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

Dan Alistarh[1], Oksana Denysyuk[†2], Luis Rodrigues[2], and Nir Shavit[3]

[1]Microsoft Research Cambridge
[2]INESC-ID / Instituto Superior Tecnico
[3]Tel-Aviv University and MIT

Abstract

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 problem as a randomized load-balancing problem, which have elegant sub-logarithmic solutions. However, careful examination reveals that such solutions do not really apply to our scenario, because they are not fault tolerant 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. An extension of the algorithm terminates 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.

[*]Regular submission. Student paper.
[†]E-mail: oksana.denysyuk@ist.utl.pt. Address: INESC-ID, R. Alves Redol, 9, Lisbon, Portugal. Tel.: +351 213 100 359.

1 Introduction

In this paper, we consider the following assignment problem: n fault-prone servers, communicating through synchronous messages, 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. 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, and has been extensively studied both in shared-memory and message-passing systems, e.g. [4, 7, 2, 9, 19, 3].

In particular, 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 [9] 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 shared-memory setting, e.g. [2]. Moreover, seen as an assignment problem, renaming is related to the extensive line of work on randomized load balancing, e.g. [18, 1, 5, 17]. Surprisingly, a careful analysis of existing load balancing techniques reveals that none of them can be used to achieve sub-logarithmic tight renaming, since they either are designed for a fault-free setting, or relax the one-to-one allocation requirement. It is therefore tempting to ask whether randomization can in fact be used to obtain a fault-tolerant, perfectly-balanced allocation in sub-logarithmic time.

In this paper, we answer this question in the affirmative. We present a new algorithm to solve tight renaming in $O(\log\log n)$ communication rounds, exponentially faster than the optimal deterministic counterpart. The algorithm, which we call *Balls-into-Leaves*, is based on the idea of arranging the target names as leaves in a binary tree; processes start at the root of the tree, and perform repeated random choices in order to disperse themselves 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. (Please see Figures 1 and 2 for an illustration.) Balls then iterate this procedure, exchanging information and progressively descending in the tree.

Our main technical contribution is showing that this natural 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 contention on a name 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 one) will disperse 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 comparison-based deterministic algorithms take $\Omega(\log n)$ rounds, 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, it will complete in a linear number of rounds, even in unlucky runs.

We extend the Balls-into-Leaves algorithm to ensure *early termination*. Roughly speaking, an early-terminating algorithm terminates faster when there are fewer failures, so its running time becomes 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 modification, 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 concentration properties of the binomial distribution) suggests that our complexity upper bound for Balls-into-Leaves is in fact tight. However, proving tightness remains an open problem that requires new lower bounds on randomized renaming. Given our own attempts, we conjecture that obtaining such bounds will be challenging, as lower bounds for other variants of renaming have required subtle topological or reduction techniques, e.g. [14, 8, 2, 12].

Due to space limitations, some proofs in this paper are deferred to an optional appendix.

2 Related Work

Renaming is introduced in [4] 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. Subsequently, 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 [7, 2] for a survey of renaming in other models.

In synchronous systems, wait-free tight renaming can be solved using reliable broadcast [6] or consensus [15] to agree on the set of existing ids. This approach requires linear round complexity [11].

Tight renaming in synchronous message-passing is first studied by Chaudhuri, Herlihy, and Tuttle [9]. They define *comparison-based* algorithms, which distinguish states only through comparisons, and show such algorithms are vulnerable 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 [9].

Order-preserving renaming (where decided names preserve the order of their original ids) was considered by Okun [19]. 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 algorithm is not *comparison-based*. This approach was extended by [3] to provide *early termination*. The round complexity of this extension is $O(\log f)$. Interestingly, the authors also observed that the algorithm in [19] terminates in constant number of rounds if the number of actual 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 Byzantine failures [10]. The authors give a tight renaming algorithm 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-balancing problem, where n balls must be randomly placed into n distinct bins. Early work on this problem addressed a scenario in which balls are placed into bins by trying randomly in sequential order, e.g. [13]. Since then, the problem has been extensively studied in different scenarios, e.g. [18, 1, 5]. 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, 17], 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 [16]. To the best of our knowledge, none of the known parallel load-balancing techniques can be used to obtain sub-logarithmic wait-free tight renaming. Existing work either relaxes the exact one-ball-per-bin requirement [17], or requires balls to always have consistent views when making their choice (which cannot be guaranteed under crash faults).

