An Efficient Randomized Decentralized Algorithm for the Distributed Trigger Counting Problem

Vijay K. Garg

University of Texas at Austin,
{garg@ece.utexas.edu

Abstract

Consider a distributed system with n processors, in which each processor receives some triggers from an external source. The distributed trigger counting problem is to raise an alert and report to a user when the number of triggers received by the system reaches w, where w is a user-specified input. The problem has applications in monitoring, global snapshots and other distributed settings. The main result of the paper is a decentralized and randomized algorithm with expected message complexity $O(n \log w)$. Moreover, every processor in this algorithm receives no more than $O(\log w)$ messages. It is known that any deterministic algorithm has message complexity $\Omega(n \log w)$ and maximum processor load $\Omega \log(w/n)$.

1 Introduction

In this paper, we study the *distributed trigger counting* (DTC) problem. Consider a distributed system with n processors, in which each processor receives some triggers from an external source. The distributed trigger counting problem is to raise an alert and report to a user when the number of triggers received by the system reaches w, where w is a user specified input. The sequence of processors receiving the w triggers is not known apriori to the system. Moreover, the number of triggers received by each processor is also not known. We are interested in designing distributed algorithms for the DTC problem that are communication efficient and are also decentralized.

The DTC problem arises in applications such as distributed monitoring and global snapshots. Monitoring is an important issue in networked systems such as sensor networks and data networks. Sensor networks are typically employed to monitor physical or environmental conditions such as traffic volume, wildlife behavior, troop movements and atmospheric conditions, among others For example, in traffic management, one may be interested in raising an alarm when the number of vehicles on a highway exceeds a certain threshold. Similarly, one may wish to monitor a wildlife region for the sightings of a particular species, and raise an alert, when the number crosses a threshold. In the case of data networks, example applications are monitoring the volume of traffic or the number of remote logins. See, for example, [9] for a discussion of applications of distributed monitoring. In the context of global snapshots (example, checkpointing), a distributed system must record all the in-transit messages in order to declare the snapshot to be valid. Garg et al. [6] showed the problem of determining whether all the in-transit messages have been received can be reduced to the DTC problem (they call this the distributed message counting problem).

Most prior work (e.g. [4, 9, 8]) primarily consider the DTC problem in a centralized setting where one of the processors acts as a master and coordinates the system, and the other processors act as slaves. The slaves can communicate only with the master (they cannot communicate among themselves). Such a scenario applies where a communication network linking the slaves does not exist or the slaves have only limited computational power. Prior work addresses various issues arising in such a setup, such as message complexity. They also consider variations and generalizations of the DTC problem. One such variation is approximate threshold computation, where system need not raise an alert on seeing exactly w triggers; it suffices if the alert raised upon seeing at most $(1+\epsilon)w$ triggers, where ϵ is some user specified tolerance parameter. Prior work also considers aggregate function more general than counting. Here, each input trigger i is associated with a value α_i. The goal is to raise an alert when some aggregate of these values crosses the threshold (an example, aggregate function is `sum`).

Note that the Echo or Wave algorithms [2, 13, 14] and the framework of repeated global computation[7] are not easily applicable for the DTC problem because the triggers arrive at processors asynchronously at unknown times. Computing the sum of all the trigger counts just once is not enough and repeated computation results in an excessive number of messages. The DTC problem is also different from the distributed resource controller problem studied in [1, 10, 5]. In the resource controller problem, there are a fixed number of *permits* (or resources), M, at the root node and the goal of the resource controller algorithm is to serve the request for a resource either with a *permit* or a *reject*. Their protocol guarantees that the request for a permit is rejected only if there are at least $M-W$ requests that are eventually going to be satisfied, where W is a parameter that gives a bound on the maximum number of "wasted" permits. The resource controller problem may appear similar to DTC problem because counting each trigger may be considered as consuming a permit. However, the ability to reject a request makes the resource controller problem quite different from the DTC problem. For example, in the resource controller problem, the number of permits stored outside the root never exceeds W. The communication complexity of their protocol is $O(n \log^2 n \log(M/(W+1)))$.

In this paper, we consider a general distributed system where any processor can communicate with any other processor and all the processors are capable of performing basic computations. We also assume that all messages are delivered reliably in first-in-first-out order. This setting is common in data networks. In such a scenario, a decentralized algorithm would be desirable, given that a centralized system suffers from congestion issues.

Our goal is to design a distributed algorithm for the DTC problem that is communication efficient and decentralized. We shall use the following two natural parameters that measure these two important aspects.

