Total Order Broadcast for Fault Tolerant Exascale Systems

Dan Schatzberg
Department of Computer Science, Boston University
111 Cummington Street
Boston, Massachusetts
dschatz@bu.edu

James Cadden
Department of Computer Science, Boston University
111 Cummington Street
Boston, Massachusetts
jmcadden@bu.edu

Orran Krieger
Department of Computer Science, Boston University
111 Cummington Street
Boston, Massachusetts
okrieg@bu.edu

Jonathan Appavoo
Department of Computer Science, Boston University
111 Cummington Street
Boston, Massachusetts
jappavoo@bu.edu

ABSTRACT

In the process of designing a new fault tolerant run-time for future exascale systems, we discovered that a total order broadcast would be necessary. That is, nodes of a supercomputer should be able to broadcast messages to other nodes even in the face of failures. All messages should be seen in the same order at all nodes.

While this is a well studied problem in distributed systems, few researchers have looked at how to perform total order broadcasts at large scales for data availability. Our experience implementing a published total order broadcast algorithm showed poor scalability at tens of nodes. In this paper we present a novel algorithm for total order broadcast which scales logarithmically in the number of processes and is not delayed by most process failures.

While we are motivated by the needs of our run-time we believe this primitive is of general applicability. Total order broadcasts are used often in datacenter environments and as HPC developers begins to address fault tolerance at the application level we believe they will need similar primitives.

1. INTRODUCTION

Researchers have advocated for algorithm-based fault tolerance [5] to improve HPC software. We have begun the design and implementation of an HPC run-time, *EbbRT* [2], which allows for developers to encapsulate and express fault tolerance and communication of individual software components. In particular, developers write software in an object model called *Elastic Building Blocks(Ebbs)*. Each instance of an Ebb can have a unique and dynamic distributed internal structure of per-processor *representatives* that are encapsulated behind it's method-based interface. The Ebb developer specifies when and how a representative should be created and behave to satisfy the Ebb's methods. This may include replicating and partitioning data to other representatives. Individual methods may operate on the data of a single per-processor representative of an Ebb instance or the data of any sub-set of the representatives by using communications primitives in the methods' implementations. The runtime provides mechanisms for locating and managing the representatives of an Ebb. The client of an Ebb does not have any knowledge of how the Ebb is implemented, only the interface to which it makes calls. A client is both unaware of the internal distributed structure or dynamic changes to it such as growth, shrinkage, or complete replacement. We have explored this object model in the context of shared memory multi-processors and found it to be an effective way to encapsulate communication oriented optimizations and dynamic adaptation [1, 15].

A central feature of the runtime is provisioning and management of an application wide consistent namespace of Ebb identifiers (EbbIds). An EbbId can be bound and unbound from an Ebb instance and all calls to an Ebb are directed through an EbbId that is bound to the instance. Scalable and dynamic behavior is facilitated through a lazy and on-demand resolution of an EbbId to an Ebb instance. Ebbs are invoked like normal objects using a common typed EbbId on all processors, however, to ensure scalability and dynamic behavior we adopt a *miss* model. When an invocation of a method of an Ebb instance is first made on a processor, through an EbbId, an Ebb specific miss function that is bound to the EbbId is invoked. In this function the Ebb programmer can specify what actions should be taken – eg. construction, initialization and binding of a new representative for this processor to the EbbId; or redirection of the call to an existing representative; or even the construction of a completely new Ebb instance to which the EbbId should be rebound too such that all future calls on all processors will target this new instance. The miss approach is core to how dynamic and reactive Ebb behavior can be achieved. The consistency, scalability and efficiency of the EbbId namespace operations, specifically the translation of an EbbId to a miss function and updates to this binding, are of important concern.

In our previous work, miss function bindings were stored

in hardware provided consistent globally shared memory and so every processor could locate the miss function associated with a particular ID when a miss occurred and updates via binds and unbinds could be well ordered. In our current work, however, given our supercomputer target, we cannot rely on global shared memory across all processors in the system and we must contend with processor/node failure. The component of our system responsible for locating the miss behavior currently bound to given Id therefore requires a new design. The functionality we require can in fact be mapped directly to a key-value store. When a miss function is bound to an EbbId, it can be viewed as a PUT operation on a key in a key-value store. Equivalently, when a miss occurs, locating the miss function for the given EbbId is a simple GET operation on the key. While a given Ebb may, for high performance, implement weak consistency of its internal data across its representatives there is value in having the EbbId namespace itself be strongly consistent. Doing so can help ensure that the Ebb developer can rely on simple global semantics for the ordering of bind and unbind operations on EbbIds.

