We consider the following natural problem: $n$ failure-prone servers, communicating synchronously through message passing, must assign themselves one-to-one to $n$ distinct items. Existing literature suggests two possible approaches to this problem. First, model it as an instance of tight renaming in synchronous message-passing systems; for deterministic solutions, a tight bound of $\Theta(\log n)$ communication rounds is known. Second, model the scenario as an instance of randomized load-balancing, for which elegant sub-logarithmic solutions exist. However, careful examination reveals that known load-balancing schemes do not apply to our scenario, because they either do not tolerate faults or do not ensure one-to-one allocation. It is thus natural to ask if sub-logarithmic solutions exist for this apparently simple but intriguing problem.

In this paper, we combine the two approaches to provide a new randomized solution for tight renaming, which terminates in $O(\log \log n)$ communication rounds with high probability, against a strong adaptive adversary. Our solution, called Balls-into-Leaves, combines the deterministic approach with a new randomized scheme to obtain perfectly balanced allocations. The algorithm arranges the items as leaves of a tree, and participants repeatedly perform random choices among the leaves. The algorithm exchanges information in each round to split the participants into progressively smaller groups whose random choices do not conflict. We then extend the algorithm to terminate early in $O(\log \log f)$ rounds w.h.p., where $f$ is the actual number of failures. These results imply an exponential separation between deterministic and randomized algorithms for the tight renaming problem in message-passing systems.

Balls-into-Leaves: Sub-logarithmic Renaming in Synchronous Message-Passing Systems

Dan Alistarh
Microsoft Research
Cambridge, UK
dan.alistarh@microsoft.com

Oksana Denysyuk
INESC-ID,
Instituto Superior Técnico,
Universidade de Lisboa
oksana@gsd.inesc-id.pt

Luis Rodrigues
INESC-ID,
Instituto Superior Técnico,
Universidade de Lisboa
ler@ist.utl.pt

Nir Shavit
MIT & Tel-Aviv University
shanir@csail.mit.edu

Categories and Subject Descriptors
F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

General Terms
Theory, Algorithms

Keywords
Randomized algorithms; synchronous message-passing model; renaming problem.

1. INTRODUCTION

We study the following assignment problem: $n$ fault-prone servers must assign themselves one-to-one to $n$ distinct items in an efficient distributed manner. This natural problem has received considerable research attention, and is an instance of the fundamental renaming problem that has been extensively studied in both shared-memory and message-passing systems, e.g. [2, 3, 1, 5, 7, 8, 11]. In renaming, $n$ processes start with distinct identifiers taken from an unbounded original namespace, and output distinct names from a smaller target namespace. If the size of the target namespace is exactly $n$, then the problem is known as tight (or strong/perfect) renaming.

In this paper, we assume that the servers are arranged in a fully connected network and communicate by synchronous messages. Therefore, the above formulation is an instance of tight renaming in a synchronous message-passing system where $t < n$ processes may crash. Previous work by Chaudhuri, Herlihy, and Tuttle [8] gave an elegant algorithm in this setting with $O(\log n)$ round complexity, and showed this to be optimal for algorithms which distinguish state through comparisons. These results concern only deterministic algorithms, and we are interested in studying whether randomization can yield better, sub-logarithmic algorithms.

Randomization is a natural approach for renaming, and has been used to achieve low-complexity solutions in the related shared-memory model, e.g. [9]. Moreover, seen as an assignment problem, renaming is related to the extensive line of work on randomized load balancing, e.g. [1, 5, 14, 15], for which different sub-logarithmic solutions have
been proposed. Surprisingly, a careful analysis of such sub-
logarithmic solutions reveals that none of them can be used to
achieve tight renaming, since they either are designed for a
fault-free setting, or relax the one-to-one allocation re-
requirement. It is therefore tempting to ask whether random-
ization can in fact be used to obtain fault-tolerant, perfectly-
balanced allocation in sub-logarithmic time.

In this paper, we answer this question in the affirma-
tive. We present a new algorithm to solve tight renaming in
$O(\log \log n)$ communication rounds, with high probability,
exponentially faster than the optimal deterministic coun-
terpart. The algorithm, which we call Balls-into-
Leaves, is based on the idea of arranging the target names as leaves
of a binary tree; processes start at the root of the tree, and
perform repeated random choices in order to disperse them-
selves towards leaves, while minimizing contention. We also
present an early-terminating variant, which terminates in
$O(\log \log f)$ rounds, where $f$ is the actual number of failures
in the execution, which is again exponentially faster than the
best known deterministic version [3].

More precisely, our algorithm works as follows: processes
(balls) start at the root of a virtual binary tree, whose $n$
leaves correspond to the target names. In each iteration,
each ball picks a random available leaf and broadcasts its
choice; in case of collisions, a deterministic rule is used to
select a winner. The remaining balls backtrack towards the
root, stopping at the lowest node at which the ball can still find an available leaf within the corresponding subtree.
(See Figures 1 and 2 for an illustration.) Balls then
iterate this procedure, exchanging information and progres-
sevously descending in the tree.

Our main technical contribution is to show that this natu-
ral strategy is fault-tolerant, and in fact converges extremely
quickly to a perfectly balanced allocation. Our argument
has two parts: we first exploit the concentration properties
of the binomial distribution to bound the maximum con-
tention on a leaf after $\Theta(\log \log n)$ communication rounds to be $O(\text{polylog } n)$, with high probability (w.h.p). Second, we
fix a root-to-leaf path, and prove via a technical argument that all $O(\text{polylog } n)$ balls, except at most one, will dis-
perse themselves off the path within the next $O(\log \log n)$
rounds, again w.h.p. Therefore, each ball reaches a leaf within $O(\log \log n)$ rounds w.h.p.

Since the number of leaves matches the number of balls, Balls-into-Leaves solves tight renaming. Given that compar-
ison based deterministic algorithms take $\Omega(\log n)$ rounds [5], this result establishes an exponential separation between
such algorithms and their randomized counterparts. (Our
algorithm is also comparison-based.) Moreover, Balls-into-
Leaves guarantees deterministic termination; in particular,
will complete in a linear number of rounds, even in un-
ucky runs.