- The *message complexity*, which is defined to be the number of messages exchanged between the processors.
- The MaxLoad, which is defined to be the maximum number of messages received by any processor in the system.

Garg et al. [6] studied the DTC problem for a general distributed system. They presented two algorithms [1]. (i) A centralized algorithm with message complexity $O(n \log w)$. However, the MaxLoad of this algorithm can be as high as $\Omega(n \log w)$. (ii) a tree-based heuristic algorithm with message complexity $O(n \log n \log w)$. This algorithm is more decentralized, but its MaxLoad can be as high as $O(n \log n \log w)$, in the worst case. They also proved a lowerbound on the message complexity. They showed that any deterministic algorithm for the DTC problem should exchange $\Omega(n \log(w/n))$ messages (i.e., for any deterministic algorithm, the message complexity must be $\Omega(n \log(w/n))$). So, the message complexity of the centralized algorithm is optimal asymptotically. However, this algorithm has MaxLoad as high as the message complexity.

In this paper, we present a randomized algorithm that is optimal in terms of both the message complexity and MaxLoad. Its expected message complexity is $O(n \log w)$. Moreover, with constant probability (arbitrarily close to 1) the MaxLoad is $O(\log w)$. We call this algorithm OptRand. In the light of the lowerbound of Garg et al. [6], notice that OptRand is optimal in terms of both the message complexity and MaxLoad, when $w \geq n^2$.

For $1 \leq i \leq w$, the external source delivers the ith trigger to some processor x_i. We call the sequence $x_1, x_2, \ldots, x_w$ as a *trigger pattern*. We note that the above bounds for the OptRand algorithm hold for any *trigger pattern*, even if fixed by an adversary. The main result of the paper is formally stated below.

Theorem 1.1 *Fix any trigger pattern. Assume that $w \geq n$. The expected number of messages exchanged in the* OptRand *algorithm is $O(n \log w)$. Furthermore, there exists a constant c such that the following is true. For any $\epsilon > 0$, with probability at least $1 - \epsilon$, every processor receives at most $c(1/\epsilon) \log w$ messages (meaning, the* MaxLoad *is at most the above quantity).*

We note that the OptRand algorithm works for any value of w. When $w \leq n$, the message complexity is $O(n \log n)$ and the MaxLoad is $O(\log n)$. However, for the ease of exposition, we will assume throughout the paper that $w \geq n$.

[1]They prove slightly better bounds for the two algorithms. But, for $w \geq n^2$, the bounds stated here are the same as their actual bounds

For the sake of better exposition, we will not present the OptRand algorithm directly. Instead, we will first present two precursors: CoinPass algorithm and SemiRand algorithm. CoinPass algorithm is a randomized algorithm with expected message complexity $O(n \log w)$, but high MaxLoad. The SemiRand algorithm has expected message complexity $O(n \log w)$ and MaxLoad $O(\log w)$, taking into account only certain type of messages. The OptRand algorithm will consider all types of message and we will analyze it completely, taking all these types of messages in to account. The advantage with first presenting CoinPass and SemiRand algorithms is that they provide us with a simpler setup to explain two key aspects of the OptRand algorithm and its analysis.

2 A Deterministic Algorithm

For the DTC problem, Garg et al. [6] presented an algorithm with the message complexity of $O(n \log w)$. In this section, we describe an alternative deterministic algorithm which is similar to theirs and has the same message complexity. The aim of presenting this algorithm is to highlight the difficulties in designing an optimal algorithm with message complexity $O(n \log w)$ and MaxLoad $O(\log w)$, simultaneously.

A naive algorithm for the DTC problem works as follows. One of the processors acts as a *master* and every processor sends a message to the master upon receiving each trigger. The master keeps count on the total number of triggers received. When the count reaches w, the user is informed and the protocol ends. The disadvantage with this algorithm is that its message complexity is $O(w)$.

A natural idea is avoid sending a message to the master for every trigger received. Instead, a processor will send one message for every B triggers received. Clearly, setting B to a high value will reduce the number of messages. However, care should taken to ensure that the system does not enter the *dead state* For instance, suppose we set $B = w/2$. Then, the adversary can send $w/4$ triggers to some selected four processors. Notice that none of these processors would send a message to the master. Thus, even though all the w triggers have been delivered by the adversary, the system will not detect the termination. We say that the system is the dead state. Naturally, it is critical to avoid entering the dead state.