In order to achieve an efficient implementation of the EbbId namespace it is worth considering in its design how EbbId will be operated on. We expect that misses occur much more frequently than bind operations do. In fact we would like misses to be able to be serviced locally on each processor so they do not have to block the executing thread to fetch remote data. If every miss blocks, then potentially any Ebb invocation could block which can be challenging for a programmer to cope with. We, therefore, have designed our system so that all miss function updates (binds and unbinds) via a PUT, are sent to all nodes in the system so that lookups via a GET on an EbbId can be performed locally. Given our goal for a simple consistent view of the namespace, updates to the miss function via a PUT to an EbbId should be seen in the same order on all processors. In distributed systems literature, this primitive is referred to as a total order broadcast [10].

While total order broadcast is a well studied primitive for building fault tolerant systems, few researchers have looked at how to perform total order broadcasts at large scales for data availability. Our experience implementing a published total order broadcast algorithm showed poor scalability with tens of nodes. We require better performance to meet the needs of our system. In this paper we present a novel algorithm for total order broadcast which scales logarithmically in the number of processes and is not delayed by most process failures. While we are motivated by the needs of our run-time and the EbbId namespace, we believe this primitive is of general applicability.

This paper is structured as follows..

2. RELATED WORK

There exists a considerable amount 9of prior work on constructing total order broadcast algorithms. Similarly, a closely related primitive, consensus, has been well studied. Both of these primitives are used commonly to allow a set of servers to behave together as a single replicated state machine [18]. All servers run a copy of the same deterministic state machine logic and all messages are received at all servers in the same order. This allows for a number of servers to fail without causing the entire state machine to fail and thus provides some degree of fault tolerance.

Paxos [16] is the most common consensus algorithm and been implemented by Google for use in their Chubby [7] distributed lock service as well as other services [3, 9]. Microsoft has also implemented Paxos for use in their Autopilot automatic data center management infrastructure [14].

Yahoo's Zookeeper [11] exports an API which allows clients to manipulate a hierarchical namespace. To guarantee that writes to the namespace satisfy linearizability, the authors designed and implemented a total order broadcast called Zab [17].

Most implementations are designed to scale only to the number of servers necessary to tolerate failures. A handful of servers is generally sufficient for this purpose. Clients requiring service must make remote requests to the relatively small number of servers in the replicated state machine. As such most uses of these primitives are to store infrequently modified data such as what services are available at which servers, work queues or synchronization primitives such as global locks. In contrast, our demands are for a primitive that allows for all machines in the system to participate in the total order broadcast and therefore we depend on an algorithm with good scalability.

Recently, the MPI Fault Tolerance Working Group has proposed a number of modifications to the MPI specification in order to allow for applications to tolerate failures. One such proposal required the construction of a primitive to allow all processes to come to consensus on the set of failed processes [13] and an efficient and scalable implementation was published [6]. While it has been shown that total order broadcast can be easily achieved with consensus [8], we have adopted to explore an algorithm that provides a stronger ordering, namely primary order [17], not achievable with a consensus algorithm. Primary order is more naturally suited to implementing namespaces such as our EbbId namespace.

3. SYSTEM MODEL

In this section, we formally describe the system model and assumptions upon which we design our total order broadcast algorithm. We use terminology from distributed systems literature and earlier total order broadcast descriptions. A brief summary of this terminology is included below.

Our system contains an ordered set of processes $\Pi = p_1, p_2, ..., p_n$ processes. We refer to a process's position in the set as its rank. Note that while we use the term *process*, one could equivalently say *server* or *node*. Processes may fail and do not recover. A process that has not failed is called a *correct* process. Each correct process can send messages to every other correct process; no network partitions can occur. Messages between each pair are delivered in the order they were sent and cannot be lost or corrupted.

Each process is equipped with a *perfect failure detector* [8]. The failure detector reports *suspected* process failures with the following properties:

Strong completeness: Every failed process is eventually suspected by every correct process.

Strong accuracy: No process is suspected before it crashes.