3 System Model and Problem Definition

Model. We consider a 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.

Renaming. 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* $1...m$, 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 $1...m$.
- *Uniqueness:* No two correct processes decide on the same new name.

When $m = n$, the problem is called *tight* renaming.

4 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 organizes the n bins as leaves of a binary tree of depth $\log n$[1]. Balls have unique labels (the processes' initial ids), and they can 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 label; balls with higher priorities keep descending, while the others stop. Because balls pick random paths, very few collide at higher levels, so balls quickly descend the tree and soon after find an exclusive leaf for them.

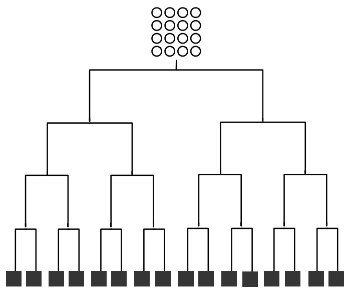
Local tree, candidate path, remaining capacity. The binary tree has $\log n$ levels. To finish in $O(\log \log n)$ rounds w.h.p., balls must descend many levels with a single round of communication. To do so, each ball b_i keeps a *local tree*, containing the current position of each ball, including 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 leafs 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.

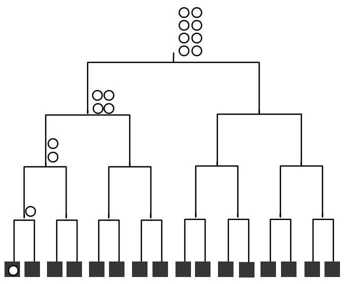
Collisions, priority. Once ball b_i has received the candidate paths of other balls, b_i can calculate new positions of these balls. Ideally, a ball just follows its candidate path. But b_i may find that candidate paths collide: more balls may try to enter a subtree than the subtree's remaining capacity. 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 η_i and η_j be the current nodes of balls b_i and b_j. Then,*

$$b_i <_R b_j \iff (depth(\eta_i) > depth(\eta_j)) \vee ((depth(\eta_i) = depth(\eta_i)) \wedge (b_i < b_j)) .$$

[1]To simplify exposition, we assume n is a power of two.


(a) All balls at the root


(a) All balls choose the first leaf

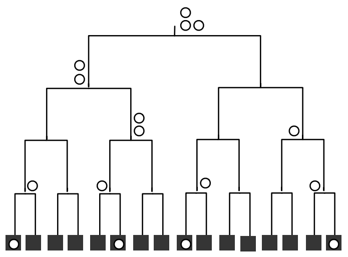

(b) Choices are well distributed

Figure 1: Initial configuration

Figure 2: After one phase

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 $\bar{b}$ in $<_R$ order; for each $\bar{b}$, b_i lets $\bar{b}$ follow its candidate path until $\bar{b}$ reaches a *full* subtree—one with no remaining capacity. If $\bar{b}$ is lucky, it ends up at a leaf in b_i's local tree; otherwise, it stops at a higher level. Irrespective of where $\bar{b}$ 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 tree of 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 position of other balls; this requires a 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 tree, b_i removes it (the ball has crashed). If b_i finds that every ball is at a leaf, it terminates. Otherwise, it picks a new candidate path for itself and 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 (Line 1). Then, b_i repeatedly executes the main loop (Lines 2–26); each iteration is called a *phase*, and each phase has two communication rounds. In round one, b_i first chooses its candidate path (Lines 5–10) one edge at a time, where the probability of each child is the ratio of remaining slots in either subtree (Line 6). Then b_i broadcasts its path (Line 11). After receiving the paths of others, b_i iterates over the balls $\bar{b}$ in $<_R$ order, to compute their new positions. Each ball $\bar{b}$ moves down its path as long as the subtree has remaining capacity (Lines 12-18). Balls that do not announce their paths have crashed and are removed from the local tree (Lines 19-20). In the second round (Lines 22–28), 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.

