Corrected Trees for Reliable Group Communication

Martin Küttler  
TU Dresden  
martin.kuettler@os.inf.tu-dresden.de

Maksym Planeta  
TU Dresden  
mplaneta@os.inf.tu-dresden.de

Jan Bierbaum  
TU Dresden  
jan.bierbaum@os.inf.tu-dresden.de

Carsten Weinhold  
TU Dresden  
weinhold@os.inf.tu-dresden.de

Hermann Härtig  
TU Dresden  
haertig@os.inf.tu-dresden.de

Amnon Barak  
Hebrew University of Jerusalem  
amnon@cs.huji.ac.il

Torsten Hoefler  
ETH Zurich  
htor@inf.ethz.ch

CCS Concepts  • Theory of computation → Massively parallel algorithms; • Mathematics of computing → Trees; • Software and its engineering → Software fault tolerance; • Ultra-large-scale systems; • Networks → Network simulations; Logical / virtual topologies.

Keywords Low-Latency Broadcast, Gossip, LogP model, MPI, HPC

Abstract
Driven by ever increasing performance demands of compute-intensive applications, supercomputing systems comprise more and more nodes. This growth is a significant burden for fast group communication primitives and also makes those systems more susceptible to failures of individual nodes. In this paper we present a two-phase fault-tolerant scheme for group communication. Using broadcast as an example, we provide a full-spectrum discussion of our approach — from a formal analysis to LogP-based simulations to a message-passing-based implementation running on a large cluster. Ultimately, we are able to reduce the complex problem of reliable and fault-tolerant collective group communication to a graph theoretical renumbering problem. Both, simulations and measurements, show our solution to achieve a latency reduction of 50% with up to six times fewer messages sent in comparison to existing schemes.

1 Introduction
With the end of Dennard scaling, more parallel and distributed applications have to run on multiple nodes to satisfy their ever increasing computational demands. Those applications often follow the bulk synchronous parallel (BSP) execution model [43] and a typical trait of all BSP programs is a periodic collective communication phase, in which all processes of the program participate. Many BSP programs build on a library implementing the Message Passing Interface (MPI) standard [12]. However, MPI libraries are usually not fault tolerant and therefore not able to complete a collective operation correctly in case of any fault. Instead, the whole application will either hang or crash. So the overall system reliability decreases with the growing number of processes involved in the computation.

Even though first exascale systems are expected to appear as early as 2020 [37], many challenges, fault tolerance in particular, still remain to be solved [6]. Technology offers a multitude of partial solutions to address faults at various levels of the hardware-software stack [39], often several approaches need to be combined for maximum efficiency. For example, a heavy-weight system-level checkpoint-restart system [31] may create checkpoints less frequently [9], if the running application employs a cheaper roll-forward recovery for correcting at least some faults [15].

We propose Corrected Trees, a simple, yet efficient protocol for fault-tolerant collective group communication. Corrected Trees split the communication into two phases: dissemination and correction. Dissemination transfers the data as fast as possible over a tree structure (parent sends to children). Tree-based dissemination will, however, miss a large number of processes if any process close to the tree’s root fails. Correction reorganises all nodes into a ring (neighbors send to each other) and ensures that any process not reached by dissemination is included in the collective. Using these two basic phases, a variety of reliable MPI collectives can be built, e.g., applying correction before dissemination allows to create a reduction tree. In this paper, we focus on the broadcast operation, for which dissemination is followed by correction.

The mapping between a node’s position in the tree and the ring influences the impact failures have on the time required for correction. For simplicity, we always use a linear ring and reflect the different tree-to-ring mappings with the
Additionally, a single process failure close to the tree’s root (a) A failure in an in-order tree leaves a large gap of unreached nodes in the ring used for correction. An interleaved tree keeps the gaps small.

Figure 1. Impact of node failures on correction: comparison of in-order and interleaved dissemination trees.

tree structure. To highlight the importance of said structure, consider a binomial tree as implemented in the popular Open MPI [13] library. This library structures the tree nodes either in-order (Figure 1a, top) or interleaved (Figure 1a, bottom). A failed process has different effects: For the in-order tree, the resulting gap of unreached nodes is large and inefficient for correction to cover, whereas an interleaved tree creates smaller gaps. Figure 1b shows the expected length of the correction phase for both numbering schemes in a broadcast with 64K processes. Without faults, the correction phase would take 8 time steps. When faults do happen, the expected correction time of an interleaved tree slightly increases with a higher fault rate, but in this case does not exceed 10.5 time steps (vertical line). In contrast, the expected correction time of an in-order tree grows with the absolute number of faults and takes even longer for the larger trees. Additionally, a single process failure close to the tree’s root may result in correction taking orders of magnitude longer than on average. This shows how node ordering is crucial for scalability of a fault tolerant tree-base broadcast.

The contributions of this paper are the description and analysis of the Corrected Tree-based broadcast protocol, a simulator for LogP-like models with support for correction-based collectives, and a prototype implementation of our scheme. We also provide various optimizations of the correction algorithm and show that interleaving facilitates fast correction. The resulting protocol (1) tolerates faults, (2) provides low overhead with and without faults, and (3) does not impose unnecessary blocking. The simplicity of our approach is key: It enables an efficient practical implementation and extensive evaluation of high-performance reliable broadcasts. Our holistic study indicates substantial improvements over competing algorithms, both in latency and number of messages.

2 System Model for Evaluation

For the purpose of analysis, we use the following system model. In a broadcast operation among \( P \) processes, the root process propagates a message reliably to all other processes. Without loss of generality we assume the root process to have rank 0, other processes have ranks 1, \ldots, \( P - 1 \). We assume the message size to be small, meaning that it does not impact latency and messages do not need segmentation.

The usual fault-agnostic way to implement reliable small-message broadcast is by sending messages along the edges of a tree [41]. We call processes colored after receiving the broadcast message, and uncolored otherwise. The root process is always said to be colored.

2.1 Failures

In the presence of failing processes, the goal of the broadcast is to guarantee information propagation from the root process to every live process. Within a fault-agnostic setup a failure of any non-leaf process in the tree results in all its descendants remaining uncolored. For better comparability with Corrected Gossip [17] we use the same fail-stop fault model\(^1\): After a failure occurs, the failed process will neither send nor process any messages. Messages sent to a failed process thus have no effect, and, barring explicit acknowledgments, the sender cannot tell whether the recipient is alive. The root node is assumed to be alive, because it initiates the broadcast. To facilitate analysis, we assume independence of failures. Even though faults are rarely uncorrelated, independence can be achieved by numbering tree nodes in a random manner. Alternatively, the ring used for correction can be structured in a way that nodes having correlated failure probabilities stay far away from each other. Detailed analysis of this problem is out of scope of this paper as is the removal or reactivation of dead processes, which can be solved independently [5, 36].

For the purpose of this paper, we consider only a single execution of a broadcast operation. During this operation every process is either dead or alive, \( i.e. \), a process either sends all messages required by the protocol or none at all, but failures can occur anywhere outside of the broadcast. With failure-proof correction, full coloring can be guaranteed even if processes fail during the broadcast [17].

In terms of reliability guarantees [1, p. 171] a reliable broadcast has the following properties:

- **Integrity** every broadcast received was previously sent,
- **No duplicates** a process receives a broadcast message at most once,
- **Non-faulty liveness** a broadcast initiated by a live root is either received by all live processes or by none,
- **Faulty liveness** messages broadcast by faulty processes are either received by all live processes or by none.