The construction of reliable networks and perfect failure detectors is outside of the scope of this paper. However, we are not alone in believing this system model is reasonable for fault tolerant supercomputers. These assumptions are

equivalent to the assumptions made in proposals for updating the MPI specification to address fault tolerance [6, 4].

In our algorithm, one process will be elected as the *primary*, ρ. The primary is the only process allowed to issue a total order broadcast. We say that a total order broadcast causes other processes to *deliver* a message. The act of delivering a message makes it available to application-level software. When a primary fails, a new primary will be elected. At most only one primary is active at a time. Over time there is a sequence of primaries: $\rho_1\rho_2...\rho_e\rho_{e+1}...$, where $\rho_e \in \Pi$. We say a primary ρ_e precedes a primary $\rho_{e'}$, $\rho_e \prec \rho_{e'}$, if $e < e'$.

Formally, our total order broadcast has the following properties [17]:

Integrity: If some process delivers a message m, then some process has broadcast m. This property prevents messages from being erroneously delivered.

Total Order If some process p_i has delivered m before m' then if a process p_j delivers m', then it must have delivered m before m'.

Agreement: If some process p_i delivers m and some process p_j delivers m', then either p_i delivers m' or p_j delivers m. This process prevents disjoint groups of message deliveries.

Additionally, our algorithm ensures *primary order* delivery expressed as follows:

Local primary order If a primary broadcasts m before it broadcasts m', then a process that delivers m' must have delivered m before m'

Global primary order If a primary ρ_i broadcasts m and a primary $\rho_j, \rho_i \prec \rho_j$, broadcasts m', then if a process delivers both m and m' then it must deliver m before m'.

4. ALGORITHM

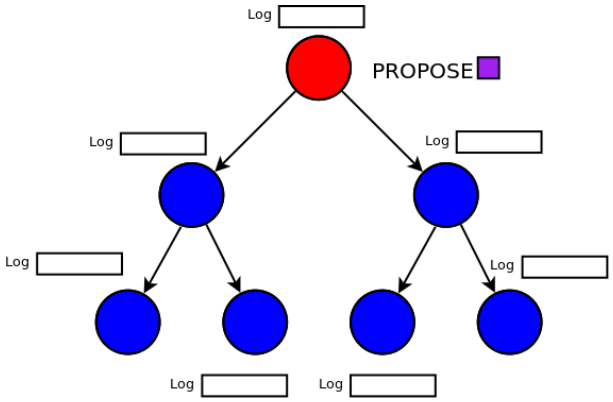
In this section we describe an initial version of the algorithm which is optimized in the next section. Our algorithm has two phases. While there is an active primary, we are in the *Broadcast* phase. When the active primary fails, a new one must be chosen in the *Recovery* phase. We first discuss the Broadcast phase followed by the Recovery phase.

4.1 Broadcast Phase

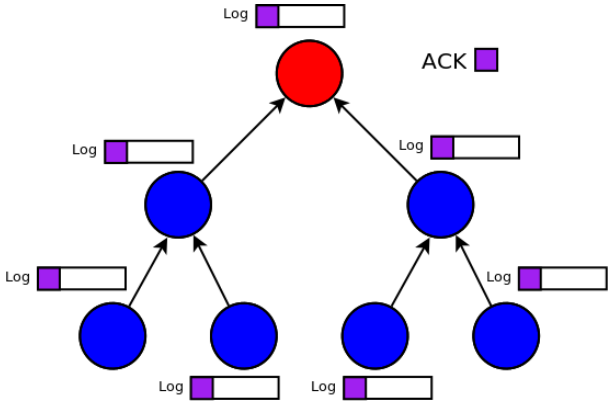
The broadcast phase is illustrated in Figure 1. Conceptually the broadcast phase is a two-phase commit without aborts. To ensure that messages which are delivered do not get lost, first the primary sends a $PROPOSE$ message including the message to be delivered. All processes that receive a propose message shall store the message in a local log but not deliver it. All processes then send an ACK back to the primary. Once all processes have acknowledged the reception of the proposal, the primary sends a $COMMIT$ message which causes all processes to deliver the previously proposed message to the clients. Note that the $COMMIT$ does not need to contain the contents of the message to be delivered, as it was contained in the $PROPOSE$. Instead, an identifier is sufficient to correlate the $COMMIT$ to an earlier $PROPOSE$.

