2 The Fault Model

The system is subject to message omission, server crashes and network partitions. We assume no message corruption and no malicious faults.

A server may crash and may subsequently recover after an arbitrary amount of time with its stable storage intact. When a server recovers, it retains its old identifier.

The network may partition into a finite number of *components*. The servers in a component can receive messages multicast by other servers in the same component, but servers in two different components are unable to communicate with each other. Two or more components may subsequently merge to form a larger component.

Message omissions are handled at the group communication layer. The replication layer handles server crashes and network partitions using the configuration notification provided by the group communication service.

2.3 Extended Virtual Synchrony

Processor failures and network partitions cannot be predicted and, therefore, the last message received by a server before a failure or disconnection cannot be determined. To overcome this problem, we have introduced the notion of extended virtual synchrony [3, 20]. The Totem and Transis systems, and more recently also the Horus system [28], implement group communication services that meet the requirements of extended virtual synchrony.

The group communication layer determines *configurations* as sets of servers such that every server within the set can communicate with every other server within the set. In a *regular configuration*, new messages are originated and delivered in the total order. In a *transitional configuration*, no new messages are transmitted but messages originated in the previous regular configuration are delivered.

A *configuration change message* is generated by the group communication layer to notify the members of a new or changed configuration about the membership of that configuration. The configuration change message contains the set of identifiers of the members of the configuration and an identifier for that configuration.

The *safe* message delivery service of extended virtual synchrony guarantees that all messages delivered to any process that is a member of a configuration are delivered to every process that is a member of the configuration, unless that process fails. The messages that are delivered in a transitional configuration could not be delivered in the preceding regular configuration because the requirements for safe delivery were not met. The configuration change message that initiates a transitional configuration defines the reduced membership within which it is possible to guarantee safe delivery of all available undelivered messages of the previous regular configuration.

Safe delivery provides additional level of knowledge compared with totally ordered message delivery (ABCAST in Isis and Horus, and Agreed delivery in Transis and Totem). This additional knowledge is utilized to eliminate end-to-end acknowledgments on a per action basis between the servers.

2.4 The Service Model

An *object* is a collection of data with no dependencies on the values of other objects. A file, a file system, and a collection of tables in a relational database can all be considered as objects. An *action* defines a transition from the current state of an object to the next state; the next state is completely determined by the current state and the action. Each action (request) contains an *update* part and a *query* part, either of which can be omitted. The update part of an action defines a modification to be made to the object, and the query part returns a value to the client. Examples of actions include updating a table in a database and reading a record from a database.

The effects of an action are encapsulated within a single object; concurrency control guarantees atomicity of the effects of the action within the object. The replication service gives the appearance of a single object, but is implemented by multiple servers in the server group. Each server maintains an individual instance of the object, and stores its instance in a local stable *object store*. The initial state of the object is identical at all of the servers.

We introduce the following notation:

- $a_{p,i}$ is the ith action performed by server p
- fail_p is a predicate that denotes the failure of server p
- $\mathrm{connect}(p, q)$ is a predicate that denotes the existence of a continuous connection between servers p and q
- $\mathrm{eventual\_path}(p, i, q)$ is a predicate that denotes the existence of an eventual path from server p to server q by which all of the actions known to p up to and including action $a_{p,i}$ are communicated to q.

An *eventual path* from server p to server q up to and including action $a_{p,i}$ is a communication path from p to q such that there exist pairs of servers along the path and intervals during which they are connected so that the message containing $a_{p,i}$, and all prior messages received by p, are eventually received by q. It does not require a continuous or direct connection between p and q.

The correctness criteria for the replication servers are defined in terms of a global total order of actions that satisfies the following properties:

- **Consistency.** If server p performs the ith action and server q performs the ith action, then those actions are identical.

  $\exists a_{p,i}, a_{q,i} \Rightarrow a_{p,i} = a_{q,i}$

- **Completeness.** If server p performs the ith action, then it has already performed all previous actions.

  $\exists a_{p,i} \Rightarrow \forall j, 1 \leq j < i, \exists a_{p,j}$

- **Replication.** If at some point in time server p performs an action and from that point in time there is a continuous connection between p and q, then server q eventually performs the action.