4.1 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. The process with original identifier id_i runs Balls-into-Leaves for the ball labeled id_i. It then returns the (left-to-right) index of the leaf where the ball terminates, and outputs this rank as its name.

Name uniqueness follows from the fact that no two correct balls can terminate on the same leaf (Theorem 1). *Validity* follows because the number of leaves is n. Termination and complexity follow from the complexity analysis of the Balls-into-Leaves algorithm.

4.2 Correctness

The correctness follows from ensuring that, in every view, the subtrees never exceed their capacities.

Algorithm 1 Balls-into-Leaves Algorithm

Data Structures and Functions

Data Structures

- binary tree with n leaf leaves;
- path$_i$: an ordered set of nodes;

Operations over the tree:

- Remove(b_j) removes b_j from the tree;
- CurrentNode(b_j): current node of b_j;
- UpdateNode(b_j,η): removes b_j from its current node and places it at node η;
- OrderedBalls() returns a set of all balls in the tree, ordered by $<_R$ (first by their depth in the tree, then by ids);
- RemainingCapacity(η): number of leaves in the subtree rooted at node η minus number of balls in that subtree;
- η.LeftChild(), η.RightChild(), η.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_i

```
1:  Initialize: broadcast $\langle b_i \rangle$; $\forall b_j$ received: insert $b_j$ at the root;
2:  repeat                                                          ▷ begin Phase $\phi \leftarrow 1, 2, 3, \ldots$
3:      $\eta \leftarrow$ CurrentNode($b_i$);                                       ▷ begin Round 1 of Phase $\phi$
4:      path$_i \leftarrow \{\eta\}$;
5:      while not $\eta$.IsLeaf() do                                          ▷ choose path randomly
6:          $coin \leftarrow$ RandomCoin($\frac{\text{RemainingCapacity}(\eta.\text{LeftChild}())}{\text{RemainingCapacity}(\eta)}$);
7:          if $coin = heads$ then $\eta \leftarrow \eta$.LeftChild();
8:          else $\eta \leftarrow \eta$.RightChild();
9:          path$_i \leftarrow$ path$_i \sqcup \{\eta\}$;
10:     end while
11:     broadcast $\langle b_i$,path$_i \rangle$;                                               ▷ exchange paths
12:     for all $b_j \in$ OrderedBalls() do
13:         if $\langle b_j$,path$_j \rangle$ has been received then                    ▷ move balls in the priority order
14:             $\eta \leftarrow$ path$_j$.First();
15:             while remainingCapacity($\eta$)$> 0$ do
16:                 $\eta \leftarrow$ path$_j$.Next();
17:             end while
18:             UpdateNode($b_j$,$\eta$);
19:         else
20:             Remove($b_j$);
21:     end for                                                         ▷ begin Round 2 of Phase $\phi$
22:     broadcast $\langle b_i$, CurrentNode($b_i$)$\rangle$;                                      ▷ synchronize
23:     for all $b_j \in$ OrderedBalls() do
24:         if $\langle b_j$,$\eta_j \rangle$ has been received then
25:             UpdateNode($b_j$ ,$\eta_j$);
26:         else
27:             Remove($b_j$);
28:     end for
29: until $\forall b_j \in$ OrderedBalls(): CurrentNode($b_j$).IsLeaf();
```

$B(n,p)$ is the binomial distribution with parameters n and p. By abuse of notation, $B(n,p)$ also denotes a random variable that follows the binomial distribution.

