

# Scalable Distributed Consensus to Support MPI Fault Tolerance<sup>\*</sup>

Darius Buntinas

Argonne National Laboratory

**Abstract.** As system sizes increase, the amount of time in which an application can run without experiencing a failure decreases. Exascale applications will need to address fault tolerance. In order to support algorithm-based fault tolerance, communication libraries will need to provide fault-tolerance features to the application. One important fault-tolerance operation is distributed consensus. This is used, for example, to collectively decide on a set of failed processes. This paper describes a scalable, distributed consensus algorithm that is used to support new MPI fault-tolerance features proposed by the MPI 3 Forum's fault-tolerance working group. The algorithm was implemented and evaluated on a 4,096-core Blue Gene/P. The implementation was able to perform a full-scale distributed consensus in 305  $\mu$ s and scaled logarithmically.

**Keywords:** MPI, Fault tolerance, distributed consensus, MPI\_Comm\_validate\_all, MPI 3

## 1 Introduction

As process counts in applications grow toward exascale, the length of time an application can run without experiencing a failure, known as the mean time between failures (MTBF), decreases. Applications will need to be fault tolerant in order to be useful on future exascale machines. Checkpointing can provide fault tolerance to an application without the need to modify it. As the failure frequency increases, however, checkpoints will need to be taken more often, decreasing the amount of useful work the application can perform between failures.

Whereas checkpointing provides fault tolerance to an application in a transparent manner, when using algorithm-based fault tolerance (ABFT) [1][3][4], the application is aware of faults and handles them explicitly. The fault-tolerance working group of the MPI 3 Forum has been working on a proposal [5], that adds fault-tolerance features to MPI in order to support ABFT applications. The proposal defines the behavior of an MPI library if processes fail. For example, existing operations such as MPI\_Comm\_split are now required to either succeed at every process or return an error at every process, even if processes fail before or during the operation. The proposal also introduces new functions,

---

<sup>\*</sup> This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

such as `MPI_Comm_validate_all`, that require all processes to return the same list of failed processes. A distributed consensus algorithm is needed to implement these operations.

This paper presents a scalable, fault tolerant, distributed consensus algorithm used to implement the `MPI_Comm_validate_all` function. The `MPI_Comm_validate_all` implementation is evaluated on a 4,096-core IBM Blue Gene/P machine and shows  $O(\log n)$  scaling.

The rest of the paper proceeds as follows. In Section 2 we review related work. In Section 3 we describe the problem. In Section 4 the algorithm is presented. In Section 5 we evaluate the performance. In Section 6 we conclude the paper and briefly discuss future work.

## 2 Related Work

The proposal presented by the MPI 3 fault-tolerance working group [5] has a fail-stop fault model; that is, failed processes stop sending messages. Transient failures and network failures, including network partitioning, are not considered in the proposal, although the working group is considering additional proposals to address these issues. The proposal also requires the failure detector to be *perfect*, as defined in [2]. A *perfect* failure detector will not report a nonfailed process as failed and eventually will report all failures to all processes.

The proposal is based on the concept that processes must explicitly recognize, or acknowledge, failures. When a process performs a point-to-point communication operation with a failed process, the operation will return an error. The process can recognize individual process failures locally (i.e., not collectively). Once the failure is recognized, any communication with that failed process is equivalent to communication with `MPI_PROC_NULL`. Processes can query for the set of failed processes it has discovered with the `MPI_Comm_validate` function. Note that this function is not collective, and the set of failed processes returned may be different at different processes.

Collective communication on a communicator with a failed process is not guaranteed to succeed at every process until the failure is recognized *collectively* by using the `MPI_Comm_validate_all` function. This function returns the same set of failed processes to all processes and recognizes the failures. This function is necessary so that the MPI library at all processes can adjust the collective communication patterns as necessary to account for failed processes. This operation requires distributed consensus.

Two-phase commit [7], three-phase commit [11], and Paxos [9] are the classical methods for achieving distributed consensus. These algorithms have scalability issues in that the coordinator process sends and receives messages individually from every process. Work has been done to improve scalability in [10] and [8]; however, these solutions are targeted for database systems that might have only tens or hundreds of committing processes in a large-scale system and are not sufficiently scalable to be used in exascale systems. The algorithm presented in this paper uses a reliable broadcast tree to distribute and collect messages, mak-

ing the algorithm highly scalable. The Paxos algorithm is tolerant to network partitioning, which is a failure mode not considered in this paper.