Our deterministic algorithm with message complexity $O(n \log w)$ is described next. A predetermined processor would serve as the master. The algorithm works in multiple rounds. We start by setting two parameters: $\hat{w} = w$ and $B = \hat{w}/(2n)$. Each processor would send a message to the master for every B triggers received. The master will keep count of the triggers reported by other processors and the triggers received by itself. When the count reaches $\hat{w}/2$, it declares *end-of-round* and sends a message to all the processors to this effect. In return, each processor sends the number of unreported triggers to the master (namely, the triggers not reported to the master). This way, the master can compute w', the total number of triggers received so far in the system. It recomputes $\hat{w} = \hat{w} - w'$; the new $\hat{w}$ is the number of triggers yet to be received. The master recomputes $B = \hat{w}/(2n)$ and sends this number to every processor. The next round starts. When $\hat{w} < (2n)$, we set $B = 1$.

We now argue that the system never enters a dead state. Consider the state of the system in the middle of any round. Each processor has less than $\hat{w}/(2n)$ unreported triggers. Thus, the total number of unreported triggers is less than $\hat{w}/2$. The master's count of reported triggers is less than $\hat{w}/2$. Thus, the total number of triggers delivered so far is less than $\hat{w}$. So, some more triggers are yet to be delivered. It follows that the system is never in a dead state and the system will correctly terminate upon receiving all the w triggers.

Notice that in each round, $\hat{w}$ decreases at least by a factor of 2. So, the algorithm terminates after $\log w$ rounds. Consider any single round. A message is sent to the master for every B triggers received and the rounds gets completed when the master's count reaches $\hat{w}/2$. Thus, the number of messages sent to the master is $\hat{w}/(2B) = n$. At the end of each round, the $O(n)$ messages are exchanged between the master and the other processors. Thus, the number of messages per round is $O(n)$. The total number messages exchanged during all the rounds is $O(n \log w)$.

The above algorithm is optimal in terms of message complexity. However, the master may receive upto $O(n \log w)$ messages and so, the MaxLoad of the algorithm is $O(n \log w)$. In the next few sections, we present a *randomized* algorithm which achieves both the message complexity and MaxLoad of an optimal deterministic algorithm.

3 CoinPass Algorithm

In this Section, we present a randomized algorithm, called the CoinPass algorithm, that has expected message complexity $O(n \log w)$. The ideas developed in this section will be applied in the OptRand algorithm.

For the ease of exposition, let us assume that $n = 2^k - 1$, for some k. The processors are arranged in the form of a complete binary tree (See Figure **??**). We will imagine that each processor occupies a node position in the tree. The configuration of the tree (namely, the information of which node occupies which position), is predetermined and known to all the processors.

The depth of the tree is $d = \log(n+1)$. Let $\ell = d - 1$. The tree has 2^ℓ leaf nodes. We will identify these nodes using bit-strings of length ℓ. Scanning leaf nodes from left to right, we assign bit-strings in the lexicographic order. For a leaf node u, let sig_u denote the bitstring assigned to the node u; sig_u is called the *signature* of u. See Figure **??**, where the signatures are shown below the leaf nodes. For $1 \leq i \leq \ell$, let $\text{sig}_u[i]$ denote the ith bit of sig_u. For an internal node u, let $\mathit{level}(u)$ denote the level number at which u appears. For instance, the root node appears in level number 1 and its two children appear in level number 2.

The CoinPass algorithm proceeds in multiple rounds. To start with, we set parameters $\hat{w} = w$ and $B = \hat{w}/(2n)$. The algorithm works by using the notion of coins. Each processor x keeps count of the number of triggers received. For each batch of B triggers received, the processor x generates a coin. Then the processor x considers the set of all processors occupying the leaf nodes of the tree and chooses one of these processors at random, say the processor y. (The processor x knows which processors occupy the leaf node positions, since the configuration of the tree is known to all the processors.) The processor x sends the newly generated coin to the processor y. The coin will then be passed through various nodes in the tree, before being *deposited* at one of the nodes.

The process would maintain the following invariant: if an internal node has a coin deposited in it, then both its children nodes have coins deposited in them. We now describe the procedure followed by a processor x occupying a node u:

- For each batch of B triggers received, the processor x *generates* a coin. It then chooses one of the processors occupying the leaf nodes at random and sends the coin to the chosen processor. Let y be the chosen processor and v be the leaf node occupied by y. We will say that the coin is *generated* by x at node u and it is *initiated* by y at node v. The signature of initiating node sig_v is associated with the coin.