We then extend the Balls-into-Leaves algorithm to ensure early termination. Roughly, an early-terminating algorithm terminates faster when there are fewer failures, as its running
time is a function of the number $f$ of failures rather than $n$.
We do so by introducing an initial phase that ensures that
balls take advantage of a small number of failures in this
round, descending deeply into the tree. With this modifica-
tion, the algorithm terminates in $O(\log \log f)$ rounds w.h.p.,
where $f$ is the actual number of crashes. Furthermore, in
a fault-free execution, it terminates in constant time.

An examination of the argument (in particular, the con-
centration properties of the binomial distribution) suggests
that our complexity upper bound for Balls-into-Leaves is in
fact tight. The main open question is therefore whether the
Balls-into-Leaves strategy is optimal for this problem. We
conjecture that answering this question will require a new
lower bound technique for randomized renaming.

Roadmap. The rest of the paper is organized as follows. In Section 2 we review the related work. In Section 3 we
describe the system model and in Section 4 we define the
renaming problem. In Section 5 we describe our Balls-into-
Leaves algorithm and Section 6 analyzes its complexity. In
Section 7 we discuss an early terminating extension. We con-
clude in Section 8.

2. RELATED WORK

Renaming was first introduced in [1], for an asynchronous
message-passing system where $t < n/2$ processes may crash.
The authors show that tight renaming is impossible in this
model, and give an algorithm for $(n-t-1)$-renaming. Sub-
sequently, the problem has been extensively studied in both
message passing and shared memory. We limit the related
work discussion to the synchronous message-passing model
considered in this paper, and we direct the reader to [2, 7]
for a survey of renaming in other models.

In synchronous systems, wait-free tight renaming can be
solved using reliable broadcast [6] or consensus [12] to agree
on the set of existing ids. This approach requires linear
round complexity [10].

Tight renaming in synchronous message-passing is first
studied by Chaudhuri, Herlihy, and Tuttle [8]. They define comparison-based algorithms, which distinguish states only
through comparisons, and show such algorithms are vulner-
able to a “sandwich” failure pattern, which forces processes
to continue in indistinguishable (order-equivalent) states for
$\Omega(\log n)$ rounds. This yields an $\Omega(\log n)$-round lower bound for deterministic comparison-based algorithms, which the
authors match via an elegant wait-free algorithm [8].

Order-preserving renaming (where decided names preserve the order of their original ids) was considered by Okun [16].
The author finds a new connection between the problems of
renaming and approximate agreement, and shows that this
approach also has round complexity of $O(\log n)$. This
approach was extended by [3] to provide early termination.
The round complexity of this extension is $O(\log f)$. In-
terestingly, the authors also observed that the algorithm in [10]
terminates in constant number of rounds if the number of ac-
tual faults is bounded by $n > 2f^2$. This is because with few
faults approximate agreement can be solved in a constant
time.

The algorithms surveyed above are deterministic. On the
other hand, randomization has been employed as a tool to improve resilience in a synchronous system prone to Byz-
antine failures. The authors of [9] give a tight renaming al-
gorithm with round complexity of $O(\log n)$, which tolerates
$n-1$ Byzantine failures under an oblivious adversary. By
contrast, the present work uses randomization to improve the round complexity in a system with crash failures against
an adaptive adversary.

Tight renaming can be seen as a balls-into-bins load bal-
ancing problem, where $n$ balls must be randomly placed into
$n$ distinct bins. Early work on this problem addressed a sce-
nario in which balls are placed into bins by trying randomly


in sequential order, e.g. [11]. Since then, the problem has been extensively studied in different scenarios, e.g. [1] [5] [15]. In particular, the closest model to ours is the one where balls are placed by contacting bins in parallel, motivated by distributed load-balancing with bandwidth limitations.

Several algorithms have been proposed for this setting, e.g. [1] [14], which show that significant complexity gains can be achieved over the naive random balls-into-bins strategy. For a complete survey of existing results on parallel load-balancing, we refer the reader to [13]. To the best of our knowledge, none of the known sub-logarithmic parallel load-balancing techniques can be used to obtain wait-free tight renaming. The existing work either relaxes the exact one-ball-per-bin requirement [14], or requires balls to always have consistent views when making their choice (which cannot be guaranteed under crash faults).

3. SYSTEM MODEL

We consider the standard round-based synchronous message-passing model with a fully-connected network and \( n \) processes, where \( n \) is known a priori. Each process has a unique id, originally known only to itself. Computation proceeds in lock-step rounds. In each round, each process can send messages to its neighbors, receive a message from each neighbor, obtain a finite number of random bits, and change state.

Up to \( t < n \) processes may fail by crashing. Crashed processes stop executing and do not recover.

4. RENAMING PROBLEM

The renaming problem is defined as follows. Each process starts with initial id in some original namespace, and it must decide on a new name in some target namespace \( 1 \ldots m \), where \( m \geq n \) is a parameter of the problem, such that the following conditions must hold [4]:

- **Termination**: Each correct process eventually decides on a new name.
- **Validity**: If a correct process decides, it decides on a new name in \( 1 \ldots m \).
- **Uniqueness**: No two correct processes decide on the same new name.

When \( m = n \), the problem is called tight renaming.

5. BALLS-INTO-LEAVES ALGORITHM

The algorithm treats processes as **balls** and target names as **bins**, where each ball wants to find an exclusive bin for it. The algorithm ensures that the correct balls terminate in distinct bins.

The algorithm organizes the \( n \) bins as leaves of a binary tree of depth \( \log n \). Balls have unique labels (the processes’ initial ids), and they communicate by broadcasting messages.