Figure 1: The Broadcast Phase

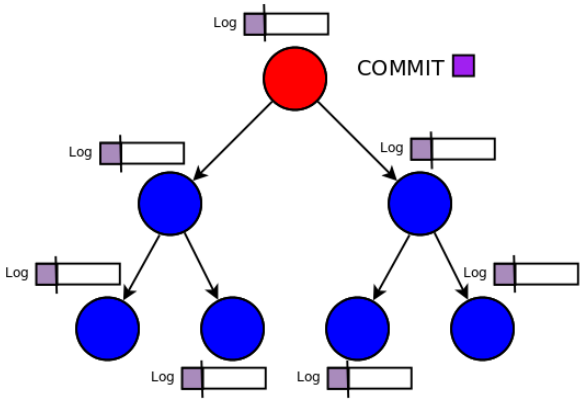
(a) The primary (in red) issues a $PROPOSE$ message which is sent to all processes down the tree



(b) Acknowledgments are combined back up the tree



(c) Once all correct processes have acknowledged the proposal, the primary sends a $COMMIT$ message down the tree



To efficiently send these messages, the primary uses a tree overlay network. The tree has the property that for every sub-tree, the root is the lowest ranked process in that subtree. We denote the largest ranked process in the subtree rooted at p, max_p. Another property of the tree is that for all processes p, the subtree p contains all correct processes in the interval $[p, max_p]$. We discuss how such a tree is formed in the next subsection. This structure allows for each process p to be responsible for sending messages to all correct processes in the interval $[p, max_p]$. This is done recursively by having each child responsible for a subset of the range of processes in the tree. A process that receives a message passes it on to its children who in turn pass it on to their children. Each process is responsible for only sending two messages (assuming a binary structure) and the longest path only goes through logarithmically many processes.

Acknowledgments flow in the reverse direction. When a leaf process receives a $PROPOSE$ message it sends an acknowledgment to its parent in the tree. Once a process p has detected that all of the processes in its subtree (the processes in the interval $[p, max_p]$) have either failed or acknowledged the message [1], p sends an ACK to its parent. In the case of no failures, ACK messages from all children is sufficient to determine that an ACK may be sent to the parent. To understand what happens in the case of failures, we observe that an ACK from process p represents that all correct processes in the interval $[p, max_p]$ have received the proposal. A process can send an ACK to its parent once the intervals for which and ACK has been received cover the set of correct processes. This means that a process must keep track of the set of correct processes in its subtree to determine that an ACK can be sent up.

If a process p recognizes that its parent has failed then it attempts to connect to its closest ancestor. This is illustrated in Figure 2. Given that p may have missed a number of messages since becoming disconnected from the tree. It sends a $RECONNECT$ message to its new parent which includes its state: the most recently seen proposal, most recently acknowledged proposal, and the most recently seen commit. Note that while a process reconnects others may be concurrently acknowledging proposals as seen in Figure 2b. The new parent, using its log, re-sends any proposals and commits that have been missed. The child then in turn sends these messages to their children as before. A simple optimization is to send all the proposals and the last committed message id as a single message upon reconnect. The parent also adopts this new child and sends it any future messages. As long as the primary does not fail, a process will always find an ancestor.

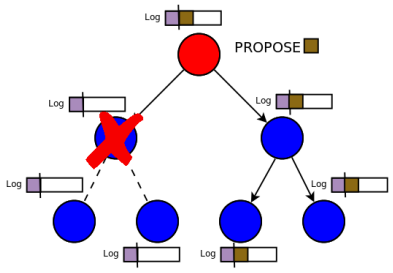
Before we explain how recovery occurs in the case where the primary fails, we discuss a few properties of this broadcast. All operations scale logarithmically in the number of processes. In practice, due to failures, the tree may become quite skewed due to parents adopted many children. Indeed, prior work has shown that rerouting over a tree to handle failures can degrade performance [12]. Therefore, it may be necessary to occasionally rebalance the tree.