\(^1\)Corrected Trees are likely to maintain their properties with other fault models as well, but further analysis is required.
We base the description and analysis of all the algorithms whether the receiver is alive. Similar assumptions are made with receive overhead on the same process. The parameter which may communicate with a uniform maximum latency to avoid duplicates for the user we ensure that a colored process. Faulty liveness is guaranteed due to failed processes entered the correction phase. The gossip phase attempts to color as many processes as possible, but due to potential failures and the random nature of gossip, the protocol cannot guarantee that all live processes are actually colored.

The correction phase tries to color these remaining processes. For correction, all processes are organized into a virtual ring according to their ranks (0 to \( P - 1 \)). Uncolored processes create gaps, where the maximum gap size is the length of the longest sequence of uncolored processes. All colored processes send messages to their neighbors in the ring. Processes that become colored during correction do not send any messages. There are three interchangeable correction algorithms: opportunistic, checked, and failure-proof with different trade-offs between overhead and reliability as detailed below. The reliability properties of all correction schemes have been proven, but to the best of our knowledge there was no practical implementation of Corrected Gossip.

**Opportunistic Correction** All processes colored by gossip send correction messages to a small set of neighbors: Process \( r \) sends to processes \( \{r + 1, r - 1, \ldots, r + d, r - d\} \), where \( d \) is the correction distance. Any non-colored process receiving a correction message becomes colored and concludes the broadcast. This correction algorithm colors all processes only if the maximum gap size does not exceed \( 2d \). Due to the probabilistic nature of gossip there are no absolute guarantees even in the failure-free case.

**Checked Correction** Each process colored in the gossip phase starts sending messages to its left and right neighbors, similar to opportunistic correction. A process stops sending messages into a particular direction, once it receives a message from this direction, from a process it had already sent a message to. For example, if process 23 received the nearest correction messages from processes 19 and 28, it continues sending correction messages, until it finally sent to both 19 and 28. Overall, process 23 sends correction messages to processes \( \{22, 24, 21, 25, 20, 26, 19, 27, 28\} \). Like in opportunistic correction, processes that get colored by correction do not send any messages themselves. This algorithm ensures that all live processes are colored, independently of the maximum gap size, as long as no failures occur during the correction.

**Failure-Proof Correction** A generalization of checked correction that guarantees each process to be colored even in the presence of failures during correction. We refrain from discussing this type of correction in detail due to its complexity and high overhead. Please refer to Corrected Gossip [17] for further details on any of the correction algorithms.

### 3.1 Revisiting Corrected Gossip Broadcast

Hoefler et al. [17] presented a fault-tolerant broadcast algorithm comprising a gossip-based dissemination followed by correction. The root sends the broadcast payload to a random subset of other processes. When a new process is thus colored, it starts sending messages as well. This dissemination runs for a fixed period of time, after which all colored processes enter the correction phase. The gossip phase attempts to color as many processes as possible, but due to potential failures and the random nature of gossip, the protocol cannot guarantee that all live processes are actually colored.

The correction phase tries to color these remaining processes. For correction, all processes are organized into a virtual ring according to their ranks (0 to \( P - 1 \)). Uncolored processes create gaps, where the maximum gap size is the length of the longest sequence of uncolored processes. All colored processes send messages to their neighbors in the ring. Processes that become colored during correction do not send any messages. There are three interchangeable correction algorithms: opportunistic, checked, and failure-proof with different trade-offs between overhead and reliability as detailed below. The reliability properties of all correction schemes have been proven, but to the best of our knowledge there was no practical implementation of Corrected Gossip.

**Opportunistic Correction** All processes colored by gossip send correction messages to a small set of neighbors: Process \( r \) sends to processes \( \{r + 1, r - 1, \ldots, r + d, r - d\} \), where \( d \) is the correction distance. Any non-colored process receiving a correction message becomes colored and concludes the broadcast. This correction algorithm colors all processes only if the maximum gap size does not exceed \( 2d \). Due to the probabilistic nature of gossip there are no absolute guarantees even in the failure-free case.

**Checked Correction** Each process colored in the gossip phase starts sending messages to its left and right neighbors, similar to opportunistic correction. A process stops sending messages into a particular direction, once it receives a message from this direction, from a process it had already sent a message to. For example, if process 23 received the nearest correction messages from processes 19 and 28, it continues sending correction messages, until it finally sent to both 19 and 28. Overall, process 23 sends correction messages to processes \( \{22, 24, 21, 25, 20, 26, 19, 27, 28\} \). Like in opportunistic correction, processes that get colored by correction do not send any messages themselves. This algorithm ensures that all live processes are colored, independently of the maximum gap size, as long as no failures occur during the correction.

**Failure-Proof Correction** A generalization of checked correction that guarantees each process to be colored even in the presence of failures during correction. We refrain from discussing this type of correction in detail due to its complexity and high overhead. Please refer to Corrected Gossip [17] for further details on any of the correction algorithms.
The failure of process \(T_f\) will be colored after the dissemination. Thus opportunistic correction with \(k - 1\) failures at level \(\ell\) leads to colored \(k^\ell\)-th process being uncolored. Thus \(k^\ell - 1\) failures on level \(\ell\) or below can be tolerated, and still every \(k^\ell\)-th process will be colored after the dissemination.

As an example, consider the right-hand tree from Figure 4 as \(T_f\), and node 1 as the root of \(T_f\). The adjacent pairs on the ring \(R_f\), built from \(T_f\) are: \((1, 3), (3, 5), (5, 7), (7, 1)\). Looking at the full tree, we see that for example pairs \((5, 6)\) or \((6, 7)\) are adjacent on the ring \(R_f\), but descend from different children of the root. This pattern holds for all subtrees and pairs, hence the tree is interleaved.

3.2 Interleaved Trees

Corrected Gossip is extremely robust against a high number of failures, but sends an excessive amount of messages. We show how to achieve a comparable level of resilience by reusing traditional fault-agnostic small-message broadcast protocols in combination with a correction phase. The general scheme stays familiar: first try to color as many processes as possible, then run a correction phase to ensure no live process remains uncolored. In contrast to Corrected Gossip, Corrected Tree-based broadcast requires correction only to tolerate faults, because non-faulty liveness is guaranteed in the fault-free case.

To ensure that correction can color uncolored processes quickly, the maximum gap size ought to be small. A tree maintaining such a property should have its subtrees spread across the correction ring to avoid the danger of having uncolored processes cluster together on the ring. For lowering correction latency multiple small gaps are better than few large ones. As we outlined in the introduction, an interleaved ordering of tree processes is particularly well suited for minimizing the maximum gap size. A binary in-order tree (Figure 3, left) can be constructed by numbering the processes in the order of depth-first traversal. A single fault (process 4) will leave two consecutive processes uncolored after the tree phase. The resulting gap has a size of 2. In contrast, the interleaved ordering (Figure 3, right) ensures that no process on level 2 shares a parent with its direct neighbors on the ring, e.g., process 4 is a child of process 2, and its direct neighbors 3 and 5 are children of process 1. The failure of process 2 will thus result in two gaps of size 1.