  $\Diamond(\exists a_{p,i} \wedge \Box \text{connect}(p,q)) \Rightarrow \Diamond \exists a_{q,i}$

- **Liveness.** If at some point in time server p performs the ith action $a_{p,i}$ and there exists an eventual path from p to q up to and including $a_{p,i}$, then eventually server q performs the ith action.

  $\Diamond(\exists a_{p,i} \wedge \Diamond \text{eventual\_path}(p,i,q)) \Rightarrow \Diamond \exists a_{q,i}$

3 The Architecture

The system architecture is structured into a group communication layer, a replication layer and an application layer, as shown in Figure 1. The application may be further structured as a set of clients that initiate actions, and a set of servers that implement those actions on an object.

The architecture is intended to allow the object to be replicated without risk of inconsistency. Replication of the object also requires replication within the replication layer, and a replication server implements the replication layer for each replica of the object.

We describe here the management of a single object that is fully replicated among the servers, and a single process group of which all servers are members. This can be generalized to management of several independent objects, each of which is replicated on its own set of servers and comprises a unique process group.

In a typical operation, the application requests an action from the replication layer. The replication layer generates a message containing this action and passes it on to the group communication layer, which multicasts the message as one or more packets. At each replica, the packets are recombined into the message and delivered

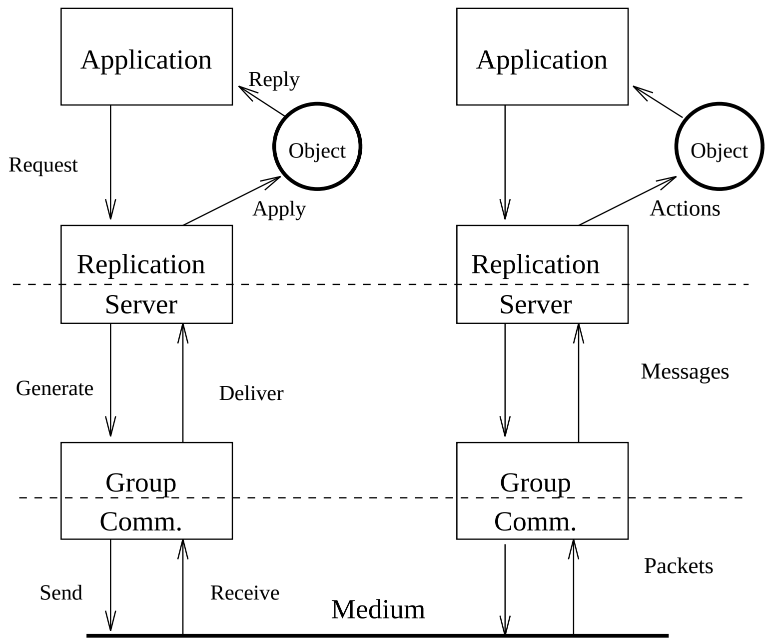

Figure 1: *The architecture.*

to the replication layer. The replication layer delivers the message, in a global total order, to the application which applies the corresponding action to the object. Some actions (updates) can change the state of the object, others (queries) can cause a reply to be sent to the application, and yet others are a combination of updates and queries. By delivering the same set of messages in the same order, the algorithm guarantees that each of the servers applies the same set of actions to its local object in the same order.

In this presentation, we assume the existence of a group communication layer that provides multicast and group membership services. More specifically, it provides the property of extended virtual synchrony, which ensures safe delivery of messages even in the presence of process crashes and recoveries, and of network partitions and remerges. This approach eliminates the need for separate agreements on the claiming of locks, since such agreements can be derived from the ordering of messages. Group communication can exploit hardware broadcasting or multicasting when it is available. Reliable ordered delivery protocols that use hardware broadcasts and multicasts [2, 3, 18, 19] can achieve higher throughput and lower latency than protocols based on point-to-point communication.