Fact 1 *If* $X{\sim}\mathrm{B}(M,p)$, $Y{\sim}\mathrm{B}(M',p)$, *and* $M \leq M'$, *then* $\Pr(|\mathrm{E}[X]-X| > x) \leq \Pr(|\mathrm{E}[Y]-Y| > x)$.

Fact 2 *If* $0 \leq p \leq 1$, $X{\sim}\mathrm{B}(M,p)$, $Y{\sim}\mathrm{B}(M,\frac{1}{2})$, *then* $\Pr(|\mathrm{E}[X]-X| > x) \leq \Pr(|\mathrm{E}[Y]-Y)| > x)$.

Fact 3 (Chernoff Bound) *If* $X{\sim}\mathrm{B}(m,p)$, *then* $\Pr(|\mathrm{E}[X]-X| > x) < e^{-\frac{x^2}{2mp(1-p)}}$.

Figure 3: Notation and facts from probability theory used in the proofs.

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

Theorem 1 *In the Balls-into-Leaves algorithm, correct balls terminate at distinct leaves.*

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

5 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 protocol. (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 during 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 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 π from the root, the set of balls on π in ϕ is a superset of balls on π in $\phi + 1$.*

Proof By construction, balls can only move down the tree. Fix some node η on path π. The only way new ball b_i at some node μ in phase ϕ reaches η in phase $\phi+1$ is by constructing in ϕ a path from μ that contains η. Thus, by construction of a binary tree, μ is on π. □

We use some notation and facts from theory of probability, shown in Figure 3.

5.1 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 η, and let pM 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 ϕ. By construction, at most M balls reach η. The balls that get stuck at η in phase $\phi{+}1$ are those which had η on their paths in ϕ, but did not fit in a subtree below η. By Lemma 1, these balls have enough space in the sibling subtree of η. We use the notation $balls(\eta,\phi)$ to denote the number of balls at node η in phase ϕ. We denote by $b_{max}(\phi)$ the most populated node in phase ϕ.

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

Proof Choosing between the left and right subtrees of η can be seen as choosing between two bins with capacities pM and $(1-p)M$. Each of the $M' \leq M$ 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, η 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, $Y \sim \mathrm{B}(M', p)$. Then, the number of rejected balls is $Y - Mp$. Since $Mp \geq M'p = E[Y]$, $Y - Mp \leq Y - E[Y]$.

From Fact 2, we obtain that $\Pr(|Y - E[Y]| > x) \leq \Pr(|\frac{M}{2} - \mathrm{B}(M, \frac{1}{2})| > x)$. $\square$

Let us now consider what happens after the first phase.

Lemma 4 *Let i be the depth of node η. Then, for some constant $c > 0$, we have that*
$$\Pr\left(balls(\eta, 2) > c(\tfrac{n}{2^i}\log n)^{\frac{1}{2}}\right) < \tfrac{1}{n^c}.$$

Proof Initially n balls start at the root.

The initial capacity of the subtree rooted at η is $\frac{n}{2^i}$. Then, at most $\frac{n}{2^i}$ balls reach η. Define $Y_\eta \sim \mathrm{B}(\frac{n}{2^i}, \frac{1}{2})$.

Applying Lemma 3 and the Chernoff bound (Fact 3),
$$\Pr(balls(\eta, 2) > x) < \Pr\left(|E[Y_\eta] - Y_\eta| > c(\tfrac{n}{2^i}\log n)^{\frac{1}{2}}\right) < \tfrac{1}{n^c}. \quad \square$$

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 η has enough capacity to accommodate all balls on the path from the root to η.

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

Proof By the path isolation property (Lemma 2), the only balls that may attempt to choose between subtrees of η are those on the path from the root to η. Let i be the depth of η. The total number of balls on path π is at most $i \times b_{max}(\phi)$. By Fact 1, the probability to have more rejected balls is the highest if we inflate the remaining capacity of the subtrees of η to $i \times b_{max}(\phi)$.