In [6] Fischer, et. al., proved that distributed consensus in an asynchronous model with one faulty process is impossible in a finite number of steps. Our algorithm does not guarantee consensus in a finite number of steps; rather, it will reach consensus with a probability of 1.

### 3 Problem Description

We present the distributed consensus algorithm as it would be used in the `MPI_Comm_validate_all` operation. However, the algorithm could also be used in other operations requiring distributed consensus, such as `MPI_Comm_split`. We first list the assumptions we make on the environment and then describe the `MPI_Comm_validate_all` function.

#### **Assumptions on the environment:**

1. The only failures will be process failures. Communication errors are masked by the MPI implementation. We do not consider network partitioning in this paper.
2. Process failures will be fail-stop failures. Once a process fails, it will stop sending messages.
3. Failure detectors are *perfect*; in other words, nonfailed processes will not be reported as failed, and all processes will eventually be notified of all failed processes.
4. Processes do not spontaneously recover after failure. Once a process has failed, it will remain failed.
5. There will always be a point in time in the future when no processes fail long enough to allow the broadcast algorithm described below to complete.

The `MPI_Comm_validate_all` function uses distributed consensus to decide on a set of failed processes, which must contain every failed process known by any participating process at the time the function is called. The same set of failed processes must be returned by the function at every process. If a process fails during the `MPI_Comm_validate_all` operation (i.e., after the first process calls the function and before the first process returns), the set of failed processes returned may or may not contain the failed processes.

The proposal discusses allowing “loose” semantics for the `MPI_Comm_validate_all` operation. The loose semantics would allow, in the case some processes fail before all processes complete the operation, for one result to be returned to the processes that failed and another result to the remaining processes. The idea is that even though different results were returned to different processes as a result of process failure during the operation, all of the remaining processes received the same result. Implementing loose semantics would be simpler and faster than implementing strict semantics. The choice of loose vs. strict semantics would be left to the user by, for example, setting an environment variable.

## 4 Algorithm

In this section, we give a brief overview of the algorithm at a high level and then describe the algorithm in detail.

The algorithm is similar to the three-phase commit algorithm except that, rather than sending and receiving individual messages, a reliable broadcast algorithm is used to send and collect messages. In the **BALLOTING** phase, after the root is chosen, the root creates a *ballot* containing the set of failed processes and broadcasts it to the rest of the processes. Once the processes receive the ballot, the responses to the ballot are collected back up the tree. If all the processes have accepted the ballot, the algorithm enters the **COMMIT** phase; otherwise a new ballot is generated, and the **BALLOTING** phase is repeated. In the **COMMIT** phase the root broadcasts a commit message. Once all processes receive the commit message, acknowledgments are collected back up to the root. The last phase is the **ALL\_COMMIT** phase. In this phase the root broadcasts the all-commit message. Once a process receives the all-commit message, it can return from the `MPI_Comm_validate_all` function.

### 4.1 Basic Reliable Tree Broadcast Algorithm

The reliable tree broadcast algorithm is used to ensure messages are received by all processes in the presence of process failure. Figure 1 shows the algorithm in guarded action form. A guarded action consists of a predicate and a set of statements. When the predicate is true, the action is *enabled*. When the program is executed, any enabled action is chosen nondeterministically, and its statements are executed atomically. A fair *scheduler* is assumed so that any action enabled for an infinite amount of time will eventually be scheduled.

Initially, all processes start with `direction` set to `UP`. When the broadcast algorithm is initiated, the root process chooses a broadcast number as a number larger than any broadcast number it has seen, computes its set of children, and sends the `BCAST` message to each child. It then sets its `direction` to `DOWN`. When the `BCAST` message arrives at the child, the child sets its broadcast number to the broadcast number of the `BCAST` message, sets its parent to be the sender of the `BCAST`, and computes its set of children. If the child has children of its own, it forwards the `BCAST` to its children and sets its `direction` to `DOWN`. Otherwise, it sends an `ACK` to its parent and sets its `direction` to `UP`. Once a process receives an `ACK` from every child, it forwards an `ACK` to its parent and sets its `direction` to `UP`. `ACK` and `NAK` messages with old broadcast numbers are ignored. `BCAST` messages with broadcast numbers that are not higher than the process's current broadcast number are `NAKed`. If no processes fail, eventually the root will receive an `ACK` from every child, and the broadcast algorithm completes.