If the system partitions into several components, the replication layer identifies at most one component as the primary component. The replication servers in a primary component determine the global total order of actions according to the order provided by the group communication layer. As the actions are ordered, the servers pass them to the application layer, where they are applied to the local replica of the object. In the primary component new actions can be ordered and applied to the object immediately upon delivery by the group communication layer. In non-primary components, actions must be delayed until communication is restored and the servers learn of the order determined by the primary component.

On notification of a membership change by the group communication layer, the replication servers exchange messages containing actions multicast before the membership change. As in [1], this exchange of information ensures that every action known to a member of the primary component is applied to the object. Therefore, each member of the primary component applies the same sequence of actions in the same order, thus guaranteeing a consistent object state.

Continued operation of the primary component is ensured, except in the rare scenarios in which multiple servers in the primary component crash or become disconnected within a window of time so short that the membership algorithm could not be completed and none of the servers is certain about which messages were ordered within that primary component. If even one server possesses this knowledge, it can provide the other servers with the necessary information to allow the formation of the next primary component. If no server has this knowledge, then recovery of, and communication with, every member of the previous primary component is required before the next primary component can be formed.

4 The Services of the Replication Layer

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 of a server group is reflected in the delivery of a configuration change message by the group communication layer to each server in that component.

The replication servers implement a symmetric distributed algorithm to determine the order of actions to be applied to the object. 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:

- **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 and which, therefore, can be applied to the object.

- **White Action:** An action for which the server knows that all of the servers have already marked the action green. Thus, the server can discard a white action because no other server will need the action subsequently.

All of the white actions precede the red and green actions in the total order and define the white zone. All of the green actions precede the red actions in the total order and define the green zone. Similarly, the red actions define the red zone. An action can be marked at different servers with different colors. However, no action can be marked white at one server, when it is marked red or does not exist at another server.

In the primary component, actions are marked green on delivery by the group communication layer and are applied to the object immediately. In a non-primary component, actions are marked red. Since the order of such actions cannot be determined immediately and since we require strong consistency, these actions cannot be applied to the object yet. In the real world, however, where incomplete knowledge is inevitable, many applications would rather have an immediate answer, rather than incur a long latency to obtain a complete and consistent answer.

We, therefore, provide an additional service to the clients in a non-primary component. In a non-primary component, two values of the object are maintained: a *dirty* version and a *clean* version. Updates are applied only to the dirty version of the object. The clean version of the object remains unchanged.

The query part of an action defines a result to be returned to the client. It has a qualifier with three possible values:

- **Consistent_Query.** The result of this query is returned to the client as soon as this action is ordered as a green action. In the primary component, the response is typically immediate but, in a non-primary component, the client will be blocked. The consistent query returns a result based on the state of the object after all previous actions in the global total order have already been applied.

- **Weak_Query.** The result of a weak query is obtained from the clean version of the object. In a primary component, the result of a weak query reflects all prior updates. In a non-primary component, the result of this query is derived from the most recent globally consistent version of the object available to the server, and red updates not yet ordered are not reflected in the result.

  The weak query is based on the consistent local object state, which reflects all of the white and green actions. The weak query yields results derived from a state that was consistent but is now possibly obsolete. In particular, the application can initiate updates and then issue a weak query to find that the updates are not reflected in the result of the query.

- **Dirty_Query.** The result of a dirty query is derived from the most recent version of the object, reflecting all prior updates including those that are not yet globally ordered. In a non-primary component, the result of a dirty query is obtained from the dirty version of the object.