The **algorithm at a high level**. Each ball starts at the root of the tree and descends the tree along a random path. As balls descend, they communicate with each other to determine if there are collisions. Collisions occur if many balls try to go to the same leaf or, more generally, if many balls try to enter a subtree without enough capacity for them. For example, if all \( n \) balls at the root tried to enter the left subtree, they would collide since the subtree has capacity for only \( n/2 \) balls. When balls collide, the algorithm assigns priorities to them based in part on their unique labels; balls with higher priorities keep descending, while the others stop. Because balls pick random paths, very few collide at higher levels, so balls quickly disperse within the tree and soon after find an exclusive leaf for them.

**Local tree, candidate path, remaining capacity.** The binary tree has \( \log n \) levels. To finish in \( O(\log \log n) \) rounds w.h.p., balls must descend many levels with each round of communication. To do so, each ball \( b_i \) keeps a local tree, containing the current position of every ball, including itself. Initially, all balls in the local tree of \( b_i \) are at the root (Fig. 1). In a single round, a ball \( b_i \) picks a random candidate path in its local tree: starting with its current position, \( b_i \) successively chooses the left or right subtree to follow for each level, until the leaf is reached. The choice between left and right subtree is weighted by the remaining capacity of each subtree (within \( b_i \)’s local tree). The remaining capacity is the number of leaves of the subtree minus the number of balls in the subtree. Say, if one subtree has no remaining capacity, \( b_i \) chooses the other with probability 1.

In this way, \( b_i \) picks the entire candidate path to the leaf locally, without communication with other balls and without regard to collisions. Ball \( b_i \) does not yet go down its path (this will happen later). Rather, \( b_i \) broadcasts its path and waits for the paths of others; this requires a round of communication.

**Collisions, priority.** Once ball \( b_i \) has received the candidate paths of other balls, \( b_i \) can calculate new positions of these balls. Ideally, a ball just follows its candidate path.

\(^*\)To simplify exposition, we assume \( n \) is a power of two.
Algorithm 1 Balls-into-Leaves Algorithm

Local Data Structures and Functions

Data Structures
- binary tree with \( n \) leaves;
- path: an ordered set of nodes of the tree;

Operations over the tree
- Remove\((b_j)\): removes \( b_j \) from the tree;
- CurrentNode\((b_j)\): current position of \( b_j \) in the tree;
- UpdateNode\((b_j, \eta)\): removes \( b_j \) from its current position and places it at node \( \eta \);
- OrderedBalls\(): returns a set of all balls in the tree, ordered by \(<_R\) (first by depth in the tree, then by ids);
- RemainingCapacity\((\eta)\): number of leaves in the subtree rooted at node \( \eta \) minus number of balls in that subtree;
- \( \eta.\text{LeftChild}() \), \( \eta.\text{RightChild}() \), \( \eta.\text{isLeaf}() \) are operation over nodes of the tree.

Additional Functions
- First\(): first element in a set;
- Next\(): iterator over a set (returns the next element in a set, advancing to the next position);
- RandomCoin\((p)\): returns heads with prob. \( p \), or tails with prob. \( 1 - p \).

Code for Ball \( b_j \)

1: Initialize: broadcast \((b_j)\);ublic
2: for all \( b_j \) received: insert \( b_j \) at the root;
3: repeat
   4: \( \eta \leftarrow \text{CurrentNode}(b_j) \);
   5: \( \text{path}_\eta \leftarrow \{ \eta \} \);
   6: while not \( \eta.\text{isLeaf}() \) do
      7: \( \text{coin} \leftarrow \text{RandomCoin}(R_{\text{RemainingCapacity}}(\eta.\text{LeftChild}()), \text{RemainingCapacity}(\eta)) \);
      8: if \( \text{coin} = \text{heads} \) then \( \eta \leftarrow \eta.\text{LeftChild}() \);
      9: else \( \eta \leftarrow \eta.\text{RightChild}() \);
   10: \( \text{path}_\eta \leftarrow \text{path}_\eta \cup \{ \eta \} \);
   11: end while
   12: broadcast \((b_j, \text{path}_\eta)\); 
     \( \triangleright \) exchange paths
   13: for all \( b_j \in \text{OrderedBalls}() \) do
     14: if \( (b_j, \text{path}_\eta) \) has been received then
        15: \( \eta \leftarrow \text{path}_\eta.\text{First}() \);
        16: while remainingCapacity\((\eta) \geq 0 \) do
           17: \( \eta \leftarrow \text{path}_\eta.\text{Next}() \);
        end while
        18: UpdateNode\((b_j, \eta) \);
       19: else
          21: Remove\((b_j) \);
       end if
   22: end for
   23: broadcast \((b_j, \text{CurrentNode}(b_j))\);
  24: for all \( b_j \in \text{OrderedBalls}() \) do
  25: if \( (b_j, \eta) \) has been received then
    26: UpdateNode\((b_j, \eta) \);
  27: else
    28: Remove\((b_j) \);
  29: end if
  30: until \( \forall b_j \in \text{OrderedBalls}() \); \( \text{CurrentNode}(b_j).\text{isLeaf}() \);

But \( b_i \) may find that different candidate paths collide: more balls may try to enter a subtree than the subtree’s remaining capacity. In this case, \( b_i \) allows balls with higher priority to proceed, while others must stop where the collision occurs (and the rest of their candidate path is discarded). The priority is determined by an order \(<_R\) where smaller balls under \(<_R\) have higher priority.

**Definition 1** (Priority Order \(<_R\)). Let \( \eta_i \) and \( \eta_j \) be the current nodes of balls \( b_i \) and \( b_j \).

\[
\begin{align*}
  b_i <_R b_j \iff \left( \text{depth}(\eta_i) > \text{depth}(\eta_j) \right) \\
  \lor \left( \text{depth}(\eta_i) = \text{depth}(\eta_j) \right) \land (b_i < b_j).
\end{align*}
\]

Under \(<_R\), balls are ordered by their depth in the tree (balls downstream ordered first), breaking ties by their unique labels. To implement these priorities, \( b_i \) iterates over all balls \( b \) in \(<_R\) order; for each \( b_i \), \( b \) follows its candidate path until \( b \) reaches a full subtree—one with no remaining capacity. If \( b \) is lucky, it ends up at a leaf in \( b_i \)’s local tree; otherwise, it stops at a higher level. Irrespective of where \( b \) stops, the algorithm ensures that there is enough space below to accommodate it. Because balls lower in the tree have a higher priority, this space cannot be displaced subsequently by balls higher in the tree. Figures 1 and 2 show the local tree before and after new positions are calculated.