If a process fails, however, then the failed process's parent sends a `NAK` to its parent and sets its `direction` to `UP`. If a process receives a `NAK` from its child, the process forwards the `NAK` to its parent and sets its `direction` to `UP`. Broadcast numbers make sure that once a process receives a new `BCAST` message it will ignore any messages from the previous broadcast operation.

The set of children is computed by using the `compute_children` function shown in Figure 1. Given a set of descendants, the process chooses a child from that set and assigns all processes in the descendant set with higher ranks than the child as the child’s descendant set. The child and its descendants are removed from the process’s descendant set, and the process repeats until the process’s descendant set is empty. The root process’s descendant set is initially set to the set of all live processes except itself. When a process sends a `BCAST` message to a child, it includes the child’s descendant set. The child uses this set to compute its children and their descendant sets. Note that if, when choosing a child from its descendant set, a process always chooses a descendant with a rank closest to the median rank, this broadcast algorithm will generate a binomial tree.

We can make the following observations about the algorithm. If the root receives an `ACK` from each child, all processes must have received a `BCAST` message (the *safety property*). If a process other than the root fails, the root will receive a `NAK` message. If the root does not fail, the root will eventually receive either an `ACK` from every child or a `NAK` from one of its children (the *progress property*). Root failure is not handled by the broadcast algorithm but will be handled by the caller by, for example, retrying the broadcast as described below. (Because of space limitations, formal proofs are omitted.)

## 4.2 Validate-All Algorithm

The validate-all algorithm starts with all processes in the `BALLOTING` phase. The root is chosen to be the lowest ranked live process. Since we assume that processes do not spontaneously recover and that there are no false positives in a process’s knowledge of failed processes, there will never be more than one live process that believes it is the root. The root creates a ballot, which consists of a set of failed processes. A slightly modified version of the basic broadcast algorithm is used to distribute the ballot and collect votes.

We modify the basic broadcast algorithm by introducing new message types. Except as described below, the `BALLOT_BCAST` message is treated as a `BCAST` message and `REJECT_ACK` and `ACCEPT_ACK` messages are treated as `ACK` messages. A `BALLOT_BCAST` carries a ballot along with the broadcast number and descendant set. Next, rather than replying with an `ACK` message, a process will reply with an `ACCEPT_ACK` if it finds the ballot acceptable, that is, if it knows of no failed processes that are not in the set of failed processes from the ballot and all of its children (if any) have replied with an `ACCEPT_ACK`. A process will reply with an `REJECT_ACK` otherwise. This effectively implements a *logical AND* reduction operation when gathering the `ACK` messages. In this way the root will receive an `ACCEPT_ACK` message from every child only if all processes find the ballot acceptable. Once the root receives an `ACCEPT_ACK` from every child, it knows that all processes found the ballot acceptable, and it enters the `COMMIT` phase. If some process does not accept the ballot, then the root’s knowledge of failed processes was stale when it issued the ballot; in this case, the root will wait for its knowledge of failed processes to be updated and broadcast a new ballot.

---

<i>Broadcast initiated at root</i>	→ Choose new <code>bcast_num</code> Compute set of children Send BCAST to children <code>direction = DOWN</code>	(1)
<code>recv BCAST</code> $\wedge \text{msg.bcast\_num} > \text{bcast\_num}$	→ <code>bcast_num = msg.bcast_num</code> <code>Parent = sender</code> Compute set of children if no children Send ACK to parent <code>direction = UP</code> else Forward BCAST to children <code>direction = DOWN</code>	(2)
<code>recv ACK from every child</code> $\wedge \text{msg.bcast\_num} == \text{bcast\_num}$ $\wedge \text{direction} == \text{DOWN}$	→ Forward ACK to parent, if any <code>direction = UP</code>	(3)
<code>child fails</code> $\wedge$ <code>direction == DOWN</code>	→ Send NAK to parent, if any <code>direction = UP</code>	(4)
<code>recv NAK</code> $\wedge \text{msg.bcast\_num} == \text{bcast\_num}$ $\wedge \text{direction} == \text{DOWN}$	→ Forward NAK to parent, if any <code>direction = UP</code>	(5)
<code>recv BCAST</code> $\wedge \text{msg.bcast\_num} \leq \text{bcast\_num}$	→ Send NAK to sender	(6)

---

```

compute_children(descendants)
  while descendants  $\neq \emptyset$ 
    choose child  $\in$  descendants
    descendants_child  $\leftarrow \{p \in \text{descendants} : \text{rank}(p) > \text{rank}(\text{child})\}$ 
    descendants  $\leftarrow \text{descendants} - (\text{descendants\_child} \cup \{\text{child}\})$ 
    children  $\leftarrow \text{children} \cup \text{child}$ 

```

**Fig. 1.** Reliable tree broadcast algorithm. Initially all processes start with `direction` set to `UP` and `bcast_num =  $\perp$` .