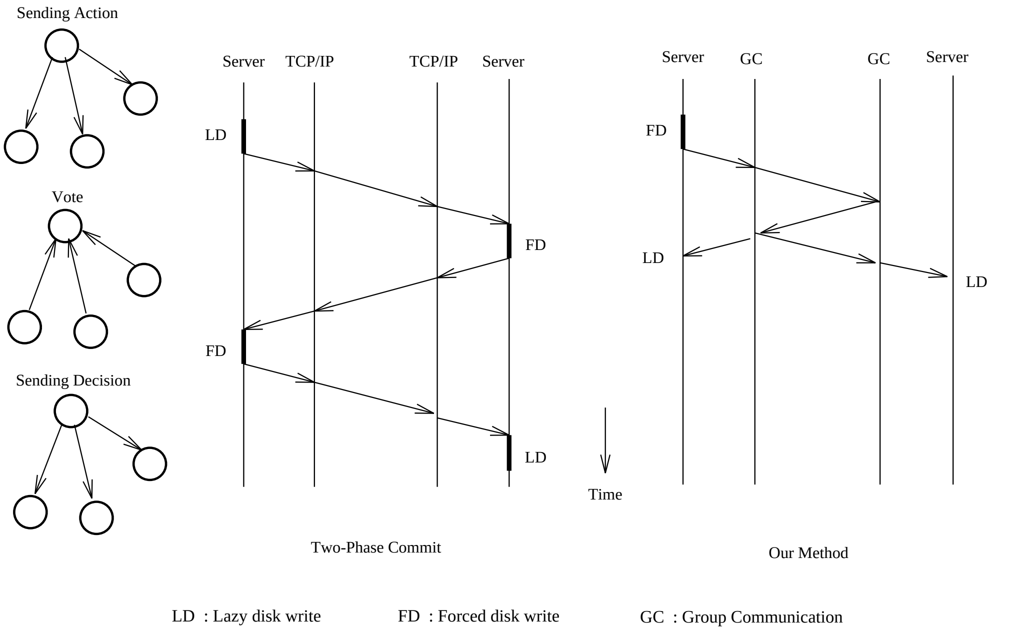

Figure 2: *A time-line comparison.*

The dirty query is based on a temporary object state, which reflects all of the white, green and red actions known to the server. An application that uses a dirty query must know that the result is tentative and based on a state of the object that may never exist. When the server is able to rejoin the primary component, the action will be re-applied to the consistent state of the object reflecting all prior actions, and the result of the dirty query may be different. The semantics of a dirty query are those described in [14].

In a primary component the results of the consistent query, weak query and dirty query are identical.

In a primary component the latency of actions is determined by the safe delivery latency of the group communication layer (see [21] for an analysis of the latency of the Totem protocol) plus the processing time within the replication layer. The same is true for actions that contain a weak query part in a non-primary component. The latency of actions that contain a dirty query part in a non-primary component are implementation dependent.

In contrast, the latency for existing database systems is determined by end-to-end acknowledgment mechanisms, such as two-phase commit, and by forced disk writes. Our approach can, therefore, provide lower latency than existing database systems, while preserving strong consistency for updates and queries.

A comparison of the time-lines of the two approaches is given in Figure 2. The time line for two-phase commit is taken from [14].

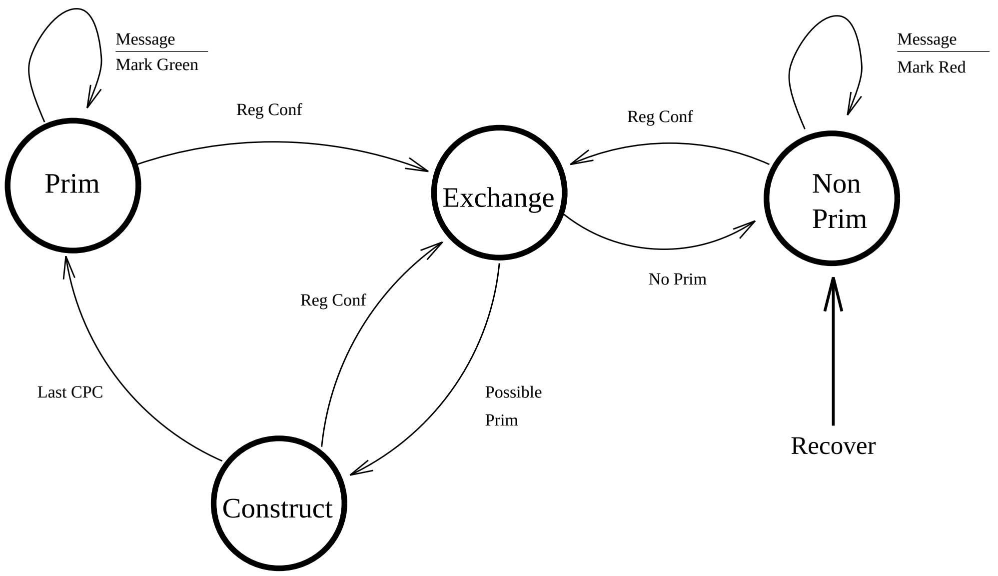

Figure 3: *High-level state machine.*

5 The Algorithm

5.1 High-Level Description

We now present a high-level description of the algorithm in the form of a finite state machine with four states, as shown in Figure 3.