Failures, synchronization, termination. A ball may crash while broadcasting its candidate path: some balls may receive this broadcast, while others do not. The result is that the local trees of the surviving balls may diverge. To resynchronize, after \( b_i \) has updated its local tree using the candidate path, \( b_i \) broadcasts its current position, and waits to learn the positions of other balls; this requires another round of communication. Based on the received information, if necessary \( b_i \) corrects the positions of balls in its local tree; if \( b_i \) does not hear from a ball in its local tree, \( b_i \) removes it (the ball has crashed). If \( b_i \) finds that every ball is at a leaf, it terminates. Otherwise, it repeats the entire procedure.

**Detailed pseudocode.** Algorithm 1 gives the detailed pseudocode. Initially, each ball \( b_i \) broadcasts its label, receives labels from other balls, and inserts them at the root of its local tree (Lines 1–2). Then, \( b_i \) repeatedly executes the main loop (Lines 3–30); each iteration is called a phase, and each phase has two communication rounds. In round one, \( b_i \) first chooses its candidate path (Lines 6–11) one edge at a time, where the probability of each child is the ratio of remaining slots in either subtree (Line 7). Then \( b_i \) broadcasts its candidate path (Line 12). After receiving the candidate paths of others, \( b_i \) iterates over the balls \( b \) in \(<_R\) order, to compute their new positions. Each ball \( b \) moves down its path as long as the subtree has remaining capacity (Lines 13–19). Balls that do not announce their paths have crashed and are removed from the local tree (Lines 20–21). In the second round (Lines 23–29), \( b_i \) sends its position, receives positions of other balls, and updates its local tree, again removing balls which fail to send their positions. If all balls in the local tree have reached leaves, \( b_i \) terminates. It is easy to change the algorithm to allow a ball to terminate as soon as it reaches a leaf. Such modification requires additional checks that have been left out in favor of clarity.

5.1 Tight Renaming using Balls-into-Leaves

We now prove that the Balls-into-Leaves algorithm solves tight renaming in \( O(\log \log n) \) communication rounds w.h.p.
**Theorem 1.** In the Balls-into-Leaves algorithm, correct balls terminate at distinct leaves.

The proofs of correctness are intuitive and have been moved to Appendix A.

### 6. COMPLEXITY ANALYSIS

In this section, we prove that Balls-into-Leaves terminates in $O(\log \log n)$ rounds with high probability.

For clarity, we first consider a failure-free execution. We then show that faults do not slow down the progress of the algorithm. (Intuitively, collisions are less likely as the number of surviving balls decreases.)

Without crashes, local views of the tree are always identical, and we therefore focus on one local view. The analysis consists of two parts. In the first part, we show that, after $O(\log \log n)$ phases, the number of balls at each node decreases to $O(\log^2 n)$. In the second part, we consider an arbitrary path from the root to the parent of a leaf. We use the fact that there are $O(\log^2 n)$ balls at each node, and show that the path becomes completely empty within the next $O(\log \log n)$ rounds, with high probability. By a union bound over all such paths, we obtain that the tree is empty w.h.p.

We first prove a useful invariant of Balls-into-Leaves algorithm that we call *Path Isolation Property.* Informally stated, this says that no new balls appear on any path from the root.

**Lemma 2.** *(Path Isolation Property).* For any phase $\phi \geq 1$ and path $\pi$ from the root, the set of balls on $\pi$ in $\phi$ is a superset of balls on $\pi$ in $\phi + 1$.

**Proof.** By construction, balls can only move down the tree. Fix some node $\eta$ on path $\pi$. The only way new ball $b_i$ located at some node $\mu$ in phase $\phi$, reaches $\eta$ in phase $\phi + 1$ is by constructing in $\phi$ a path from $\mu$ that contains $\eta$. Thus, by construction of a binary tree, $\mu$ is on $\pi$. \(\Box\)

In our proofs, we use some notation and facts from theory of probability, shown in Figure 3.

#### 6.1 Part 1 - Bounding the Number of Balls at a Node

We now give a lower bound on how fast the number of balls at each node decreases. Consider some node $\eta$, and let $\eta M$ and $(1-p)M$ be the remaining capacities of its left and right subtrees, respectively, for some integer $M$ and $0 \leq p \leq 1$ in phase $\phi$. By construction, at most $M$ balls reach $\eta$. The balls that get stuck at $\eta$ in phase $\phi + 1$ are those which had $\eta$ on their paths in $\phi$, but did not fit in a subtree below $\eta$. By Lemma 2 these balls have enough space in the sibling subtree of $\eta$. We use the notation $\text{balls}(\eta, \phi)$ to denote the number of balls at node $\eta$ in phase $\phi$. We denote by $\text{bmax}(\phi)$ the most populated node in phase $\phi$.

**Lemma 3.** For some node $\eta$, let $\eta M$ and $(1-p)M$ denote remaining capacities of its left and right subtrees respectively, in some phase $\phi$. If $M' \leq M$ balls choose between the left and right subtrees of $\eta$ in $\phi$, then, for any $x > 0$, $\Pr(\text{balls}(\eta, \phi + 1) > x) \leq \Pr(\frac{M'}{2} - B(M, \frac{1}{2}) > x)$.

**Proof.** Choosing between the left and right subtrees of $\eta$ can be seen as choosing between two bins with capacities $pM$ and $(1-p)M$. Each of the $M'$ balls chooses independently between the two bins with probabilities $p$ and $(1-p)$ respectively. If there is space in the chosen bin, then the ball is accepted; if the bin is full, then the ball is rejected. Since $M' \leq M$, there are three cases: either both bins are filled (perfect split), no bin has been filled, or exactly one bin has been filled, and there is some overflow. Note that, in the first two cases, $\eta$ is empty in phase $\phi + 1$.