If a process other than the root should fail during the broadcast step, the root will receive a NAK message. The root then waits for its knowledge of failed processes to be updated and broadcasts a new ballot. If the root's knowledge of failed processes is still not up to date or additional processes have failed, then the broadcast will fail again, and a new ballot will be issued. The phase repeats until no new processes fail and the root's knowledge of failed processes is up to date. If the root process fails during a broadcast of the ballot, eventually the next lowest-ranked process will detect that the root has failed, and it will start the `BALLOTING` phase over again by broadcast a new ballot.

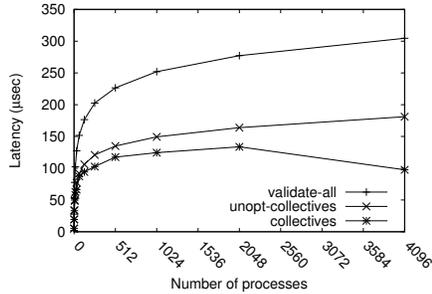
Because of our assumption that there will eventually be a period of time where no process fails long enough to complete the broadcast algorithm, we know that the broadcast will eventually complete, at which point the root will enter the `COMMIT` phase.

Once the root has committed, the root broadcasts a `COMMIT_BCAST` message to the other processes. If a nonroot process fails during the broadcast, the root will rebroadcast the message as described above. If the root fails, however, a new root is appointed; and if the new root has committed, it initiates a new broadcast of a `COMMIT_BCAST` message. It is possible that when the original root failed, some of the processes had not received the `COMMIT_BCAST` message, so the new root may not have committed. If the new root has not committed, it does not know whether the original root committed the last ballot. The new root must then restart the validate-all algorithm by broadcasting a new ballot. If another process has committed to the previous ballot, upon receiving the new ballot, that process will respond with a `FORCED_NAK` message containing the committed ballot, which will be forwarded back to the root. When the root receives the `FORCED_NAK` message, it enters the `COMMIT` phase and broadcasts the `COMMIT_BCAST` message including the committed ballot. Upon receiving the `COMMIT_BCAST`, a process will commit to the ballot included in the message and enter the `COMMIT` phase.

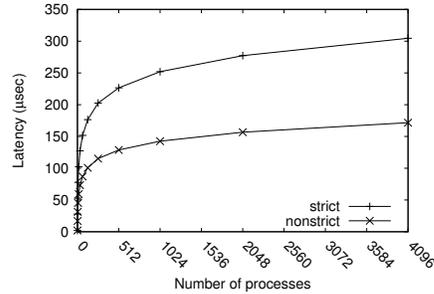
When the root receives an `ACK` from every child, it knows that every process has reached the `COMMIT` phase. At this point, even if the root fails, a new ballot will not be issued, and no process can commit to a different ballot. The root now enters the `ALL_COMMIT` phase and broadcasts the `ALL_COMMIT_BCAST` message. When a process receives the `ALL_COMMIT` message, the process can return from the `MPI_Comm_validate_all` function. However, every process must continue to handle protocol messages after returning from `MPI_Comm_validate_all` in the event that the `ALL_COMMIT_BCAST` needs to be rebroadcast.

### 4.3 Loose Semantics

In order to implement the loose semantics of `MPI_Comm_validate_all`, the acknowledgment of the `COMMIT_BCAST` and the broadcast of `ALL_COMMIT_BCAST` are omitted, and processes can return from the `MPI_Comm_validate_all` function once they reach the `COMMIT` phase, essentially implementing a two-phase commit. As before, processes need to continue to respond to protocol messages to ensure that the `COMMIT_BCAST` broadcast completes.



**Fig. 2.** Comparison of the validate-all operation with collectives operations performing a similar communication pattern, using Blue Gene/P optimized collectives and unoptimized collectives.