To define interleaved trees formally, consider a tree \(T_f\), where parent-child relationships between tree nodes reflect sender-receiver relationships between processes. A tree \(T_f\) has a corresponding ring \(R_f\), where the nodes are ordered according to the process ranks and the immediate neighbors in the ordering are connected (the first and last node are also connected). A subtree \(T_s\) of an interleaved tree \(T_f\) starts at some node of \(T_f\) with all of its descendants. Nodes of \(T_s\) map to a ring \(R_s\), that preserves the original relative order of the corresponding nodes in \(R_f\). We use \(\text{root}(T_f)\) to designate the root node of \(T_f\).

**Definition 1.** A tree \(T_f\) is interleaved iff for any of its subtrees \(T_s\) and a ring \(R_s\) comprising the nodes of \(T_s\), any adjacent pair of distinct nodes in \(R_s\) either descend from each other or their only common ancestor is \(\text{root}(T_s)\).

As an example, consider the right-hand tree from Figure 4 as \(T_f\), and node 1 as the root of \(T_f\), the adjacent pairs on the ring \(R_f\) built from \(T_f\) are: \((1, 3), (3, 5), (5, 7), (7, 1)\). Looking at the full tree, we see that for example pairs \((5, 6)\) or \((6, 7)\) are adjacent on the ring \(R_f\), but descend from different children of the root. This pattern holds for all subtrees and pairs, hence the tree is interleaved.

In the remainder of this section, we describe how to construct various interleaved trees. Although we focus on full trees, our node numbering scheme maintains the interleaving (and therefore the associated resilience properties) also for incomplete trees.

3.2.1 \(k\)-ary Tree

We start with \(k\)-ary trees as they are widely used and have the simplest interleaving scheme. Given the root process at level 0, a full \(k\)-ary tree has \(k^\ell\) processes at level \(\ell\), and \(k^\ell\) subtrees that have their root process at that level. The processes in these subtrees can have a distance of \(k^\ell\) in the ring. Figure 3 shows an example of this tree for \(k = 2\). In general, process \(r\) has the child processes \(r’\):

\[
\{r’ \mid r’ = r + i \cdot k^\ell, 0 < i \leq k, r' < P\}
\]

This interleaving ensures that a failing process on level \(\ell\) leads to every \(k^\ell\)-th process being uncolored. Thus \(k^\ell - 1\) failures on level \(\ell\) or below can be tolerated, and still every \(k^\ell\)-th process will be colored after the dissemination.

As an example, assume process 2 of the right-hand tree in Figure 3 failed. Its children (4 and 6) will not be colored during the dissemination. Note that these have a distance of 2 = \(k^0\) on the ring because the failed process 2 (the root of the uncolored subtree) is at level 1. The maximum gap size is 1 and every second process on the ring is colored. With \(k - 1\) failed processes at least every \(k\)-th process is colored after the dissemination. Thus opportunistic correction with \(k - 1\) messages sent per process is guaranteed to color all processes as long as there are less than \(k\) failures. No such guarantee can be given for \(k\) or more failures, because in the (very unlikely) worst case all children of the root process can fail. The expected size of uncolored gaps grows slowly with the number of failures (see Section 4.3).

3.2.2 Binomial and Lamé Trees

For describing binomial trees and their interleaving we find it helpful to use a notation proposed by Takaoka [40]. Specifically, Takaoka defines a linking operation “•” for trees \(T_a = T_b \bullet T_c\), where \(T_a\) is created by linking the root of \(T_c\) as a child to the root of \(T_b\). Consequently, the number of processes \(|T_a|\) in tree \(T_a\) is the sum of \(|T_b|\) and \(|T_c|\)
Hoefler and Moor [19] refer to these trees as Fibonacci trees. The children are computed as follows:

$$R_{i+1} = R_i + \frac{1}{2} R_{i+1}$$

The processes with lower ranks are considered to be created in sequence. At a given iteration, the children of a tree are ready to send if they have just finished their last send and can start a new one, in addition to $$R(i - k)$$ processes that just finished receiving messages.

To make the tree interleaved, new processes get ranks assigned in succession. At a given iteration, the children of the processes with lower ranks are considered to be created first. After $$t - 1$$ iterations a tree can color $$R(t + k - 1)$$ processes, then at iteration $$t$$ the root will create a child with rank $$R(t + k - 1)$$ and the remaining $$R(t) - 1$$ ready-to-send processes create children with subsequent ranks. In overall, the children are computed as follows:

$$\{r' \mid r' = r + R(i + k - 1), i \geq s', R(s') > r, r' < P\}$$

A Lamé tree of order $$k$$ is defined as

$$T_l = T_{l-1} \cdot T_{l-k}$$

With $$k = 1$$ the tree is binomial; in Section 3.2.3 we describe how a Lamé tree can be optimal. As a boundary condition we maintain that $$T_l$$ with $$t < 0$$ contains a single process. We build an interleaved tree iteratively. Starting from iteration $$t = 0$$ with one process that is ready to send, each process ready to send gets assigned a child. Processes created at an iteration $$t$$ become ready to send, i.e., can create children, at iteration $$t + k$$. The number of processes ready to send $$R(t)$$ is defined as

$$R(t) = \begin{cases} 0 & \text{if } t < 0, \\ 1 & \text{if } 0 \leq t < k, \\ R(t-1) + R(t-k) & \text{if } t \geq k. \end{cases}$$

Before $$t = k$$ only the root can send messages. At iteration $$t = k$$, the second process can start sending itself. In general, at iteration $$t$$, there are $$R(k - 1)$$ processes, that have just finished their last send and can start a new one, in addition to $$R(i - k)$$ processes that just finished receiving messages.

To make the tree interleaved, new processes get ranks assigned in succession. At a given iteration, the children of the processes with lower ranks are considered to be created first. After $$t - 1$$ iterations a tree can color $$R(t + k - 1)$$ processes, then at iteration $$t$$ the root will create a child with rank $$R(t + k - 1)$$ and the remaining $$R(t) - 1$$ ready-to-send processes create children with subsequent ranks. In overall, the children are computed as follows:

$$\{r' \mid r' = r + R(i + k - 1), i \geq s', R(s') > r, r' < P\}$$

**Lemma 1.** A tree constructed by Equation (2) is interleaved.

**Proof.** The Proof goes by induction.

**Base.** A tree with a single node is interleaved.

**Step.** Consider a tree $$T_{i+1}$$, constructed from an interleaved tree $$T_i$$. For that, nodes $$\{p \mid 0 \leq p < R(l - k + 1)\}$$ acquire nodes $$\{c \mid R(l) \leq c < R(l + 1)\}$$ as their children.

Now, consider all possible pairs of nodes in $$T_{i+1}$$. Nodes $$\{0 \leq i, j < R(l)\}$$ do not have a pair that would violate conditions for an interleaving tree, because $$T_i$$ is interleaved. Nodes $$\{R(l) \leq i, j < R(l + 1)\}$$ do not have a pair that would violate conditions for an interleaving tree, because any pair $$(i, j)$$ has nodes with different parents. Nodes $$i$$ and $$j$$ preserve the same relative order on a ring as their parents. The rule holds for any possible subtree $$T_i$$ that includes both $$i$$ and $$j$$. No pair of nodes $$\{(i, j) \mid 0 \leq i < R(l), R(l) \leq j < R(l + 1)\}$$ violates conditions for an interleaving tree, because $$i$$ is either ancestor of $$j$$; or the only common ancestor of both $$i$$ and $$j$$ is root($$T_i$$) (like for a pair of $$i$$ and the parent of $$j$$). No other pair exists, thus $$T_{i+1}$$ is also interleaved.