Thus, by Lemma 3, the probability that at least x balls get stuck at η can be bounded as follows,
$$\Pr(balls(\eta, \phi+1) > x) \leq \Pr\left(\left|\tfrac{i \times b_{max}(\phi)}{2} - \mathrm{B}(i \times b_{max}(\phi), \tfrac{1}{2})\right| > c\right).$$

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

Since $i \leq \log n$, the claim follows. $\square$

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

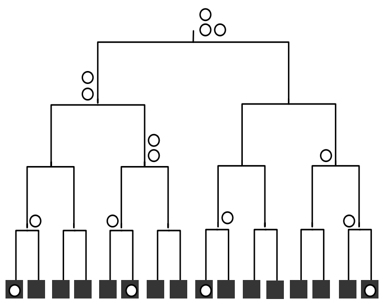
Proof Fix some constant $c > 0$. By Lemma 4, $\Pr\left(balls(\eta, 2) > c(n\log n)^{\frac{1}{2}}\right) < \frac{1}{n^c}$.

By Lemma 5, $\Pr\left(balls(\eta, \phi+1) > c(b_{max}(\phi))^{\frac{1}{2}}\log n\right) < \frac{1}{n^c}$. Since $n^{\frac{1}{2}^{\log\log n}} = O(1)$, we pick some constant c_2 such that $n^{\frac{1}{2}^{c_2 \log\log n}} = 1$.

Let $f(x) = x^{\frac{1}{2}} c \log n$. Taking $x = c(n\log n)^{\frac{1}{2}}$, $f^{c_2 \log\log n}(x) = c^2 \log^2 n$.

Applying Lemma 4, and then Lemma 5 iteratively for $c_2 \log\log n$ phases, we obtain that
$$\Pr\left(b_{max}(\eta, c_2 \log\log n) > c^2 \log^2 n\right) < \tfrac{c_2 \log\log c}{n^c}.$$

Therefore, there exists some small const. $\epsilon < 1$, such that $\frac{c_2 \log\log n}{n^c} < \frac{1}{n^{(c-\epsilon)}}$, and the claim follows. $\square$


(a) entire tree

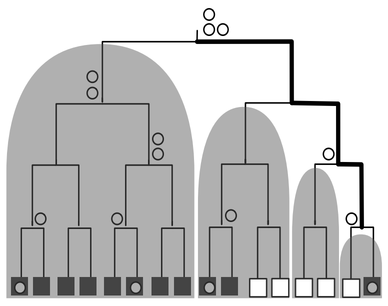

(b) 5 balls on the rightmost path and 5 empty bins reachable from the path

Figure 4: Closer look at a path in a possible configuration

5.2 Part 2 - Bounding the Number of Balls on a Path

Lemma 6 shows that, after $O(\log\log n)$ phases, the number of balls on each path is at most $O(\log^3 n)$ 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 ϕ and a path π from the root to a parent of a leaf. Let $\eta_1, \eta_2, \ldots, \eta_{\log n}$ be the nodes on π, ordered by depth. A *gateway* node (or simply a gateway) is a child of η_i that is not on π. For uniformity, we combine both children of the last node on π (tree leaves) into one gateway meta-child [2]. For instance, in the sample configuration from Figure 4a, consider the rightmost path (highlighted in Figure 4b); all the left children of nodes on π are gateways. By construction, the sum of remaining capacities of all gateway subtrees (corresponding to empty leaves reachable from π) is equal to the total number of balls on π. In phase ϕ, most 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 π either in phase ϕ or $\phi + 1$.

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

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

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

Case 2: b_k reaches η_i. By assumption, b_k is among $\min(m, c_i)$ highest priority balls that have chosen the same gateway. Thus, b_k escapes π into the gateway subtree of η_i.[3]

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

We now bound the probabilities with which balls try each gateway on π.

[2]Obviously, a collision in a subtree with 2 leaves is solved within one phase.

[3]In Case 2, ball b_k can stop somewhere else deeper in the tree, but this no longer affects the analysis of π.

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

Proof (Sketch) Recall from Lines 5-9 of Algorthim 1 that balls construct paths trying between the children of each node with probabilities indexed by their remaining capacities.