If one bin has filled, then, w.l.o.g., assume it is the left bin. Let $Y$ be a random variable that counts the number of balls that have chosen left. Clearly, $Y \sim B(M', p)$. Then, the number of rejected balls is $Y - Mp$. Since $M \geq M' p = E[Y]$, then $Y - Mp \leq Y - E[Y]$. From Fact 2 we obtain that,

$$\Pr(Y - E[Y] > x) \leq \Pr(\frac{M'}{2} - B(M, \frac{1}{2}) > x).$$

Let us now consider what happens after the first phase.

**Lemma 4.** Let $i$ be the depth of node $\eta$. Then, for some constant $c > 0$, we have that

$$\Pr\left(\text{balls}(\eta, 2) > c\left(\frac{\sqrt{n}}{\log \log n}\right)^{\frac{1}{2}}\right) < \frac{1}{n}.$$
The analysis of the next phases is more involved, since we do not know the exact remaining capacities of each subtree. Therefore, we consider the worst case scenario by assuming that any node \( \eta \) has enough capacity to accommodate all balls on the path from the root to \( \eta \).

**Lemma 5.** For any \( \phi > 1 \), node \( \eta \), and some const. \( c > 0 \),
\[
\Pr \left( \text{balls}(\eta, \phi+1) > c(b_{\max}(\phi))^{\frac{1}{2}} \log n \right) < \frac{1}{n^c}.
\]

**Proof.** By the path isolation property (Lemma 2), the only balls that may attempt to choose between subtrees of \( \eta \) are those on the path from the root to \( \eta \). Let \( i \) be the depth of \( \eta \). The total number of balls on path \( i \) is at most \( i \times b_{\max}(\phi) \). By Fact 4, the probability to have more rejected balls is the highest if we inflate the remaining capacity of the subtrees of \( \eta \) to \( i \times b_{\max}(\phi) \).

Thus, by Lemma 3, the probability that at least \( x \) balls get stuck at \( \eta \) can be bounded as follows,
\[
\Pr (\text{balls}(\eta, \phi+1) > x) \leq \Pr \left( \frac{i \times b_{\max}(\phi)}{2} - B(i \times b_{\max}(\phi), \frac{1}{2}) > c(i \times b_{\max}(\phi) \log n) \right).
\]

By the Chernoff Bound (Fact 3),
\[
\Pr \left( \frac{i \times b_{\max}(\phi)}{2} - B(i \times b_{\max}(\phi), \frac{1}{2}) > c(i \times b_{\max}(\phi) \log n) \right) < \frac{1}{n^c}.
\]

Since \( i \leq \log n \), the claim follows.

**Lemma 6.** For some const. \( c' > 0 \), after \( O(\log \log n) \) phases, \( \Pr (\text{balls}(\eta, \phi) > O(1) \log^2 n) < \frac{1}{n^c} \).

**Proof.** Fix some constant \( c > 0 \). By Lemma 3,
\[
\Pr \left( \text{balls}(\eta, 2) > c(n \log n)^{\frac{1}{2}} \right) < \frac{1}{n^c}.
\]

By Lemma 4, \( \Pr \left( \text{balls}(\eta, \phi+1) > c(b_{\max}(\phi))^{\frac{1}{2}} \log n \right) < \frac{1}{n^c} \).

Since \( n \frac{1}{\gamma \log \log n} = O(1) \), we pick some constant \( \gamma \) such that \( n \frac{1}{\gamma \log \log n} \approx 1 \).

Let \( f(x) = x^c \log n \). Taking \( x = c(n \log n)^{\frac{1}{2}} \),
\[
f(\gamma \log \log n) (x) = c^2 \log^2 n.
\]

Applying Lemma 3 and Lemma 6 iteratively for \( \log \log n \) phases, we obtain that
\[
\Pr (b_{\max}(\eta, \gamma \log \log n) > c^2 \log^2 n) < \frac{\gamma \log \log c}{n^\epsilon}.
\]

Therefore, there exists some small constant \( \epsilon < 1 \), such that \( \frac{\log \log n}{n^\epsilon} \), and the claim follows.

### 6.2 Part 2 - Bounding the Number of Balls on a Path

Lemma 4 shows that, after \( O(\log \log n) \) phases, the number of balls at each node is at most \( O(\log^2 n) \) w.h.p., which means that there are at most \( O(\log^3 n) \) balls on each path w.h.p. In the following, we complete the argument by showing that all inner nodes of the tree are empty after another \( O(\log \log n) \) phases.

To show this, instead of looking at nodes, we focus on paths from the root to a parent of a leaf (there are \( n/2 \) such paths). By the path isolation property (Lemma 2), new balls never appear on a path. (In other words, new balls arriving at a node can only come from nodes higher on the same path.) We show that at least a constant fraction of balls escapes from each path once every two phases. Intuitively, this analysis captures how fast balls disperse within the tree.

Formally, let us fix a phase \( \phi \) and a path \( \pi \) from the root to a parent of a leaf. Let \( \eta_1, \eta_2, \ldots, \eta_{\log n} \) be the nodes on \( \pi \), ordered by depth. A *gateway node* (or simply a gateway) is a child of \( \eta \), that is not on \( \pi \). For uniformity, we combine both children of the last node on \( \pi \) (tree leaves) into one gateway meta-child. For instance, in the sample configuration from Figure 4(b), consider the rightmost path (highlighted in Figure 4), all the left children of nodes on \( \pi \) are gateways.

By construction, the sum of remaining capacities of all gateway subtrees (corresponding to empty leaves reachable from \( \pi \)) is equal to the total number of balls on \( \pi \). In phase...
\( \phi \), all balls on the path propose paths going through these gateways.