An example of a Lamé tree is given in Figure 5. At each iteration $$i$$ process 0 sends a message to processes with ranks $$R(i + 2)$$. Process 1 sends for the first time at iteration 3, as the smallest iteration when $$R(s') > 1$$ is $$s' = 3$$. Then process 2 can send at iteration 4, since $$R(4) = 3$$ and so on. If $$L = a = 1$$, this particular instance guarantees minimal latency. If network parameters change, the tree structure stays the same, though the protocol stops being latency-optimal.
3.2.3 Optimal Tree
Knowing exact timings of events allows to create a tree topology that is optimal with regard to latency [8, 26]. A Lamé tree can constitute an optimal communication topology when \(2o + L = k\) (see Figure 5). If, for example, the latency is higher than expected for a given Lamé tree, processes deeper in the tree will be delayed. As a result many processes will either wait until the delayed processes can start correction or start correction early (see Section 3.3).

We can improve the total latency if all processes stop communication approximately at the same time: Processes that start sending early send out more messages and no process sends more than one message after the last message of any other process. With the exact timings of the network model, it is possible to build an optimal communication graph [26] that ensures minimum latency.

Building an optimal communication tree \(T_t\) is similar to Lamé. Consider a subtree \(T_{t-2o-L}\) rooted at the recipient of the first message and a subtree \(T_{t-o}\) that is a complement of \(T_{t-2o-L}\) in \(T_t\). Then \(T_t\) is defined as:

\[
T_t = T_{t-o} \ast T_{t-2o-L}
\]

Here \(T_t\) represents an optimal tree using \(t\) time steps to send out messages. The first message of \(T_{t-2o-L}\) starts exactly \(o + L\) time steps later than \(T_{t-o}\), ensuring both trees finish simultaneously. Analogous to Equation (1), ready to send \(R(t)\) is defined as:

\[
R(t) = \begin{cases} 
0 & \text{if } t < 0, \\
1 & \text{if } 0 \leq t < 2o + L, \\
R(t - o) + R(t - 2o - L) & \text{if } t \geq 2o + L.
\end{cases}
\]

For an interleaved ordering, the children of \(r\) are determined similarly to Equation (2):

\[
\{r' \mid r' = r + R(i + o + L), i \geq s', R(s') > r, r' < P\}
\]

3.3 Correction with Trees
A correction phase follows the dissemination phase and is independent of the tree type. The correction phase ensures coloring of the processes that the dissemination phase left uncolored due to failures. All three correction algorithms presented in Corrected Gossip [17] and summarized in Section 3.1 are directly applicable.

Synchronized and Overlapped Correction The correction phase of Corrected Gossip starts at a pre-specified moment in time simultaneously on all processes. We call this mode of correction *synchronized*. The correction is *overlapped*, if a process running Corrected Tree-based broadcast, starts correction immediately after the dissemination phase. With overlapped correction, a correction message is *early* if a process is reached by the correction message, before a tree message. After receiving an early correction, a process still sends tree messages to its children. Early correction messages from opportunistic correction decrease latency. For checked correction, our experiments (omitted due to space constraints) have shown that early correction can both increase or decrease the latency. The latency may increase, if processes receiving early correction do not participate in the correction phase themselves, meaning that the processes participating in the correction phase are stopped later. The exact behavior is very sensitive to concrete timing characteristics, network model parameters, and tree type.

Optimized Opportunistic Correction Overlapping opportunistic correction from Corrected Gossip is straightforward to reuse: A process sends \(d\) correction messages in each direction on the ring immediately after sending tree messages. Tree-based dissemination allows a simple optimization of opportunistic correction that preserves non-faulty liveness. Given processes \(i\) and \(j\), such that \(j - d < i < j\), assume \(i\) receives a message from \(j\). Process \(i\) can be sure that \(j\) also covers processes \(j - d, \ldots, i + d\) with correction messages. Now, \(i\) has to send correction messages only to processes \(i - d, \ldots, j - d - 1\). For example, process 19 receives a correction message from process 23. With \(d = 8\), 23 surely sends messages to processes 22, \ldots, 15, so 19 has to send only to processes 14, \ldots, 11. For the rest of the paper Corrected Trees use optimized opportunistic correction.

Delayed Correction Guaranteed full process coloring in the fault-free case with tree-based dissemination enables an additional optimization, that minimizes the number of messages. For that, a node participating in the correction waits out a delay after sending the first correction message to the left. After a delay, a process expects to receive a correction message from the right. If no message has been received, the process starts sending messages to the right until a message from the right eventually arrives. If a process colored by dissemination receives a message from the left, it immediately replies to stop the sender. The delay should be short enough, so that a process must not *suspect* a neighbor to be faulty, if a correction message is missing. As result, the delay does not introduce a failure detector, although non-faulty liveness and termination are still guaranteed. The reduced overhead in the fault-free case comes at the cost of a higher latency when a failure does occur or when a live process sends a correction message too late. We do not evaluate delayed correction further, because the appropriate delay is application-specific.

4 Evaluation and Analysis
In this section we evaluate the performance of the Corrected Tree-based broadcast. We look at the two most important performance metrics: latency and network load. We define *coloring* latency as an interval between the time when the root starts sending the first message and the last process becomes colored. We also define *quiescence* latency as an
interval between the moment when the root starts sending
the first message and the moment when all activity related
to the broadcast operation is over. We measure network load
by the number of messages sent.

For experiments, we developed a discrete event simulator\textsuperscript{3}
to study collective operations with LogP-like models and
an MPI-based implementation to evaluate the algorithms
on a real system. Two main features of the simulator are
the possibility to model faults and run collectives with a
dynamically changing communication graph (used for checked
correction). All our simulations are fully reproducible as we
keep the random generator seed of every experiment. We
experimented with around 50 different broadcast variants and
various model parameters, but here we show only the subset
we deemed to be representative. Unless specified otherwise,
we model the following system parameters. For LogP, we
use $L = 2$, $o = 1$, which corresponds to the range of LogP
parameters measured on real systems \cite{18, 28, 34}. For the
Lamé tree, we chose $k = 2$. These parameters result in a
tree structure in-between a binomial and an optimal tree.
Opportunistic correction will send up to

\begin{align*}
\text{Messages per process} &= \{\text{Binomial}, \text{4-ary}, \text{Lamé}, \text{Optimal}, \text{Gossip}\} \\
\text{Correction Type (distance $d$)} &= \{\text{Opport. (1)}, \text{Opport. (2)}, \text{Opport. (4)}, \text{Checked}\} \\
\end{align*}

\textbf{Figure 6.} Average number of messages per process. Opportunistic Corrected Gossip sends more messages to color all the processes. A traditional tree-based broadcast first sends the payload (solid line), then it sends an acknowledgment in a second message (dashed line). Corrected Trees send fewer messages than gossip and still color all the processes reliably.

overlapped opportunistic correction. Additionally, we show
checked and opportunistic Corrected Gossip. We picked the
smallest gossiping time for opportunistic Corrected Gossip
where we observed no uncolored processes in $10^5$ simulations with 64K processes. For checked Corrected Gossip we
optimized gossiping time for the lowest latency. For the trees,
the average number of messages does not depend on the
number of processes. Corrected Trees require significantly
fewer messages for correction than Corrected Gossip. Opportunistic Correction allows Corrected Tree-based broadcast
to save even more on messages in exchange for lower reliability guarantees. The dashed line at two messages per
process represents a broadcast with acknowledgments or op-
portunistic Correction with only one correction message in
one direction. Despite stochastic nature of Corrected Gossip,
variation in the average number of messages was negligible,
given a concrete number of processes. Optimized opportunis-
tic correction sends at most as many messages as checked
correction. If some tree processes finish earlier, the average
number of messages decreases further, but the effect of op-
timization reduces with smaller correction radius $d$. With
synchronized checked correction, all trees exhibit the same
behavior and since there are no faulty processes, each of them
sends 5 correction messages (more detail in Section 4.2).