- **Prim.** The server currently belongs to the primary component. When a message containing an action is delivered by the group communication layer, the action is immediately marked green and is applied to the object.

- **Non_Prim.** The server belongs to a non-primary component. When a message containing an action is delivered by the group communication layer, the action is marked red.

- **Exchange.** The server shifts to this state when a new (regular) configuration is formed. All of the servers belonging to the new configuration exchange information that allows them to define the set of actions that are known to some, but not all, of them. After all of these actions have been exchanged and the green actions have been applied to the object, the servers check whether this configuration can become the primary component. If so, they shift to the Construct state; otherwise, they shift to the Non_Prim state and form a non-primary component. We use dynamic linear voting [15] to determine the primary component. This check is done locally at each server without the need for additional exchange of messages between the servers.

- **Construct.** In this state, all of the servers in the component have identical knowledge about the configurations. After writing the data to stable storage, each of the other servers multicasts a Create Primary Component (CPC) message. On receiving a CPC message from each of the other servers, a server shifts to the Prim state. If a configuration change occurs before it has received all of the CPC messages, the server returns to the Exchange state.

When a membership change occurs, the connected servers exchange information and try to reach a common state. If another membership change occurs before the servers establish that common state, they try again. When they reach a common state, that state will be a Prim state or a Non_Prim state. The exchange of messages in the Exchange state is the mechanism by which the eventual path property is achieved.

When the servers within a component are in the Prim state, they do not acknowledge messages. As long as no membership change occurs, all of the connected servers receive the same set of messages in the same order, and there is no need for end-to-end acknowledgments. The same property holds for the Non_Prim state.

5.2 Refinement

Due to the asynchronous nature of the system model, we cannot reach common knowledge about which messages were received by which processes. Instead, we rely on the semantics of extended virtual synchrony for safe delivery, particularly when a smaller transitional configuration must be introduced to deliver a message as safe.

During a membership change, the servers must decide whether the new primary component was installed or was not installed. There must be no possibility that one server decides that the component has been installed while another server decides that it has not. Extended virtual synchrony allows each server three possible views of the situation:

1. At least one server received all of the CPC messages and installed the new primary component.

2. It is unknown whether any server received all of the CPC messages and installed the new primary component.

3. At least one server did not receive all of the CPC messages and did not install the new primary component.

The semantics of extended virtual synchrony ensure that, if a server p receives all of the CPC messages as safe in a regular configuration, then all of the servers in that configuration will receive all of the CPC messages before the next regular configuration unless they crash; they might, however, receive some of the CPC messages in