We now show that, if ball \( b_i \) is among the highest priority balls that have chosen the same gateway, then \( b_i \) will escape from path \( \pi \) either in phase \( \phi \) or \( \phi + 1 \).

**Lemma 7.** Consider node \( \eta_i \) on \( \pi \) and let \( c_i \) be the remaining capacity of its gateway subtree. If \( m \) balls choose that subtree in \( \phi \), then at least \( \min(m, c_i) \) balls escape in \( \phi \) and \( \phi + 1 \).

Proof. Let ball \( b_k \) be among \( \min(m, c_i) \) highest priority balls that have chosen the gateway at \( \eta_i \). Then, \( b_k \) is in one of the following scenarios.

**Case 1:** \( b_k \) attempts to move down towards \( \eta_i \), and stops at some node \( \eta_j \) above \( \eta_i \). This happens because the subtree down on \( \pi \) has exceeded its capacity. In this case, in \( \phi + 1 \), \( b_k \) tries the gateway subtree at \( \eta_j \) and, by Lemma 1, that subtree has enough space to accommodate \( b_k \).

**Case 2:** \( b_k \) reaches \( \eta_i \). By assumption, ball \( b_k \) is among \( \min(m, c_i) \) highest priority balls that have chosen the same gateway. Thus, \( b_k \) escapes in \( \phi \) into the gateway subtree of node \( \eta_i \).

If \( b_k \) is in Case 1, it escapes path \( \pi \) in phase \( \phi + 1 \). If \( b_k \) is in Case 2, it escapes \( \pi \) in phase \( \phi \). There are at least \( \min(m, c_i) \) such balls, and the claim follows. \( \square \)

We now bound the probabilities with which balls try each gateway on \( \pi \).

**Lemma 8.** Let \( M_i = \sum_{1 \leq j \leq i} \text{balls}(\eta_j, \phi) \) be the number of balls on the subpath of \( \pi \) from the root to \( \eta_j \). If \( c_i \) is the remaining capacity of the gateway subtree of \( \eta_i \), then \( M_i \) balls try this gateway in \( \phi \) with probability at least \( c_i / M_i \).

Proof. (Sketch) Recall that balls construct paths trying between the children of each node with probabilities indexed by their remaining capacities (Lines 6–10 of Algorithm 1).

Intuitively, each ball competes on a subtree with every other ball that can reach the same subtree. And, by construction, the remaining capacity of all subtrees reachable from some path from the root is equal to the number of balls on this path. Note that the total remaining capacity of the subpath from the root to \( \eta_i \) (i.e., all gateway subtrees and the non-gateway subtree at \( \eta_i \)) is \( M_i \). Thus, every ball tries the gateway at \( \eta_i \) with prob. at least \( c_i / M_i \). \( \square \)

In the following, we show that at least a constant fraction of balls escape \( \pi \) in every two phases.

**Lemma 9.** Let \( M = \sum_{1 \leq i \leq n} \text{balls}(\pi, \phi) \) be the total number of balls on \( \pi \) in phase \( \phi \). For some const. \( c > 1 \), less than \( M / c \) balls have escaped \( \pi \) after phase \( \phi + 1 \) with probability \( e^{-\frac{M}{cn}} \).

Proof. By Lemma 2, \( M \) balls try the gateway at \( \eta_i \) with prob. at least \( c_i / M_i \). By Lemma 4, for any \( \gamma > 1 \), \( \gamma \leq c_i \) highest priority balls that choose the gateway at \( \eta_i \), escape \( \pi \) in \( \phi \) or \( \phi + 1 \). Define as success the event that some ball chooses the gateway at \( \eta_i \). By Lemma 3, such an event occurs with prob. at least \( c_i / M_i \) among \( M \) tries. Thus, the number of successes follows \( B(M_i, c_i / M_i) \). From the Chernoff Bound (Fact 3),

\[
\Pr \left( B(M_i, c_i / M_i) < c_i - \frac{\gamma M_i}{2} \right) < e^{-\frac{\gamma^2 M_i}{8}}.
\]

Recall that \( \sum_{1 \leq i \leq n} c_i = M \). Considering all \( \eta_i \), \( 1 \leq i \leq \log n \) on \( \pi \), the sum of successes is less than \( M - \frac{M}{2} \) with prob. \( < e^{-\frac{\gamma^2 M}{8}} \). Since \( \gamma \) is arbitrary, the claim follows. \( \square \)

From the above lemma, it follows that all \( M \) balls on \( \pi \) escape the path within \( O(\log M) \) phases, with probability at least \( 1 - (1/e)^{O(M)} \).

**Lemma 10.** Consider a path \( \pi \) containing \( M \) balls. After \( O(\log M) \) phases, the probability that \( \pi \) remains non-empty is less than \( e^{-\frac{M}{c}} \), for some constant \( c > 0 \).

Proof. Fix some constant \( c \). By Lemma 2 at least \( M / c \) balls have escaped from \( \pi \) after phase \( \phi + 1 \), with probability at least \( 1 - e^{-\frac{M}{c}} \). Starting with \( M \) balls, and iterating in Lemma 2 over the remaining balls for \( 2c \log M \) phases, we obtain that the probability that \( \pi \) is not empty after \( 2c \log M \) phases is at most \( 2c \log M e^{-\frac{M}{2c}} < e^{-\frac{M}{3c}} \) for some small constant \( c < 1 \). \( \square \)

Finally, we combine the two parts of the proof in the following theorem.

**Theorem 2.** Balls-into-Leaves terminates in \( O(\log \log n) \) rounds with probability at least \( (1 - \frac{1}{n}) \), where \( c > 0 \) is a constant.