\textbf{Figure 7.} Quiescence latency in fault-free case. Gray ribbon shows 5% and 95% percentiles for checked Corrected Gossip. Solid lines stand for trees with acknowledgments. Dashed lines represent Corrected Tree-based broadcast.

For experiments, we developed a discrete event simulator\textsuperscript{3} to study collective operations with LogP-like models and
an MPI-based implementation to evaluate the algorithms
on a real system. Two main features of the simulator are
the possibility to model faults and run collectives with a
dynamically changing communication graph (used for checked
correction). All our simulations are fully reproducible as we
keep the random generator seed of every experiment. We
experimented with around 50 different broadcast variants and
various model parameters, but here we show only the subset
we deemed to be representative. Unless specified otherwise,
we model the following system parameters. For LogP, we
use $L = 2$, $o = 1$, which corresponds to the range of LogP
parameters measured on real systems \cite{18, 28, 34}. For the
Lamé tree, we chose $k = 2$. These parameters result in a
tree structure in-between a binomial and an optimal tree.
Opportunistic correction will send up to 4 messages in each
direction.

Section 4.1 discusses corrected tree- and gossip-based
broadcast in the failure-free case. In Section 4.2 we focus
on the correction phase. Section 4.3 compares how perfor-
ance changes for Corrected Tree and Gossip-based broad-
cast when some processes fail. Section 4.4 presents the results
of our MPI implementation.

\subsection{4.1 Failure-Free Operation}

Figure 6 shows the average number of messages a process
sends during a broadcast. We present each tree from Sec-
tion 3.2 with synchronized checked correction and optimized

\textsuperscript{3}https://github.com/TUD-OS/flogsim
8 time steps. Gossip shows low latency, as it sends more messages and keeps significantly more processes busy during the dissemination, whereas processes relying on trees mostly send few messages before becoming silent until a correction phase. Consider an example when dissemination lasts 30 time steps; each non-leaf process in a 4-ary tree sends at most 4 messages. At the same time, gossiping processes will send $30 - t_c \gg 4$ messages during the same period, where $t_c$ is the time when a process was colored by a gossip message. Still, in a later stage of dissemination, gossip suffers from redundancy by sending messages to already colored processes. As Section 3.3 mentions, opportunistic correction may speed up coloring and a binomial tree benefits the most. Acknowledgments are a traditional approach for fault tolerance (used among others, by Buntinas [5]). They are sent along the same tree used for dissemination. A process sends an acknowledgment to the parent after receiving acknowledgments from all its children.

### 4.2 Correction Phase

Choice of a specific correction type creates a trade-off between overhead and resilience guarantees. Synchronized checked correction guarantees full coloring independent of the number of failures happening before the correction phase [17]. Optimized opportunistic correction, as shown in Section 4.1, requires less messages than checked and does not depend on precise timing.

**Lemma 2.** In the failure-free case the quiescence latency of synchronized checked correction $L_{SCC}^{FF}$ is

$$L_{SCC}^{FF} = 4o + L + \left\lfloor \frac{L}{o} \right\rfloor \cdot o$$

**Proof.** A process stops sending messages when it receives a message from each direction on the ring. If processes send the first message to the left, then the message to the right is sent after $o$ time steps. Each process receives the second message after $o + 2o + L$ time steps. A process sends the last message at time $no$, where $no \leq 3o + L < (n + 1)o$, and after additional $L+o$ time steps, the message is finally received. \qed

**Corollary 1.** In the fault-free case the number of messages sent per process in synchronized checked correction $M_{SCC}$ is

$$M_{SCC} = 3 + \left\lfloor \frac{L}{o} \right\rfloor$$

When some processes fail, live processes may stay uncolored after dissemination. Then the quiescence latency is limited by the maximum gap size $g_{max}$.

**Lemma 3.** Given the presence of failed processes and $g_{max}$ (assuming $P \gg g_{max}$), the quiescence latency of a synchronized checked correction $L_{SCC}^{FF}$ is bounded as follows

$$L_{SCC}^{FF} + g_{max} \cdot o \leq L_{SCC} \leq L_{SCC}^{FF} + (2g_{max} + 1) \cdot o$$

With failures, the total number of messages depends on the number of gaps (not size): given $g_{max}$, the more gaps there are, the fewer messages are sent. With overlapped correction, the quiescence latency depends additionally on the latency of the dissemination phase. Opportunistic correction guarantees full node coloring, if the number of faults is small enough not to disconnect the root from its children. For example, in a $k$-ary tree opportunistic correction with $d \geq k$ is guaranteed to tolerate at least up to $k - 1$ failures.

### 4.3 Resilience

In this section we explore resilience of corrected broadcast to process failures. We ran fault injection experiments in the simulator. After deciding on the fraction of failed processes (0.01%–4%), we randomly select a list of processes to be marked as “failed” and are not allowed to send messages. Other processes were running the communication protocol without raising any suspicion about the failed ones. We simulated $10^5$ broadcasts of every type on 64K processes. We picked large fault rates (≥1%), because the impact of failures at small fault rates is negligible.

**Latency** In Figure 8 we show how the quiescence latency changes as more and more processes are going down. The average latency of all Corrected Tree-based broadcast schemes degrades proportionally by 12–14% from 0.01% to 4% fault rate. With the same change of fault rates, latency of Corrected Gossip degrades only by 4%. At the same time different trees behave differently with regard to latency variation with increased fault rates: the standard deviation of the average latency is 17 times higher at 4% fault rate than at 0.01% for a binomial tree and only 10 times higher for the optimal one. The difference appears because slower trees have larger height and lower average fan-out at the same process count. This means that starting from the children of the root, non-leaf processes in slower trees are ancestors to more processes in comparison to an optimal tree. Failure of a process with more ancestors may create a higher impact on the state after dissemination.
Both mean latency and latency variation of Corrected Gossip-based broadcast change very little in comparison to tree-based broadcast, proving high stability of gossip even in the presence of failures. We observed that even for higher error rates ($\approx 10\%$), latency of gossip-based broadcast could be even less than for an optimal tree, because gossip has the freedom to increase gossiping time to compensate for a higher process failure rate. We think this scenario is less realistic, as with such a high fraction of failed processes, the latency surely is not the biggest concern.

**Messages** The real difference between tree and gossip-based broadcast appears in the number of messages each of these broadcast operations creates. Again, a small number of faults has very little impact on the number of messages (Figure 9) in comparison to the fault-free case. With more faults, the number of messages drops for all types of collectives, and the variation in messages per process grows, but Corrected Tree-based broadcast still maintains a significantly lower number of messages than Corrected Gossip. A drop in network activity is rather an unintended side effect of the lower number of colored processes after dissemination and of the fact that only processes colored during dissemination participate in correction.