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 η_i (i.e., all gateway subtrees and the non-gateway subtree at η_i) is M_i. Thus, every ball tries the gateway at η_i with prob. at least c_i/M_i. $\square$

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

Lemma 9 *Let M be the total number of balls on π in phase ϕ. For some const. $c > 1$, less than $\frac{M}{c}$ balls have escaped π after $\phi + 1$ with prob. $< e^{\frac{-M}{c}}$.*

Proof By Lemma 8, M_i balls try the gateway at η_i with prob. at least $\frac{c_i}{M_i}$.
By Lemma 7, for any $\gamma > 1$, $\frac{c_i}{\gamma} \leq c_i$ highest priority balls that choose the gateway at η_i, escape π in ϕ or $\phi + 1$. Define as success the event that some ball chooses the gateway at η_i. By Lemma 8, such an event occurs with prob. at least c_i/M_i among M_i tries. Thus, the number of successes follows $\mathrm{B}(M_i, \frac{c_i}{M_i})$. From the Chernoff Bound (Fact 3),

$$\Pr\left(\mathrm{B}(M_i, \tfrac{c_i}{M_i}) < c_i - \tfrac{c_i}{\gamma}\right) < e^{-\frac{M_i}{\gamma}}.$$

Recall that $\sum_{1\leq i\leq \log n} c_i = M$. Considering all η_i, $1 \leq i \leq \log n$ on π, the sum of successes is less than $M - \frac{M}{\gamma}$ with prob. $< e^{-\frac{M}{\gamma}}$. Since γ is arbitrary, the claim follows. $\square$

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

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

Proof Fix c to be a constant. By Lemma 9, at least $\frac{M}{c}$ balls escape from π after phase $\phi+1$, with probability at least $1 - e^{-\frac{M}{c}}$. Starting with M balls, and iterating in Lemma 9 over the remaining balls for $2c\log M$ phases, we obtain that the probability that π is not empty after $2c\log M$ phases is at most $2c\log M e^{-\frac{M}{c'}} < e^{-\frac{M}{(c-\epsilon)}}$ for some small constant $\epsilon < 1$. $\square$

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 $\left(1 - \frac{1}{n^c}\right)$, where $c > 0$ is a constant.*

Proof From Lemma 6, after $O(\log\log n)$ phases, the probability there is a path with more than $O(\log^3 n)$ balls, is less than $\frac{1}{n^{c'}}$, for some const. $c' > 0$. Taking $M = O(\log^3 n)$ balls on path π from the root to a parent of some leaf, by Lemma 10 and a union bound over all such $n/2$ paths, the probability that some path is not empty is less than $\frac{1}{n^{c'}}$, for some constant $c' > 0$. Putting together the above results, we get that the probability that the tree still has a populated inner node is at most $\frac{1}{n^{c'+c''}}$ 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 $> 1 - \frac{1}{n^c}$.
Since each phase consists of 2 rounds, the claim follows. $\square$

5.3 Crashes

To show that crashes do not slow down termination, we continue the analysis of some path π 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 ϕ which contains at least one failure, we compare the local view V of ball b_i in ϕ with its hypothetical view V' in an execution with a failure free ϕ. 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 π 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 which occurs in a subtree of b_i. Such a failure implies that, in V, the total capacity of all gateways on π is larger than the number of balls on π. Thus, b_i is at most as likely to be among the first highest priority balls that have chosen the same gateway in ϕ. Since the choice of b_i and π is arbitrary, the argument applies to every ball in every view.

6 Early Terminating Extension

We now extend the algorithm to terminate more quickly 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_{lowest} sends to every second ball (by label order) and then crashes, so that all other balls collide in pairs.

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 $\Omega(\log n)$, due to the lower bound of [9].

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 5-10 in Algorithm 1 with the following: ball b_i constructs path deterministically towards the leaf ranked by b_i in OrderedBalls(); 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.