Additionally, one may be concerned that the log of messages must grow indefinitely. However, any message which

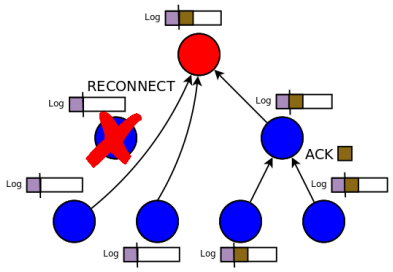
[1]Here we rely on the failure detectors to inform us that a failure has occured. As such in the presence of failures the latency of an acknowledgment is determined in part by the failure detector's timeliness

Figure 2: A non primary failure

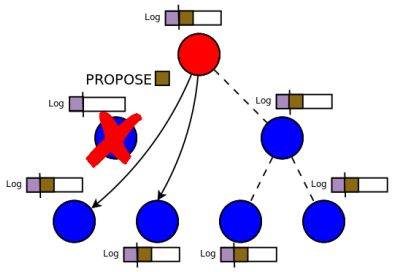
(a) If a failure occurs, some processes may miss messages



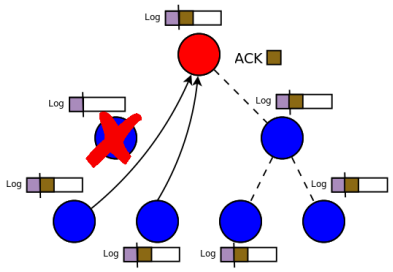
(b) Once a process recognizes that its parent has failed, it attempts to reconnect to its nearest ancestor



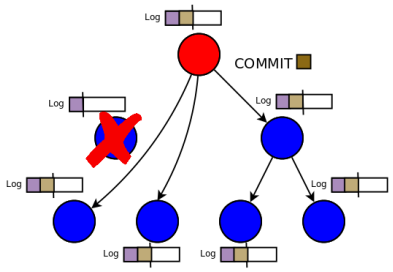
(c) The new parent sends any missed messages



(d) The children acknowledge any proposals as before



(e) The new parent continues to send new messages to the children



is committed is known to be logged on all processes and so is safe to be deleted (it is only necessary to know the id of the most recently committed message). So the log only needs to be as long as the maximum number of outstanding proposals. Such a limit can be easily imposed by the primary by allowing only m proposals to be outstanding.

4.2 Recovery Phase

The Recovery phase begins when the primary has failed. Once a process determines that all lower ranked processes have failed, it knows it shall be the next primary. Due to the properties of the failure detector, this is an accurate assessment. The first responsibility of a new primary is to construct a tree overlay network. Once the tree is completed, the primary must ensure that any message that was previously delivered by any process gets delivered by all processes before new messages get proposed. Because all messages are acknowledged by all correct processes before a commit message is sent, the new primary must have all messages that could have been committed by any process.

The new tree construction is quite complicated due to the fact that processes that have not been connected to the tree yet cannot initiate a reconnection on the new tree because they are unaware of its existence. The new tree is constructed by having the primary, ρ, select the next smallest ranked correct process, min_ρ, and the median ranked correct process med_ρ, as its children. These children are then responsible for the processes in the range $[min_\rho, med_\rho - 1]$ and $[med_\rho, max_\rho]$ respectively. The primary then sends a $CONSTRUCT\_TREE$ message to its children containing the maximum rank they are responsible for. This message also must contain the child's path to the root of the new tree to allow it to recover from failures. This path is at most logarithmic in the number of processes. The children then recursively choose children and assign ranges. Once a leaf process receives a message from its parent, it acknowledges successful construction of the tree with an ACK_TREE message. Once a parent receives acknowledgments from both of its children, it sends an acknowledgment to its parent. In the event that the child of a parent fails, the parent chooses the next smallest ranked process (compared to the failed child) as its new child and attempts to reconstruct the subtree again.

Once the tree is constructed, the new primary sends the set of outstanding proposals to all processes; this is a done by sending a $RECOVER\_PROPOSE$ message. Some processes may have proposals that were not seen by the new primary. These proposals could not have been committed as there are processes which have not seen the proposal (at least the primary has not), and so they are deleted. Once all correct processes acknowledge reception of the proposals with a $RECOVER\_ACK$ message from the primary, the primary can issue a $RECOVER\_COMMIT$ message which causes all outstanding proposals to be delivered. Once this is complete, the recovery is complete and the broadcast is re-entered. One may be concerned with sending all outstanding proposals to recover, as this may be quite a large message. One optimization would be for each child to send to its parent the id of the most recently seen proposal in the ACK_TREE message. The parent then only needs to send those proposals that the child does not have (or a message to truncate the log if necessary).