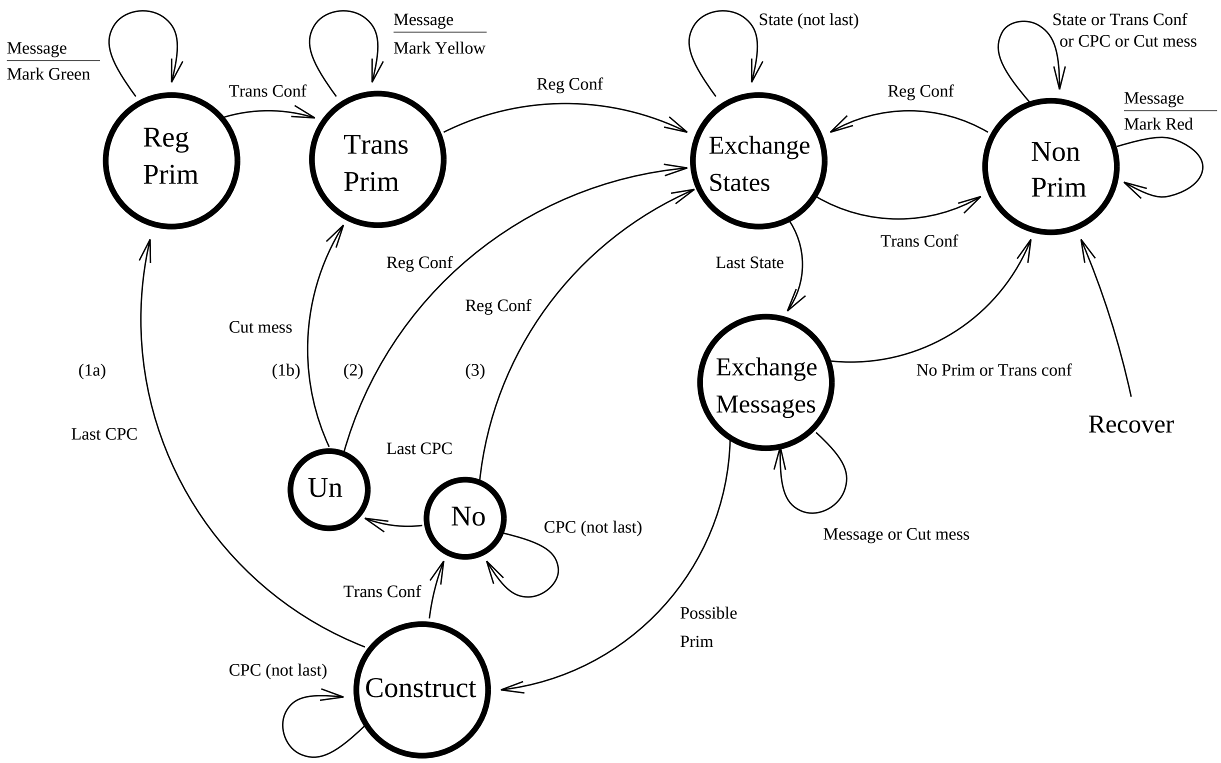

Figure 4: *More detailed state machine.*

a transitional configuration. Thus, p knows that it is in case 1 and that case 3 is excluded for all of the servers.

Likewise, if a server q receives a configuration change for a regular configuration before it has received all of the CPC messages, then no server can have received a message as safe in the previous configuration that q did not receive. In particular, no server received all of the CPC messages as safe in the previous configuration. Thus q knows that it is in case 3 and that case 1 is precluded for all of the servers.

If a server r received all of the CPC messages, but received one or more of those messages as safe in the transitional configuration, then r cannot know whether there is a server p that received all of the CPC messages as safe in a regular configuration, or whether there is a server q that did not receive some of the CPC messages at all. Thus, server r is in case 2 and does not know whether some server is in case 1 or in case 3. It does, however, know that case 1 and case 3 cannot occur together.

Based on the above observations, we refined the algorithm. We split the Prim state into the Reg_Prim and Trans_Prim states, and we added the No state (No server has yet installed this component) and Un state (Unknown whether any server has installed this component) as refinements of the Construct state. We mark as yellow those actions that were delivered in a transitional configuration of a primary component. This action could have been marked as green by another member of a primary component that partitioned. A yellow action becomes green at a server as soon as this server learns that another server marked it green, or with the installation of the next primary component.

- **Yellow Action:** An action that was received in a transitional configuration of a primary component.

We use a single message (the Cut message) to determine a new primary component. The leader of a server group is the server with the lowest identifier in the group. The leader of a new primary component group broadcasts the Cut message to the group when it has received as safe a CPC message from every server in the group. The Cut message initiates transmission of messages containing actions within the new primary component. The refined algorithm considers four cases shown in Figure 4.