**Checked Correction** For a broadcast with checked correction, dissemination time stays constant and correction time is the variable part of the overall quiescence latency. We decided to look into the correction phase of a tree-based broadcast in more detail. In Figure 10 we marked all unique pairs ($g_{\text{max}}, L_{\text{SCC}}$) we observed across two million simulations for all combinations of tree types and fault rates. Figure 10 present distribution of gap sizes observed in simulations from Section 4.3. Each point represents at least one simulation. Most large gaps happened only for binomial trees (red crosses), confirming the observation that binomial trees have a tendency to degrade more with an increased failure rate. Still, the disadvantages of binomial trees are marginal: at lower and more realistic fault rates $g_{\text{max}}$ and correction time stay low (see Table 1) for all trees. Upper and lower bounds from Section 4.2 surround the data points obtained from simulation tightly, showing that simulation and analysis agree in this aspect and the maximum gap size is an adequate proxy for correction latency. The density of points is much lower for $g_{\text{max}}$ greater than 20, because with interleaved trees, the probability of having a big $g_{\text{max}}$ is extremely low. In fact, only less than 1% of all simulations ended up having a maximum gap size greater than 10, with the vast majority of these gaps coming from 4% fault rate.

**Opportunistic Correction** With less than 1% of failed processes, Corrected Trees with opportunistic correction colored all the processes and was indistinguishable in latency and the number of messages from the failure-free case. A single unreached process appeared approximately every 10$^5$ simulations with 1% of failed processes, two with 2% failed processes and three with 4% failed processes. The total number of not fully coloring broadcast operations was around 2,600 out of two million simulations. More than 90% of the simulations with uncolored processes came at 4% failure rate leaving a single uncolored process.

### 4.4 MPI-Based Implementation

We created a prototype implementation of Corrected Trees and Corrected Gossip to run it on a petascale supercomputer. For simplicity, we implemented only optimized overlapped

---

**Figure 9.** Average number of messages goes down with higher fault rate (whiskers show 5% and 95% percentiles)

**Figure 10.** Simulations agree with analysis on the relation between $g_{\text{max}}$ and correction time

<table>
<thead>
<tr>
<th>F (%)</th>
<th>$g_{\text{max}}$ (Percentile)</th>
<th>$L_{\text{SCC}}$ (Percentile)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99% 99.9% max</td>
<td>99% 99.9% max</td>
</tr>
<tr>
<td>0.01</td>
<td>1 2 3</td>
<td>10 12 14</td>
</tr>
<tr>
<td>0.1</td>
<td>2 3 6</td>
<td>12 13 16</td>
</tr>
<tr>
<td>1</td>
<td>5 7 19</td>
<td>16 19 32</td>
</tr>
<tr>
<td>2</td>
<td>8 11 35</td>
<td>19 24 56</td>
</tr>
<tr>
<td>4</td>
<td>13 20 55</td>
<td>26 34 86</td>
</tr>
</tbody>
</table>

**Table 1.** Cost of correction under faults aggregated for all tree types (with no faults $g_{\text{max}} = 0$ and $L_{\text{SCC}} = 8$)

---

4https://github.com/TUD-OS/dying-tree
opportunistically correcting errors in a single direction. Faults were emulated as crash failures and deadlocks without noticeable differences between the two. As our prototype only features a fault-tolerant broadcast implementation at this point, we applied a wrapper that hides the "failed" processes from any other collective operation used by the benchmarks. Processes "failed" during benchmark initialization and stayed as such during the whole benchmark run.

We used the MPI broadcast benchmark from version 5.4.1 of the OSU benchmark suite [29]. This benchmark repeatedly executes MPI_Bcast and measures its runtime across all the processes of the application. To reduce the impact of system noise, we ran each test configuration 10 times. We ran the experiments at different node counts and, to not leave reserved resources idle, within several allocations. As a consequence, measurements at varying process counts were subject to different network background noise from the workloads of other users. Variation within an allocation turned out to be mostly small.

We ran the experiments on “Piz Daint”\footnote{https://www.cscs.ch/computers/piz-daint/}, a petascale supercomputer that comprises Cray XC40 racks and Cray Aries Dragonfly network. A rack has up to 384 two-socket, 18-core, hyperthreading-enabled compute nodes with 64 GiB of RAM. In total, we used up to 512 nodes with 72 MPI processes each. Because only Cray MPI, a proprietary modification of MPICH [14], was available, we implemented the prototype on top of MPI.

First, we validated our prototype implementation against the binomial broadcast implementation provided by Cray. Cray MPI uses shared memory for node local communication by default, so for a more comparable evaluation we also show line for Cray MPI without shared memory. Figure 11 shows the median of the average broadcast latency in the fault-free case; ribbons show the range between 25-th and 75-th percentiles of average latency. Indeed, our implementation shows comparable performance, especially up to 128 processes. The lower scalability is caused by the higher complexity of our code, as it is generic and also used for Lamé trees and Corrected Gossip. Corrected Gossip we implemented by fixing the number of rounds the gossip phase lasts, instead of setting a wall clock time limit. The latter would be problematic due to limited clock synchronisation precision. Each message carries the current gossip round, which gets incremented each time a message is sent. When a node receives a message with the gossip round equal to the predefined limit, it enters the correction phase. To save on expensive communication channel establishment, we set up (random) communication partners for each node once during initialization. Fixing the number of correction messages to four, we empirically selected a number of gossip rounds that resulted in the lowest latency. Nevertheless, the performance of Corrected Gossip turned out to be consistently worse than trees. We presume the reason for that is the complex and irregular communication pattern of Gossip.

For the second set of experiments (see Figure 12), we studied variations of Corrected Trees, having our implementation of the binomial broadcast without correction as a baseline. Again, we saw significant variation between different process counts\footnote{Note that this time our binomial tree implementation is almost twice as fast as in the previous figure.}, because of running in different allocations. Still, relative performance of different methods stayed consistent. In contrast to the simulations but in accordance with anecdotal experience of others we saw almost no performance improvement from Lamé trees (we experimented with various values for the parameter k). Overall, a single correction message introduced slight performance overhead and the second one added even more, but granted fault tolerance in return. For a binomial tree sending two correction messages we additionally emulated failures of 72 randomly chosen processes and saw no change in the latency.
5 Related Work

We target our algorithm at the high performance computing domain where fault tolerant communication still remains an underexplored topic. Whereas the current MPI standard does not provide fault-tolerance capabilities whatsoever, the next version is expected to. ULFM [3, 4], one of the most prominent proposals, includes fault tolerant collective operations. The standard will only specify the high-level interface/semantics though and actual implementations will require fault-tolerant algorithms. Hursey and Graham considered various collective operations in general, and evaluated a fault-tolerant broadcast that, in contrast to our work, relies on a fault detector [22].

Using unreliable communication [44] or gossip [2] for reliable broadcast was already known and even used to implement atomic broadcast [10]. Unfortunately, the guarantees provided by these protocols are often only probabilistic. Hoefler et al. proposed the use of correction to transform an unreliable hardware multicast into a reliable broadcast [21]. Corrected Gossip [17] combined correction and gossip in two strictly separate phases to build a reliable broadcast. We build on this work, replacing the non-deterministic gossip algorithm with various deterministic trees, improving the overall efficiency of the reliable broadcast. We provide an analysis and evaluation of our algorithm, assuming reliable point-to-point communication, as provided by TCP or InfiniBand [33].