- Suppose x receives a coin from some other node:
  (*Case 1*) Suppose the node u already has a coin deposited in it: In this case, the coin is forwarded to the parent of u.
  (*Case 2*) Suppose the node u does not possess a coin. If u is a leaf node, the coin is consumed by x and deposited at u. On the other hand, if u is an internal node, then the following procedure is applied.
  (*Case 2.1*) Suppose both the children of u have coins. In this case, the coin is deposited at u. (*Case 2.2*) Suppose only one of the children of u possesses a coin. In this case, the coin is passed on to the child that does not have a coin.
  (*Case 2.3*) Suppose both the children of u do not have coins [2]. In this case, we consider the leaf node v where the coin was initiated (Recall that this information is associated with the coin itself). We consider the bit $b = \mathrm{sig}_v[i]$, where $i = \mathit{level}[u]$. If $b = 0$, the coin is passed to the left child; and if $b = 1$, then the coin is passed to the right child.

Example: See Figure **??**. Suppose a coin gets initiated at node u_{10}; it will be deposited at node u_2. On the other hand, suppose a coin gets initiated at node u_{13}; it will get deposited at node u_{15} □

The action of the processor x occupying the root node of the tree is important. Eventually, when a coin gets deposited at the root node, the processor x behaves as follows. At this point, the processor x declares *end-of-round*. Then it computes the total number of triggers received in the system by contacting all the processors; let this number be w'. Then the processor resets $\hat{w} = \hat{w} - w'$. The value B is recomputed as $B = \hat{w}/(2n)$ and sent to all the processors. The next round starts. (The process of collecting the number of triggers received by each processor can be performed in a recursive fashion; this would avoid increasing the load on the processor occupying the root node).

We shall first argue that the system never enters the dead state. Consider the system in the middle of any round. Notice that once n coins are generated, all the nodes get a coin each. In particular, in this case, the root node gets a coin and initiates the next round. So, consider the scenario where at most $(n-1)$ coins have been generated. Each such coin represents $\hat{w}/(2n)$ triggers received. On the other hand, at each processor, the number of unreported triggers is $< B$ (because, for each B triggers a coin is generated). Thus, the total number triggers received so far in the system is at most

$$(n-1)\frac{\hat{w}}{2n} + n\frac{\hat{w}}{2n} < \hat{w}.$$

It follows that the system enters the next round at least when all the $\hat{w}$ messages are received.

Next, let us count the number of rounds taken by the algorithm. The system enters the next round only when n coins are generated. This happens only when at least $n(\hat{w}/(2n))$ triggers are received. Thus, $w' \geq n(\hat{w}/(2n)) = \hat{w}/2$. Hence, in each round, $\hat{w}$ goes down by a factor of at least two. So, the maximum number of rounds involved is at most $\log w$.

Now, let us analyze the number of messages exchanged. The number of coins generated in each round is exactly n. Each coin generated may climb up all the way to the root and climb down to the leaf level. The depth of the tree is $O(\log n)$ and so, each coin passes through at most $O(\log n)$ nodes. Passing a coin from one node to another node can be accomplished using a constant number of messages. Thus, the number of messages exchanged in a single round is $O(n \log n)$. Summed up

[2]In a practical setting, this case can be handled by passing the coin to a randomly chosen child of u. But, we adopt this specific procedure, since it simplifies the analysis of the algorithm.

over all the rounds, the number of messages is $O(n \log n \log w)$. Notice that this is a factor $\log n$ higher than the optimal message complexity of $O(n \log w)$.

Using a more refined analysis of the CoinPass algorithm, we prove the following theorem, which shows that in each round, the expected number of messages is only $O(n)$. The proof is presented in Appendix B.

Theorem 3.1 *For any trigger pattern, in any round, the expected number of messages exchanged is $O(n)$.*

We can apply the linearity of expectation and get the expected number of messages for all the rounds put together.

Corollary 3.2 *Consider the* CoinPass *algorithm. For any trigger pattern, the expected number of messages exchanged is $O(n \log w)$.*