In case 1a, server p receives all of the CPC messages in the regular configuration and installs the next primary component. It knows that there cannot be a server q in the configuration that received the configuration change message for the next regular configuration without receiving all of the CPC messages (this is because p received them all as safe messages in the regular configuration). This corresponds to the transition from the Construct state to the Reg_Prim state in Figure 4. Yellow messages are converted to green and applied to the object. Transmission of the Cut message by the leader and its reception by other group members initiates transmission of messages containing actions within the new primary component. Red messages are converted to green and applied to the object upon marking the corresponding Cut message as green.

In case 1b, server r receives all of the CPC messages, but some are delivered in the transitional configuration that precedes configuration c. Furthermore, r receives a Cut message in the transitional configuration, while it is in the Un state. Thus, r knows that some server p installed the new primary component (the leader that sent the Cut message) and r installs the new primary component too. The yellow messages are converted to green and applied to the object. Server r must, however, transition to the Trans_Prim state, because it has already received a transitional configuration change message for that primary component. The Cut message is marked as yellow.

In case 2, server r receives all of the CPC messages, some in the transitional configuration, but r receives no Cut message before the configuration change message for the new regular configuration. Server r cannot be certain whether any other server has installed the primary component, and must allow for that possibility. This corresponds to the transition from the Un state to Exchange_States. The previous primary component becomes unknown, yellow messages remain yellow, and red messages remain red. The server must remain uncertain as to whether this primary component has been installed, and can install no subsequent primary component, until it has received information from a server in case 1 or in case 3, or until it determines that every server was in case 2 where we opt not to install that primary component. Case

2 is critical for avoiding the need for common knowledge, and we know of no prior algorithm that considers this state of incomplete knowledge.

In case 3, server r receives the next regular configuration change message without receiving all of the required CPC messages. This corresponds to the transition from the No state to Exchange_States. Server r knows that no server could have installed the new primary component. The previous primary component remains the previous primary component and the state of the yellow messages remains unchanged.

5.3 Selection of a Primary Component

In a system that is subject to partitioning, we must ensure that two different components do not apply contradictory actions to the object. Therefore, we need a mechanism for selecting a primary component that can continue to apply actions to the object (the clean version of the object). Several techniques have been described in the literature [11, 15, 27].

- **Monarchy.** The component that contains a designated server becomes the primary component.
- **Majority.** The component that contains a (weighted) majority of the servers becomes the primary component.
- **Dynamic Linear Voting.** The component that contains a (weighted) majority of the last primary component becomes the primary component.

Dynamic linear voting is generally accepted as the best technique, when certain reasonable conditions hold [22]. The choice of the weights and adapting them over time is beyond the scope of this paper. We employ dynamic linear voting.

Any system that employs (weighted) dynamic linear voting can use (weighted) majority, since majority is a special case of dynamic linear voting. Monarchy is a special case of weighted majority (when all servers except the master have weight zero). However, it is not always easy to adapt systems that work well for monarchy or majority to dynamic linear voting.

6 Related Work

Much work has been done in the area of distributed replicated databases and in the area of group communicaiton. Two-phase-commit protocols [10, 13] are the main tools for providing a consistent view in a distributed replicated database system over an unreliable network. These protocols impose a substantial additional communication cost on each transaction but, despite this substantial cost, if the transaction coordinator fails, other processors may be unable to commit the last transaction and the entire system may be blocked until every failed processor has recovered. Three-phase-commit protocols [25] try to overcome some of the availability problems of two-phase-commit protocols, paying the price of an additional communication round and, therefore, of additional latency.

Protocols have been developed to optimize specific cases: Some protocols limit the transactional model to commutative transactions [24]. Others give special weight to a specific processor or to a specific transaction [26]. Explicit use of timestamps enables other protocols [4] to avoid the need to claim locks or to globally totally order actions. Other applications settle for relaxed consistency criteria [12]. To reduce some of the communication costs, [9] maintains an approximate view of the server membership.

In a fully replicated database, achieving a quorum is enough to commit a transaction. The basic quorum scheme uses majority voting [27] or weighted majority voting [11]. Dynamic linear voting [15] is a more advanced approach that defines the quorum adaptively. This approach outperforms the static techniques as shown by [22]. When a partition occurs, if a majority of the quorum can communicate among themselves, a new and smaller quorum is established within which updates can be performed.