Other reliable broadcast and multicast protocols rely on fault detection combined with tree restructuring [2, 5, 11, 16, 25, 30, 32, 35]. Fault detection is implemented via explicit (positive or negative) acknowledgments to the parent process. Such broadcasts achieve reliability by effectively traversing the communication tree down and up. Even in the fault-free case the tree has to be traversed twice, effectively doubling the latency in comparison to a non-resilient algorithm. In contrast, we built redundancy into the communication topology to tolerate faults pro-actively and without the need for acknowledgments.

Multi-tree approaches [7, 24] disseminate information concurrently over several trees, such that non-leaf nodes of different trees do not map to the same processes. To tolerate multiple faults, multi-trees rely on trees with higher fanout. As a consequence, optimizing the tree structure for low latency often becomes impossible.

We implemented a simulator that is in principle similar to LogGOPSIm [20]. We decided for a custom simulator because the existing one can simulate only static communication, where all messages sent and received have to be known a priori. However, dynamic communication is necessary to simulate gossip and checked correction. Additionally, LogGOPSIm does not have fault injection capabilities and misses some additional configuration parameters for its system model.

6 Conclusion

We introduced Corrected Trees as a simple, yet powerful idea that allows to build a low-latency reliable broadcast that comprises two separate phases: An unreliable dissemination phase efficiently spreads the information in the network and is followed by a reliable correction that ensures all processes are reached. Unlike similar probabilistic algorithms [17, 27], Corrected Trees feature a stable communication pattern that can be tuned to the topology of the underlying network [42].

In our analysis, simulation, and evaluation we found that Corrected Trees provide reliable low latency broadcast. Unlike other approaches, we avoid costly requirements such as the need for a failure detector or establishing (often superfluous) global knowledge of failed processes. Based on a tree numbering scheme at their core, Corrected Trees can be used to implement other collective communication primitives as well. As an additional method to reduce system noise, fault tolerant collectives are important for allowing future latency-sensitive applications to efficiently run on exascale high-performance computing systems.

Acknowledgments

The authors would like to thank the anonymous reviewers for their time and constructive comments. The work presented in this paper is supported by the German Priority Programme 1648 “Software for Exascale Computing” (SPP/EXA) via the research project FFMK (https://ffmk.tudors.org/), the German Priority Programme 912 “Highly Adaptive Energy-Efficient Computing” (HAEC), and European Research Council grant “Data-Centric Parallel Programming” (DAPP) with principal investigator T. Hoefler. Compute resources were generously provided by the Swiss National Supercomputing Centre (CSCS) and the Center for Information Services and High Performance Computing (ZIH) at TU Dresden.

References

Corrected Trees for Reliable Group Communication


A Proof for Lemma 3

Lemma 3. Given the presence of failed processes and a maximum gap size \( g_{\text{max}} \) (assuming \( P \gg g_{\text{max}} \)) the quiescence latency of a synchronized checked correction \( L_{\text{SCC}} \) is bounded as follows

\[
L_{\text{SCC}}^{\text{FF}} + g_{\text{max}} \cdot o \leq L_{\text{SCC}} \leq L_{\text{SCC}}^{\text{FF}} + (2g_{\text{max}} + 1) \cdot o
\]  

(3)

Proof. Consider a process \( c \) (possibly non-unique) sending the latest correction message (see Figure 13a). This happens when \( c \) gets messages both from the left and the right processes on the ring (\( l \) and \( r \)). The maximum latency is limited by gaps \( g_{\text{max}} \) from both sides. Assume, processes send a correction to the left first, then the correction from \( l \) reaches \( c \) last. Before sending to \( c \), \( l \) sends to \( 2o + 1 \) other processes, reaching \( c \) after \( 2o + 3o + L \) time steps. Meanwhile, \( c \) sends \( n \cdot o \) messages, such that

\[
n \cdot o \leq 2o \cdot g_{\text{max}} + 3o + L < (n + 1) \cdot o
\]

The last message from \( c \) is delivered in \( L + o \) time steps after \( n \cdot o \). The process receiving the very last message from \( c \) may receive another message from the opposite direction at the same time. This delays quiescence by additional \( o \) time steps, giving the final upper bound.

The lower bound results from the case with a single gap (see Figure 13b). Again, \( c \) is the last process to send a correction, and is stopped by \( l \) from the left. Process \( k \) reaches \( l \) after \( 3o + L \) time steps. Process \( l \) sends \( n_{ll} \) messages to the left, before receiving from \( k \), such that the receive from \( k \) ends in the interval \( (2o \cdot (n_{ll} - 1), 2o \cdot n_{ll}] \)

\[
2o \cdot (n_{ll} - 1) < 3o + L \leq 2o \cdot n_{ll}
\]

If \( n_{ll} \leq g_{\text{max}} \), \( l \) first sends messages for \( 2o \cdot n_{ll} \) time steps bidirectionally, then receives a message from \( k \) and sends in one direction from thereon. In total, \( l \) sends \( n_{ll} + g_{\text{max}} \) messages before sending to \( c \). In the meantime \( c \) can send \( n_{1} \cdot o \) messages, such that:

\[
n_{1} \cdot o \leq (n_{ll} + g_{\text{max}}) \cdot o + 2o + L < (n_{1} + 1) \cdot o
\]

\[
n_{1} = g_{\text{max}} + 3 + \left\lceil \frac{L}{o} \right\rceil + \left\lceil \frac{o + L}{2o} \right\rceil
\]  

(4)

If \( n_{ll} > g_{\text{max}} \), \( l \) sends a message to \( c \) after sending \( 2g_{\text{max}} + 1 \) messages. Then, \( c \) sends \( n_{2} \cdot o \) messages before being stopped by \( l \):

\[
n_{2} \cdot o \leq (2g_{\text{max}} + 1) \cdot o + 2o + L < (n_{2} + 1) \cdot o
\]

\[
n_{2} = g_{\text{max}} + 3 + \left\lceil \frac{L}{o} \right\rceil + g_{\text{max}}
\]  

(5)

B Coloring latency

Coloring latency (see Figure 14) limits when all processes receive the broadcast message and can leave broadcast operation. In the fault-free case all nodes are colored when the last process receives a tree message and there is no difference in coloring latency between corrected and acknowledged trees. Corrected Gossip usually finishes full coloring only during correction. For acknowledged trees, if the processes do not wait for acknowledgments, there is no guarantee that all
live nodes get colored in case of faults. When faults happen, coloring latency for acknowledged trees is limited by the timeout for expected acknowledgments and can be in the range of seconds.

**C Evaluation on “Taurus”**

In Section 4.4 we found the overall performance of our Corrected Tree-based broadcast comparable to Cray’s native version with shared memory disabled. We sought to investigate the remaining minor differences further and, more specifically, rule out additional overhead added because our implementation works on top of MPI and not inside the library. Cray MPI is a proprietary modification of MPICH [14] so it cannot be modified easily. Therefore, we re-implemented our Corrected Tree-based broadcast as an Open MPI component on the same level as the already existing binomial broadcast.