Proof. From Lemma 3 after \( O(\log \log n) \) phases, the probability there is a path with more than \( O(\log^4 n) \) balls, is less than \( \frac{1}{n^2} \), for some const. \( c' > 0 \). Taking \( M = O(\log^4 n) \) balls on path \( \pi \) from the root to a parent of some leaf, by Lemma 4 and a union bound over all such \( n/2 \) paths, the probability that some path that is not empty is less than \( \frac{1}{n^{2c'}} \), for some constant \( c' > 0 \). Putting together the above results, we get the probability that the tree still has a populated inner node is at most \( \frac{1}{n^{2c'}} \) where \( c', c'' \) are constants. By construction, the algorithm terminates when all balls have reached leaves. Choosing some constant \( c > c' + c'' \), the algorithm terminates in \( O(\log \log n) \) phases with probability at least \( 1 - \frac{1}{n} \).

Since each phase consists of 2 rounds, the claim follows. \( \square \)

### 6.3 Crashes

To show that crashes do not slow down termination, we continue the analysis of some path \( \pi \) that starts at the root. By construction, at the end of each phase, in every local view, the position of each surviving ball \( b_i \) is updated according to \( b_i \)’s local view. We thus focus on the progress of \( b_i \) in its local view.

Iterating on each phase \( \phi \) that contains at least one failure, we compare the local view \( V \) of ball \( b_i \) in \( \phi \) with its hypothetical view \( V' \) in an execution with a failure-free \( \phi \). First, note that views \( V \) and \( V' \) are equivalent if \( b_i \) has not seen a failure. Now, assume \( b_i \) has seen a failure in \( V \).

We show that \( b_i \) is at least as likely to escape from \( \pi \) in \( V \), as it is in \( V' \). First, we note that if \( b_i \) has seen a failure in some disjoint subtree, or the crashed ball had lower priority than \( b_i \), then, by construction, such a failure does not affect the progress of \( b_i \). Consider now a failure that occurs in a subtree of \( b_i \). Such a failure implies that, in \( V \), the total capacity of all gateways on \( \pi \) is larger than the number of balls on \( \pi \). Thus, \( b_i \) is at least as likely to be among the first
highest priority balls that have chosen the same gateway in \( \phi \). Since the choice of \( b_i \) and \( \pi \) is arbitrary, the argument applies to every ball in every view.

6.4 Experimental Evaluation

We complement with simulations the analytical analysis of the round complexity of Balls-into-Leaves. The results are depicted in Table 1 (the number of rounds was obtained by averaging over 200 failure-free runs of the algorithm).

<table>
<thead>
<tr>
<th>Number of balls</th>
<th>Number of rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^2 )</td>
<td>2.76</td>
</tr>
<tr>
<td>( 2^{10} )</td>
<td>4.00</td>
</tr>
<tr>
<td>( 2^{15} )</td>
<td>4.92</td>
</tr>
<tr>
<td>( 2^{20} )</td>
<td>5.24</td>
</tr>
<tr>
<td>( 2^{25} )</td>
<td>6.00</td>
</tr>
</tbody>
</table>

Table 1: Experimental evaluation of Balls-into-Bins

Our simulations confirm that the running time grows very slowly with \( n \): e.g., for \( 2^{25} \) balls the algorithm takes only 6 rounds to terminate. These results are consistent with our \( O(\log \log n) \) bound. Moreover, the simulations show that the hidden constants are small.

7. EARLY TERMINATING EXTENSION

We now extend the algorithm to terminate faster in executions with fewer crashes. Without failures, balls can use their unique labels to pick distinct leaves in one round: balls exchange their labels, and each ball chooses a leaf indexed by the rank of its label in the ordered set of all labels. But collisions may occur due to failures. A single crash can cause up to \( n/2 \) collisions: the ball with the lowest label \( b_{\min} \) sends to every second ball (by label order) and then crashes, so that all other balls collide in pairs.

On the other hand, if balls use this scheme to deterministically pick paths in the Balls-into-Leaves algorithm, it is easy to see that the paths are well distributed; in the first phase, the tree collapses into small subtrees of depth 2 in every local tree. But the balls cannot use deterministic paths in every phase, otherwise the algorithm’s round complexity would be no better than \( \Omega(\log \log n) \), due to the lower bound of \( \Omega(1) \).

We combine the deterministic and randomized approaches, by first deterministically collapsing the tree into disjoint subtrees of depth \( O(\log f) \) (where \( f \) is a number of failures that have occurred in an execution), and then resorting to randomization. The modified Balls-into-Leaves algorithm works as follows. In Round 1 of phase 1, replace Lines 6–11 in Algorithm 1 with the following: ball \( b_i \) constructs path deterministically towards the leaf ranked by \( b_i \) in Ordered-Balls(); the rest of phase 1 is executed as in the original algorithm. In the remaining phases, \( b_i \) executes the code of the original algorithm.

It is easy to see that, in a failure-free execution, the modified algorithm terminates in one phase.

**Theorem 3.** In a failure-free execution, the modified Balls-into-Leaves algorithm terminates in \( O(1) \) rounds.

In an execution with \( f \) failures, we show that the algorithm terminates in \( O(\log \log f) \) rounds w.h.p. (the proof can be found in Appendix B).

**Theorem 4.** The modified Balls-into-Leaves algorithm terminates in \( O(\log \log f) \) rounds with prob. at least \( (1 - \frac{1}{c}) \), where \( c > 0 \) is a constant.

8. DISCUSSION

Extending the classical balls-into-bins technique, we have proposed Balls-into-Leaves, a randomized algorithm that places \( n \) balls into \( n \) leaves of a binary tree in \( O(\log \log n) \) rounds w.h.p. Our simulations confirm that, in fact, the algorithm terminates in few rounds even for very large values of \( n \). However, we do not optimize the algorithm for the message complexity—in each round Balls-into-Leaves employs all-to-all communication. Therefore, the total message complexity is \( O(n^2 \log \log n) \). It is possible to reduce it to \( O(n^2) \) by eliminating communication between balls in disjoint subtrees. However, to overcome the \( O(n^2) \) barrier (\( O(n) \) messages per ball) would require a different approach.

We have also presented an extension of the algorithm that provides early termination in \( O(\log \log f) \) rounds w.h.p. when there are \( f \) failures, and deterministic termination in \( O(1) \) rounds in failure-free executions.