**Fig. 3.** Comparison of validate-all using strict and loose semantics.

Because the loose semantics algorithm implements a two-phase commit, rather than the three-phase commit, some processes may reach the `COMMIT` phase and return from `MPI_Comm_validate_all` but then fail before all processes have committed, leaving no live committed processes. In this case, the validate-all algorithm will be restarted, and the remaining processes will commit and return from `MPI_Comm_validate_all` with a set of failed processes different from the set previously returned by the failed processes. The user must decide whether to trade off the correctness of the distributed consensus for the performance gained by eliminating a phase of the algorithm.

## 5 Performance Evaluation

To evaluate the validate-all operation, we implemented it as an MPI program. This allowed us to evaluate the operation at a large scale on a Blue Gene/P without modifying the MPI implementation. We expect the performance of the operation implemented this way to be an upper bound on the performance of the operation if it were integrated into an MPI implementation. The evaluation was performed at Argonne National Laboratory on *Surveyor*, a Blue Gene/P with 1,024 quad-core nodes.

Figure 2 shows the results of the evaluation. As expected, the operation scales logarithmically. For comparison, we evaluated the time taken to perform a communication pattern similar to that of the validate-all operation using broadcast and reduction operations. The figure shows the results with optimized collectives using the Blue Gene/P collective tree network and with unoptimized collectives using the same torus network that the validate-all operation uses. At full scale, the validate-all implementation took 305  $\mu$ s to perform the operation, which is 1.66 times slower than performing a similar communication pattern with unoptimized collectives. We expect the performance of the validate-all algorithm to

improve when the operation is integrated into the MPI implementation, making the algorithm more responsive to incoming messages.

We also evaluated the performance of the operation with loose semantics. Figure 3 shows the comparison. The loose implementation performs the operation 133  $\mu$ s faster at full scale than does the strict implementation (which is 1.78 times as fast). Depending on the requirements of the application and the frequency at which the application calls `validate-all`, using the loose implementation can provide some performance improvement.

## 6 Conclusion

This paper presented a scalable distributed consensus algorithm used to implement the `MPI_Comm_validate_all` operation proposed by the MPI 3 fault-tolerance working group. The algorithm was evaluated on a 4,096-core Blue Gene/P machine and was shown to be extremely scalable. The implementation was able to perform a full-scale `validate-all` operation in 305  $\mu$ s and scaled logarithmically.

Using the loose implementation saved only 133  $\mu$ s over the strict implementation. Therefore, unless the application performs many `validate-all` operations, relaxing the semantics is unlikely to improve the overall performance of the application significantly.

We intend to implement the `MPI_Comm_validate_all` operation in MPICH2. We expect that this implementation will improve the responsiveness of the algorithm and hence improve its performance somewhat. Furthermore, we intend to use a similar algorithm to implement other operations requiring distributed consensus, such as the communicator creation routines.

## Acknowledgments

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

## References

1. Anfinson, J., Luk, F.T.: A linear algebraic model of algorithm-based fault tolerance. *IEEE Transactions on Computing* 37, 1599–1604 (1988)
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43, 225–267 (March 1996)
3. Chen, Z., Dongarra, J.: Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems* 19(12) (December 2008)
4. Chen, Z., Dongarra, J.: Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Transactions on Computers* (July 2009)
5. Fault Tolerance Working Group: Run-through stabilization proposal, [http://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run\\_through\\_stabilization](http://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization)

6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 374–382 (April 1985), <http://doi.acm.org/10.1145/3149.214121>
7. Gray, J.: Notes on data base operating systems. In: *Operating Systems, An Advanced Course*. pp. 393–481. Springer-Verlag, London, UK (1978), <http://portal.acm.org/citation.cfm?id=647433.723863>
8. Jurczyk, P., Xiong, L.: Adapting commit protocols for large-scale and dynamic distributed applications. In: *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems*. pp. 465–474. OTM '08, Springer-Verlag, Berlin, Heidelberg (2008), [http://dx.doi.org/10.1007/978-3-540-88871-0\\_33](http://dx.doi.org/10.1007/978-3-540-88871-0_33)
9. Lamport, L.: The part-time parliament. *ACM Trans. Comput. Syst.* 16, 133–169 (May 1998), <http://doi.acm.org/10.1145/279227.279229>
10. Ranganathan, S., George, A.D., Todd, R.W., Chidester, M.C.: Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing* 4, 197–209 (July 2001), <http://dx.doi.org/10.1023/A:1011494323443>
11. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng.* 9, 219–228 (May 1983), <http://dx.doi.org/10.1109/TSE.1983.236608>

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.