We ran this Open MPI implementation on the High Performance Computing and Storage Complex (HRSK-II, known as “Taurus”)† at TU Dresden, which consists of 18-node racks. Each node of the system has two 12-core Intel Xeon E5-2680 v3 CPUs with hyperthreading disabled, 64 GiB RAM and Mellanox InfiniBand fat-tree network. Using the non-fault-tolerant Open MPI broadcast is a baseline, we ran the benchmarks on one, two, and four full racks (see Figure 15). Our binomial-tree-based broadcast employs opportunistic correction with two or four correction messages. We additionally emulated two non-root processes failures to check the correctness implementation and saw no performance degradation. Our broadcast lagged only around by 1 µs behind the optimized Open MPI implementation.

†https://tu-dresden.de/zih/hochleistungsrechnen
D Artifact Appendix

D.1 Abstract

The provided artifact comprises the source code for the discrete-event simulator and the MPI-based prototype, the raw measurement data, and scripts to both recreate said data from scratch and to refine them into the graphs used in the paper. All software is FLOSS and publicly available but also included in the artifact for convenience. No specific hardware is required to run any of the programs. Some of the evaluation scripts may use significant amounts of free memory (several gigabytes). For replicating the MPI-based measurements from the paper, access to a (high-performance) cluster with a low-latency interconnect is necessary. We used "Piz Daint" (Cray XC40) for that purpose.

D.2 Artifact Check-List (Meta-Information)

- **Algorithm**: Two-phase, fault-tolerant broadcast based on dissemination in a tree followed by correction in a ring
- **Program**: OSU MPI benchmarks, publicly available
- **Compilation**: CMake 3.8 or above, C99 and C++17 compilers, Boost
- **Run-time environment**: MPI, optional but pre-configured in scripts: SLURM, MariaDB
- **Hardware**: any (simulations and small-scale experiments); recommended: cluster with > 10k CPUs and low-latency interconnect; scripts pre-configured for "Piz Daint"
- **Run-time state**: sensitive to compute load and network traffic from other users
- **Execution**: exclusive usage of compute nodes and network recommended to avoid interference
- **Metrics**: latency in MPI runs, many more in simulations
- **Output**: simulation: raw data + trace visualization + graphs + interactive exploration; MPI: raw data + graphs
- **Experiments**: scripts invoked according to step-by-step guide; MPI-based results are bound to vary
- **How much disk space required (approximately)**?: 3 GiB
- **How much time is needed to prepare workflow (approximately)**?: < 1 hour
- **How much time is needed to complete experiments (approximately)**?: 5 hours for full replication (given sufficient parallel compute resources)
- **Publicly available?**: Yes
- **Code/data licenses?**: GPLv3 (code), CC BY-SA 4.0 (data and documentation)
- **Archived?**: 10.5281/zenodo.1493446

D.3 Description

D.3.1 How Delivered

We provide an archive (compressed, approximately 400 MiB) that contains all sources, data and scripts with detailed descriptions as well as step-by-step instructions for reproducing the measurements and graphs from the paper. When appropriate, additional information is given on how to extend or customize the experiments. The archive comes with a Docker file to, optionally, automate the setup of all software prerequisites.

D.3.2 Hardware Dependencies

Though no specific hardware is strictly required for running any of the software, we strongly recommend to use a (high-performance) cluster to speed up the simulated fault injection experiments and to run the MPI-based implementation in large scale. For the latter, the performance of the compute nodes is of minor importance but a low-latency interconnect should be available. To fully replicate our experiments, up to 36,864 cores/hyperthreads are required for the MPI-based experiments.

D.3.3 Software Dependencies

The simulator uses C++17 and Boost, the MPI prototype is written in C99. Appropriate compilers are required. For configuration we use CMake, which supports C++17 starting from version 3.8. Evaluation and graph generation is done in R and the fault injection simulations are designed to use an instance of MariaDB for coordination. All software is FLOSS and freely available. We include our simulator, fault injection infrastructure, and MPI prototype in the artifact archive for convenience and better integration.

D.4 Installation

Download and unpack the artifact archive. Follow the detailed instructions in the Readme.md, parts of which are repeated in the following section.

D.5 Experiment Workflow

Build a docker image (here named ct)

```bash
sudo docker build -t ct .
```

and launch a Bash shell inside the container to get access to the artifact in a pre-configured environment

```bash
sudo docker run --rm -it -p 3060:3060 --entrypoint bash ct

export ROOT=$(pwd)
```

Alternatively, set up the required software (see "Requirements" in Readme.md) and save the top-level directory of the artifact

```bash
export ROOT=$(pwd)
```

This variable is used in several scripts and must be available.

---

9https://www.cscs.ch/computers/piz-daint/
8https://doi.org/10.5281/zenodo.1493446
10https://github.com/TUD-OS/flogsim
11https://github.com/TUD-OS/flogsim.fi
12https://github.com/TUD-OS/dying-tree
D.5.1 Simulator (flogsim)

The simulator implements a LogP-based model to simulate a message passing system with discrete send and receive events. We used it to study various broadcast algorithm and the architecture itself is extensible. For more documentation please refer to 

```
$ROOT/src/simulations/flogsim/README.md
```

Compile the simulator

```
cd "$ROOT/src/simulations/flogsim"
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make
export PATH="$(@dlink -f .):$PATH"
```

Optionally, you can re-run the failure-free simulation experiments

```
cd "$ROOT/scripts/simulations/fault-free"
./fault-free.sh | tee
← "$ROOT/data/simulations/fault-free.csv"
```

To interactively explore the results of the failure free experiments run

```
cd "$ROOT/plot-scripts"
./interactive.R -f
← ../data/simulations/fault-free.csv
```

and open http://127.0.0.1:3060 in your browser.

D.5.2 MPI-based Implementation (libdying)

The library implements various variations of corrected broadcasts alongside an emulation for failed MPI ranks. It is pre-loaded to an MPI program (LD_PRELOAD) and controlled via various environment variables. Please refer to

```
$ROOT/src/mpi-measurements/dying-tree/README.md
```

Compile the library

```
cd "$ROOT/src/mpi-measurements/dying-tree"
mkdir build
 cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make
```

and also compile the OSU MPI broadcast benchmark

```
cd "$ROOT/src/mpi-measurements/osu-micro-benchmarks-5.5"
./configure CC=mpicc # use MPI-aware C compiler
```

D.6 Evaluation and Expected Result

To re-generate all graphs presented in the paper, run

```
cd "$ROOT/plot-scripts"
./plot_generate.sh
```

The script comes with the paths to the respective raw data predefined and creates PDFs in the directories

```
$ROOT/plot-scripts/mpi-measurements/dying
$ROOT/plot-scripts/simulations
```

Note, that it will be hard/impossible to reproduce our exact results from the MPI runs, even if you also run them on “Piz Daint”. As mentioned in the Section 4.4, different allocations already have significant impact on the latency. The simulation results, on the other hand, are fully reproducible, because the seed used for the random fault generator is stored with the results.

D.7 Experiment Customization

Individual runs of either the simulator or the MPI-based implementation are easy to customise. Please consult their respective README.md files for more information.

Adapting a full failure injection campaign or benchmark batch requires preparation and adaptation of the scripts involved. Please refer to the (top-level) Readme.md for details and feel free to contact the authors should you experience difficulties.