Theorem 3 *In a failure free execution, the modified algorithm terminates deterministically 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 algorithm terminates in $O(\log\log f)$ rounds with prob. at least $\left(1 - \frac{1}{n^c}\right)$, where $c > 0$ is a constant.*

7 Conclusion

Extending the classical balls-into-bins technique, we proposed Balls-into-Leaves, a randomized algorithm that places n balls into n leaves of a tree in $O(\log\log n)$ rounds w.h.p. An extension of the algorithm 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. These 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.

References

[1] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 238–247, New York, NY, USA, 1995. ACM.

[2] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. Accepted to J. ACM. Available at: http://cs-www.cs.yale.edu/homes/aspnes/papers/asynchronous-renaming.pdf, September 2013.

[3] Dan Alistarh, Hagit Attiya, Rachid Guerraoui, and Corentin Travers. Early deciding synchronous renaming in O(log f) rounds or less. In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity*, SIROCCO '12, Reykjavik, Iceland, June 2012.

[4] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.

[5] Petra Berenbrink, Kamyar Khodamoradi, Thomas Sauerwald, and Alexandre Stauffer. Balls-into-bins with nearly optimal load distribution. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 326–335, Montreal, Canada, 2013.

[6] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[7] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119–134, 2011.

[8] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology upper and lower bounds for renaming. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 295–304, Toronto, Canada, 2008. ACM.

[9] Soma Chaudhuri, Maurice Herlihy, and Mark Tuttle. Wait-free implementations in message-passing systems. *Theoretical Computer Science*, 220(1):211–245, June 1999.

[10] Oksana Denysyuk and Luís Rodrigues. Byzantine renaming in synchronous systems with $t < N$. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 210–219, Montreal, Canada, 2013. ACM.

[11] Danny Dolev and Raymond Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, STOC '82, pages 401–407, San Francisco (CA), USA, 1982.

[12] Eli Gafni. The extended bg-simulation and the characterization of t-resiliency. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC '09, pages 85–92, Bethesda, Maryland, USA, 2009. ACM.

[13] Gaston Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, April 1981.

[14] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999.

[15] Leslie Lamport, Robert Shostak, and Michael Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[16] Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing. *CoRR*, abs/1102.5425, 2011.

[17] Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing: Extended abstract. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 11–20, San Jose, CA, USA, 2011. ACM.

[18] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, Oct 2001.

[19] Michael Okun. Strong order-preserving renaming in the synchronous message passing model. *Theoretical Computer Science*, 411(40-42):3787 – 3794, 2010.

A Correctness

We now 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 at ball b_i is the same as the position of b_i at 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 (Line 1).

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 ϕ (Line 22 of Algorithm 1). However, since b_j is correct, b_j must have executed Line 22 in phase ϕ. 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 position of correct balls is synchronized across the views. Assume, w.l.o.g., $b_i < b_j$. Thus, by definition of the order $<_R$, $b_i <_R b_j$ 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 $\leq$ the number of leaves in that subtree.*

Proof We prove the claim by induction over the phase index ϕ. 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 all least correct balls (Proposition 2), and (Proposition 3), balls simulate the descent of correct balls in a consistent order (if $b_j <_R b_k$, b_j is moved before b_k), 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 From Proposition 1 and Lemma 1, after correct ball b_i reaches a leaf and announces its position, all correct balls have b_j at its 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 ϕ, 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 ϕ. 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. □

Since by Lemma 11 processes require at most n fault-free phases to reach the bottom, and 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 4.

Theorem 4 *The modified algorithm terminates in* $O(\log \log f)$ *rounds with prob. at least* $\left(1 - \frac{1}{n^c}\right)$*, 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. On the other hand, all surviving balls have b_i in their local views. Thus, at most $k-1$ other survivors see their ranks in the interval $i..(i+k)$.

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 $\frac{n}{2^{\log n - \lceil \log f \rceil}} \leq n$ parallel instances of Balls into Leaves with at most f balls each. From Theorem 2, and for a sufficiently large const. $c > 0$, the claim follows. □