Our results imply an exponential separation between deterministic and randomized algorithms for tight renaming. An open question is whether the Balls-into-Leaves algorithm is optimal for this problem. Answering this question requires new lower bounds for randomized renaming. We conjecture that obtaining such lower bounds will be challenging, given that lower bounds for other variants of renaming have required subtle topological or reduction techniques.

Acknowledgements

Dan Alistarh was partially supported by the SNF Postdoctoral Fellows Program, NSF grant CCF-1217921, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

Oksana Denysyuk and Luís Rodrigues were partially supported by Fundação para a Ciência e Tecnologia (FCT) via the project PEPIITA (PTDC/EEI-SCR/2776/2012) and via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013.

Nir Shavit was supported in part by NSF grants CCF-1217921 and CCF-1301926, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

9. REFERENCES


We now present the omitted proofs.

A. CORRECTNESS
We prove that in the Balls-into-Leaves algorithm correct balls terminate at distinct leaves.

We first notice that, at the beginning of each phase, the positions of all correct balls are synchronized across the views. Positions are considered at the beginning of a phase.

**Proposition 1.** For any phase $\phi \geq 1$, if balls $b_i$, and $b_j$ are correct, then the tree position of $b_i$ in the view of ball $b_j$ is the same as the position of $b_i$ in the view of ball $b_j$.

**Proof.** We proceed by induction on the phase index. At the beginning of phase 1, this holds since all $n$ correct balls have broadcasted their labels and have been placed at the root in local views of all correct balls (Lines 1–2).

For the induction step, assume by contradiction that the claim holds in $\phi \geq 1$, but not in $\phi + 1$. This is only possible if $b_j$ has not sent its position to $b_i$ in $\phi$ (Line 23 of Algorithm 1). However, since $b_i$ is correct, $b_j$ must have executed Line 23 in phase $\phi$. Contradiction.

This implies the following.

**Proposition 2.** For any phase $\phi \geq 1$ and any local view, the number of correct balls in each subtree is $\leq$ the total number of balls in that subtree.

The priority order $<_R$ ensures that the descent of correct balls is simulated consistently across views.

**Proposition 3.** For any phase $\phi \geq 1$, if $b_i$ and $b_j$ are correct, then either $b_i <_R b_j$ in every view, or $b_i >_R b_j$ in every view.

**Proof.** By Proposition 1 the positions of correct balls are synchronized across the views. Assume, w.l.o.g., $b_i < b_j$. Thus, by definition of order $<_R$, $b_i < b_j$ in each view.

Below we restate and prove Lemma 1:

**Lemma 1.** For any phase $\phi \geq 1$, in any local view, the number of correct balls in each subtree is $\leq$ the number of leaves in that subtree.

**Proof.** We prove the claim by induction over the phase index $\phi$. For $\phi = 1$, the claim holds since all $n$ balls are at the root of the tree.

Assume the claim is true for $\phi \geq 1$. Since each subtree contains at least all correct balls (Proposition 2), and (Proposition 3), balls simulate the descent of correct balls in a consistent order (if $b_j <_R b_i$, then $b_j$ is moved before $b_i$), when $b_i$ simulates the descent of each ball locally, every ball (including $b_i$) stops in a subtree where it still fits among at least all correct balls.

We now prove Theorem 1:

**Proof of Theorem 1.** By Proposition 1 and Lemma 1, after correct ball $b_i$ reaches a leaf and announces its position, all correct balls have $b_i$ at the same leaf in their local views and thus never propose a path to that leaf. By algorithmic construction, balls do not move once they have reached the bottom.

Finite Deterministic Termination

We now prove the algorithm terminates always in a finite number of rounds.

**Lemma 11.** If no failures occur in some phase $\phi$, then at least one ball at an inner node reaches a leaf in its local view of the tree.
Proof. Let \( b_j \) be the highest priority correct ball among the balls at inner nodes (not leaves) at the beginning of \( \phi \).

By algorithmic construction, \( b_i \) chooses in Round 1 a path to an empty leaf in its local view of the tree. Since by assumption no crashes occur, balls receive identical sets of paths and move balls down in the same order. By algorithmic construction, balls at the leaves are not moved. So, \( b_j \) is the first to move down in its own view and thus it will reach the leaf chosen by its path. \( \square \)

By Lemma \ref{prop:balls-into-leaves-faultless} balls require at most \( n \) fault-free phases to reach the bottom; since there are at most \( t < n \) faults in total, the algorithm terminates in \( O(n) \) phases deterministically.

**B. EARLY TERMINATION**

In the following, we restate and prove Theorem \ref{thm:balls-into-leaves-binary-faults}.

**Theorem \ref{thm:balls-into-leaves-binary-faults}** The modified Balls-into-Leaves algorithm terminates in \( O(\log \log f) \) rounds with prob. at least \( 1 - \frac{1}{n^c} \), where \( c > 0 \) is a constant.

Proof. Let \( i \) be rank of ball \( b_i \) among the surviving balls. Assume \( b_i \) has not seen \( k \leq f \) failures. The rank in its local view has shifted right by at most \( k \) with regard to other views. Since all survivors have each other, including \( b_i \), in their local views, at most \( f - 1 \) other survivors see their ranks in the interval \( i..(i + f) \).

Consider binary representation of leaf ranks. We note that each subtree that contains some leaf is indexed by the binary prefix of the leaf rank. From the previous discussion, the surviving balls collide on at most \( \lceil \log f \rceil \) least significant bits. Thus, in every local view, collisions occur at the depth at least \( \log n - \lceil \log f \rceil \) in phase 1.

In the subsequent phases, balls in disjoint subtrees propose non-overlapping paths. Therefore, the rest of the execution is equivalent to running at most \( \frac{n}{\lceil \log f \rceil} \leq n \) parallel instances of Balls into Leaves with at most \( f \) balls each. From Theorem \ref{thm:balls-into-leaves-binary-faults} and for a sufficiently large const. \( c > 0 \), the claim follows. \( \square \)