Because there may a delay between processes recognizing that a primary has failed, it is possible that a process receives messages from multiple trees. Therefore, included with all messages sent in both phases is the rank of the current primary. This allows a process to ignore messages sent on a tree from an earlier, failed, primary. The first message any process will receive on a new tree is a $RECOVER\_PROPOSE$ message which may cause a process to adopt a higher ranked primary. Once all processes have acknowledged a $RECOVER\_PROPOSE$ it can be determined that no $PROPOSE$ or $COMMIT$ messages from earlier primaries will be handled.

5. REFINEMENT

In the case of no primary failure, the latency of a single total order broadcast is the latency of the $PROPOSE$ message being sent and acknowledged by all processes plus the latency of the $COMMIT$ message. While each part scales logarithmically in the number of processes, the latency is quite sensitive to failures. If a single non-primary process fails before acknowledging the $PROPOSE$ message, then the entire broadcast is delayed until its children detect the failure and reconnect to the tree to receive the broadcast. This may take quite some time as, in practice, the failure detectors may take a while to be confident that a process has failed. As the number of processes in the system increases, the likelihood of any process failing, and therefore a delay occurring, increases.

The purpose of waiting for the acknowledgment from all processes is to ensure that every message is sufficiently replicated in case of a primary failure. However, it is likely unnecessary to replicate the message to all processes to tolerate primary failure. If we would like to tolerate f failures then $f + 1$ replicas is sufficient. We expect that for many applications f is significantly smaller than the total number of processes. We wish to send the message to all processes for the purposes of availability, but this can be done out of band from the replication of a message to tolerate primary failure. Doing so increases throughput and decreases latency of broadcasts.

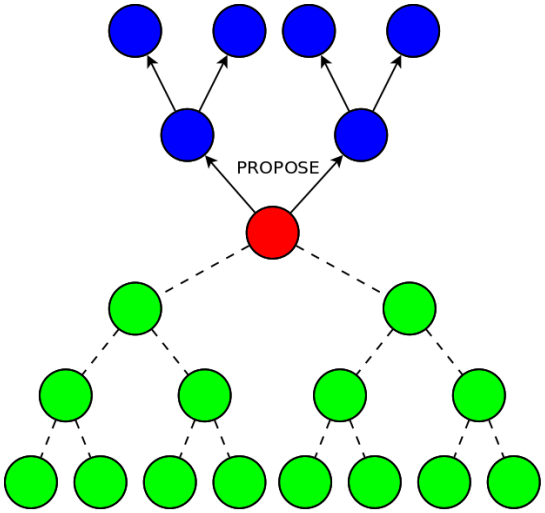
We therefore propose a modification to the algorithm described above. The modification is illustrated in Figure 3. The first $f + 1$ processes shall be called *replicas*. Only a replica is allowed to become the primary. When a new primary takes over it constructs a tree of replicas and issues proposals and commits to them as before. The remaining processes are called *listeners*. A new primary constructs a second tree containing only listeners. Instead of sending proposals and awaiting acknowledgments, the primary only needs to send an $INFORM$ message to all the listeners once a proposal has been acknowledged by all replicas. The listeners then immediately deliver the message. Remembering that a $COMMIT$ message only needs to contain the id of a message previously sent and not the contents of the entire message, it is not sufficient to send $COMMIT$ messages. Rather, the $INFORM$ must contain the message contents.

The advantage of this modification is that any listener failure (which is considerably more likely than any replica failure) only delays reception of a message for the members of the subtree rooted at the failure site. Further messages can still be proposed and acknowledged while this failure is repaired. Once the failure is repaired, the subtree will receive all missed $INFORM$ messages.

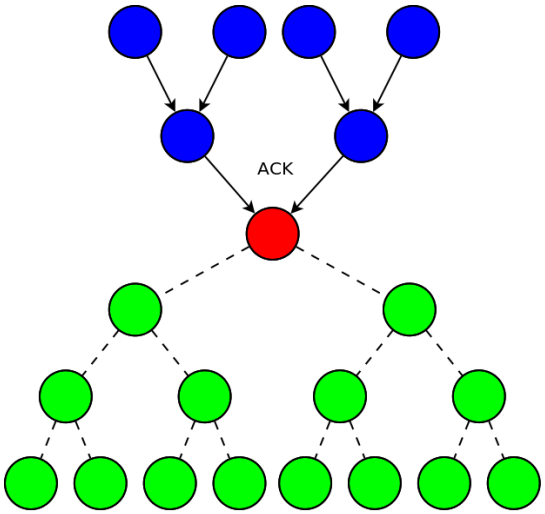
Recovery proceeds similarly, only replicas re-