The above result shows that the CoinPass algorithm has optimal message complexity. However, it is not a good algorithm from the perspective of load balancing. Consider the processor x occupying the root node of the tree. We call the subtree rooted at the left child (resp. right child) of the root node as the *left subtree* (resp. *right subtree*). Of the n coins generated, let L denote the number of coins that get initiated in the leaves of the left subtree and let R denote the number of coins that get initiated at the leaves of the right subtree. Notice that the random variables L and R follow the binomial distribution. The expectation of L and R are $\mathbf{E}[L] = \mathbf{E}[R] = n/2$. However, with reasonably high probability $|L - (n/2)|$ will be $\Omega(\sqrt{n})$. Once the leaves of the left subtree initiate $(n-1)/2$ coins, this subtree will be full of coins. Then, the "extra" $\sqrt{n}$ coins will pass through the root node and get deposited somewhere in the right subtree. (A similar phenomenon will happen, if the right subtree becomes full). Thus, the processor x occupying the root node is likely to receive $\Omega(\sqrt{n})$ messages. The same issue arises for processor occupying the other positions as well; namely, a processor occupying a node u is likely to receive $\Omega(\sqrt{n'})$ messages, where n' is the size of the subtree rooted at u. Thus, with reasonably high probability, the MaxLoad will be $\Omega(\sqrt{n})$.

Recall that our main result aims for an algorithm with MaxLoad $O(\log w)$. When w is polynomial in n, this algorithm should have MaxLoad $O(\log n)$. Thus, the plain CoinPass algorithm has MaxLoad much higher than what we are aiming for. In the next section, we modify the CoinPass algorithm and describe a new algorithm called SemiRand that makes progress towards rectifying this issue.

Remark: A minor issue with the CoinPass algorithm is that it will cease to function properly, when eventually $\hat{w}$ gets below n. In this case, we must generate a coin for each trigger and number of such coins is $\hat{w} < n$. So, these coins cannot fill all the n nodes in the tree and no coin will be deposited in the root. As a consequence, the algorithm will not terminate correctly. We handle by issue by switching to a more straightforward algorithm, when $\hat{w}$ gets below n. This is explained in Appendix A.

4 The SemiRand Algorithm

In this section, we present an algorithm called SemiRand. As discussed in the previous section, the main issue with the CoinPass algorithm is that certain node positions are hot-spots that receive a lot of messages. The key idea behind the SemiRand algorithm is *swapping*: the idea is to regularly move out the processors occupying the hot-spot positions. This is accomplished by picking some other processor at random and swapping the node positions of the two processors. The algorithm is described next.

The SemiRand algorithm is a modification of the CoinPass algorithm. The SemiRand algorithm introduces some additional type of messages. We will refer to the original messages exchanged in the CoinPass algorithm as the *coin related messages*.

We start with a randomly chosen tree configuration; namely, which processor occupies which position is fixed randomly. The algorithm is driven by a threshold parameter Δ to be fixed later. In addition, for each node position u, we will keep track of the number of coin related messages received at this node position in a counter δ_u. Consider a processor x occupying a node position u. Whenever δ_u crosses Δ, the processor initiates a *swap process*. The processor x chooses a processor y uniformly at random. Let v be the node position occupied by y. Then the processors x and y swap their node positions. This involves the following steps. First, the nodes exchange state information associated with the original node positions; the state information includes information such as the current neighbors (i.e., processors occupying the children and parent nodes), whether the node has a coin or not, and the δ value of the node. Once the swap happens, the processor y will continue executing the code or procedure associated with the node u (from where x left off); similarly, the processor x will continue executing the code associated node v (from where y left off). We will call u the *source-end node* of the swap and v the *destination-end node* of the swap.

An interesting implication of swapping is that the processors occupying the parent and children nodes of u and v must be informed about the swap. This is necessary because these neighboring processors should thereafter communicate with the new processors. All these can be accomplished using $O(1)$ message exchanges, per swap. We call these the *swap related messages*.

Another interesting issue arises due to swapping and it is related to the process of initiating a coin. Recall that when a processor x generates a coin, the coin must be initiated at some random leaf node position. However, x does not know the current set of processors occupying the leaf nodes (since the set of processors occupying the leaf node positions may keep changing because of swapping). To overcome the issue, we use a *leaf polling process*. The process uses a hot-potato-like algorithm, described next. The processor x will randomly pick a processor y and pass the coin to y. If y is occupying a leaf node, it will initiate the coin; otherwise, y will chose another processor y' at random and pass the coin to it. These trials are continued until the coin reaches a processor occupying a leaf node position. Luckily, the process does not increase the message complexity much. Since there are $n/2$ leaf node positions, each trial has a probability of $1/2$ to succeed. So, the expected number of trials or polls is only 2. We will call the messages associated with this polling process as *poll related message*. This completes the description of the SemiRand algorithm.