One of the leading systems in the area of group communication is the Isis system [6], which provides totally ordered message delivery. Isis also introduced the concept of virtual synchrony [5] for systems that are subject to fail-stop processor faults. Virtual synchrony ensures that two processors that transition together from one view (configuration) to another should see that view change at the same point in the message sequence and should deliver exactly the same set of messages in the first view. Extended virtual synchrony [3, 20] extends the concept of virtual synchrony to systems in which the network can partition and remerge and to systems in which failed processors can be repaired and recover with stable storage intact.

The Trans and Total protocols [19], the Lansis and Toto protocols [2], and the Totem protocol [3, 18] for reliable totally ordered delivery all utilize the broadcast capability of the communication network. These protocols avoid the need to send separate messages and acknowledgments for each of the destinations. Similar approaches can be found in Psync [23], in Horus [28], in Amoeba [16], in Delta-4 [29], and in the Chang and Maxemchuk protocol [7]. These systems and protocols make better use of the communication medium, particularly where data replication is required in an environment built as a collection of several local-area networks.

Prior research [1] described an architecture that uses group communication to achieve consistent replication. The overall architecture and the liveness requirement of this proposal are taken from there. The serious inefficiency of [1] is the method of global total ordering, which uses a Lamport clock and requires an eventual path from every server to apply a new action. This technique is highly efficient if causal order is sufficient and no faults occur, but has poor fault tolerance. Keidar [17] uses the framework of [1] but replaces the requirement of an eventual path from every server by the requirement that a majority of the servers are connected. As far as we know, that method is the only method of providing replication that always allows a connected majority of the servers to make progress. Both [1] and [17] use the flow control and multicast properties of group communication, but both still need an end-to-end acknowledgment on a per action basis, diminishing the performance advantages of multicasting.

7 Conclusion

Replicated database systems can provide high performance, as well as high availability, by utilizing advanced group communication protocols. The architecture and algorithm presented here minimize communication costs and latency because they eliminate end-to-end acknowledgments and forced disk writes in the critical path. End-to-end agreement between the replicated servers is required only after a change in the membership of the connected servers, rather than on a per action basis. Furthermore, the use of the eventual path technique to propagate knowledge between network components is superior to the use of a continuous and direct connection. Currently, we are implementing the architecture and algorithm, described here, in the Transis system and in the Totem system.

References

[1] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. In *Proceedings of the Workshop on Hardware and Software Architectures for Fault Tolerance, Lecture Notes in Computer Science 774*, June 1993.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

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

[5] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Annual ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[6] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transaction on Computer Systems*, 5(1):47–76, February 1987.

[7] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[8] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1993.

[9] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 240–251, March 1986.

[10] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[11] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[12] R. A. Golding. *Weak Consistency Group Communication and Membership*. PhD thesis, Computer and Information Sciences Board, University of California, Santa Cruz, 1992.

[13] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60*, pages 393–481. Springer-Verlag, 1978.

[14] J. Gray and A. Reuter. Transaction processing: Concepts and techniques. Morgan Kaufman Publishers, 1993.

[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, June 1990.

[16] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[17] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.

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

[19] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[20] L. E. Moser, Y. Amir, P. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, June 1994.

[21] L. E. Moser and P. M. Melliar-Smith. Probabilistic bounds on message delivery for the Totem single-ring protocol. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994.

[22] J. F. Paris and D. D. E. Long. Efficient dynamic voting algorithms. In *Proceedings of the 4th International Conference on Data Engineering*, pages 268–275, February 1988.

[23] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[24] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data*, May 1991.

[25] D. Skeen. A quorum-based commit protocol. In *Berkeley Workshop on Distributed Data Management and Computer Network*, number 6, pages 69–80, February 1982.

[26] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, 3(3):188–194, May 1979.

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

[28] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Department of Computer Science, August 1994.

[29] P. Verissimo, L. Rodrigues, and J. Rufino. The Atomic Multicast Protocol (AMp). In D. Powell, editor, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, pages 267–294. Springer-Verlag, 1991.