Figure 3: Modified Algorithm Broadcast Phase

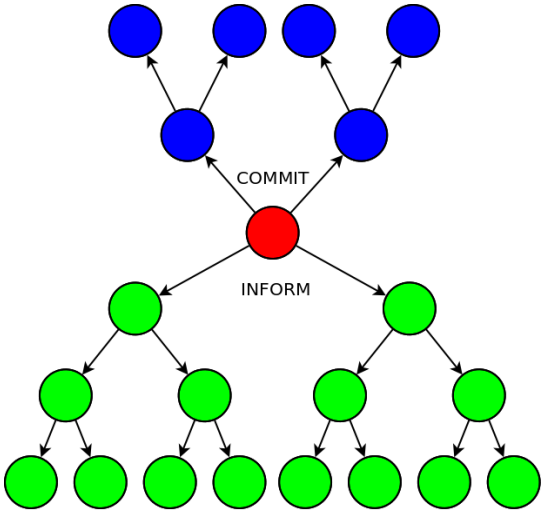
(a) The *PROPOSE* message is only sent to the replicas (in blue) on a separate tree



(b) The replicas combine acknowledgments back to the primary as before



(c) Once all correct replicas have acknowledged the proposal, a *COMMIT* message is sent to the replicas while a *INFORM* message is sent to the listeners (in green)



ceive *RECOVER_PROPOSE* messages and acknowledge them. When the new primary sends a *RECOVER_COMMIT* message to the replicas, it also sends a *RECOVER_INFORM* message to the listeners.

One issue that may arise is that it is not possible to bound the number of messages that a listener has missed during a failure. Processes therefore must keep a complete history of messages in order to handle all possible reconnects. This could be addressed in a number of ways. For example, only a limited number of messages are stored in the log and a snapshot that represents the delivery of older messages could be sent in the event that the log is not sufficient.

6. CONCLUSION

In this paper we have presented a scalable, fault tolerant total order broadcast algorithm. A single process, the primary, is responsible for ordering all messages. Messages are replicated sufficiently before delivering them in order to recover from a primary failure. The algorithm makes use of a tree overlay network to efficiently replicate, acknowledge, and deliver messages. Failures in the overlay network are efficiently routed around by having children reconnect to their nearest ancestor.

The latency of a single broadcast scales logarithmically with the number of processes participating in the broadcast. We believe that this will enable future exascale systems to take advantage of the strong ordering guarantees we provide even in the face of failures. Additionally, due to the modification of our initial algorithm, most failures will not delay future messages from being replicated, acknowledged and delivered. This is critical at scales where failures are common place.

We have defined a more formal specification for our algorithm and derived an associated proof of its correctness. We will present both in future publications. We are currently developing an implementation targeting Bluegene for experimental evaluation. Our next steps will be to integrate an implementation into EbbRT in order to implement the EbbId namespace. As part of this work we will expand our approach to permit nodes to be dynamically added to the namespace and hence extend the algorithm as necessary. Finally we will evaluate the algorithm and associated implementations for their fault tolerance and performance.

7. REFERENCES

[1] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling Autonomic Behavior in Systems Software with Hot Swapping. *IBM Syst. J.*, 42(1):60–76, January 2003.

[2] Jonathan Appavoo, Dan Schatzberg, James Cadden, and Orran Krieger. Ebbrt. In *OS/R Workshop*. DOE, 2012.

[3] Jason Baker, Chris Bond, James C. Corbett, Jj Furman, Andrey Khorlin, James Larson, Jean-Michel Le ÌĄon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services, 2011.

[4] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 193–203, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithmic based fault tolerance applied to high performance computing. *CoRR*, abs/0806.3121, 2008.

[6] Darius Buntinas. Scalable distributed consensus to support mpi fault tolerance. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 325–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[8] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[10] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[11] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[12] Joshua Hursey and Richard L. Graham. Preserving collective performance across process failure for a fault tolerant mpi. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1208–1215, Washington, DC, USA, 2011. IEEE Computer Society.

[13] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: an mpi proposal for process fault tolerance. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 329–332, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Michael Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, April 2007.

[15] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[16] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[17] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1–2:6, New York, NY, USA, 2008. ACM.

[18] Fred B. Schneider. Distributed systems (2nd ed.). chapter Replication management using the state-machine approach, pages 169–197. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.