We now analyze the SemiRand algorithm and show that with high probability, no processor receives more than $O(\log w)$ coin related messages. Notice that this claim does not bound the swap related and poll related messages received by the processors. Handling these messages requires modifying the algorithm and leads to a more involved analysis. This is done in Section 5, where we describe the OptRand algorithm and show that in the OptRand algorithm, with high probability, the total number of messages received by any processor is only $O(\log w)$.

We first describe and analyze a probabilistic ball game that captures the essence of the SemiRand algorithm. The analysis of this game will be used to derive a probabilistic bound on the number of coin related messages. The probabilistic game involves seats, girls and balls. These correspond to node positions, processors and coin related messages in the SemiRand algorithm, respectively.

4.1 A Probabilistic Ball Game

The game is driven by two parameters n and K, both positive integers. The game involves a set of n seats $S = \{s_1, s_2, \ldots, s_n\}$, a set of n girls $G = \{g_1, g_2, \ldots, g_n\}$ and K balls. By a *seating arrangement*, we mean a one to one onto mapping $\sigma : S \rightarrow G$ (in other words, σ is a permutation). Given a seating arrangement σ, we imagine that a seat s is occupied by the girl specified by $\sigma(s)$. Let Π denote the set of all $n!$ seating arrangements possible. A *ball sequence* is a sequence seats $\tau = b_1, b_2, \ldots, b_K$, where each $b_i \in S$.

Imagine that an adversary has fixed a ball sequence τ. The game proceeds in K rounds. In the ith round, a ball is given to the girl occupying the seat b_i. Intuitively, we would like to achieve load balancing; meaning, each girl g should receive roughly K/n balls. We wish to achieve the above load balancing irrespective of the ball sequence fixed by the adversary. (Notice that this cannot be achieved, if the seating arrangement does not change across rounds. For instance, if the adversarial ball sequence τ simply repeats the same seat s_5 K times, then the girl occupying the seat s_5 will get all the balls.) In our game, we will randomly swap positions of the girls in each round. The procedure is explained next.

Fix any ball sequence $\tau = b_1, b_2, \ldots, b_K$. The game proceeds in K rounds. We start with some random seating arrangement π_0. For $1 \leq i \leq K$, two events happen in the ith round:

1. A ball is given to the girl occupying the seat b_i, in the seating arrangement π_{i-1}; namely, the ball is given to the girl $g = \pi_{i-1}(b_i)$.

2. The girl g chooses a girl $g' \in G$ at random. Then the girls g and g' swap their seats. Let π_i be the new seating arrangement, after the swapping. Notice that with probability $1/n$, g' could be the same as g; in this case, no swapping happens and we get $\pi_i = \pi_{i-1}$.

We call the ball given in the ith round as the ith ball.

For a girl g, let B_g denote the number of balls received by g. Let $B_{\max}$ denote the maximum of B_g over all girls g. We will show that with high probability every girl receives only $O(K/n)$ balls; in other words, $B_{\max} = O(K/n)$. The above claim will be proved by appealing to Chernoff bounds. This requires us to establish certain independence property about our process; this is done next.

Define random variables $Y_1, Y_2, \ldots Y_K$, where Y_i represents the girl who received the ith ball. The following result shows that these random variables are uniformly distributed and that they are mutually independent. The theorem is proved in Appendix C.

Theorem 4.1 *The random variables $Y_1, Y_2, \ldots, Y_K$ are uniformly distributed and are mutually independent. Meaning: (i) for each $1 \leq i \leq K$ and any girl $g \in G$, $\Pr[Y_i = g] = 1/n$; (ii) for any sequence of girls, $a_1, a_2, \ldots, a_K \in G$,*

$$\Pr[(Y_1 = a_1) \wedge (Y_2 = a_2) \wedge \cdots \wedge (Y_K = a_K)] = \frac{1}{n^K}.$$

As a corollary, we get the following result, which shows that with high probability every girl receives at most $O(K/n)$ balls. The corollary is proved in Appendix C.

Theorem 4.2 *Fix any ball sequence $\tau = b_1, b_2, \ldots, b_K$ and any constant $t \geq 6$. Then,*

$$\Pr[B_{\max} \geq t(K/n)] \leq n2^{-\frac{tK}{n}},$$

Moreover, the above inequality is true, if we replace K on both LHS and RHS by any $K' \geq K$.

Extended Ball Game: We now extend the above game by introducing an additional parameter Δ. Let $\tau = b_1, b_2, \ldots, b_K$ be any ball sequence. For each seat $s \in S$, scan the sequence from left to right and set a mark on every Δth appearance of s; the balls corresponding to the marked occurences are declared *golden balls*. The rest of the balls are called *white balls*. Figure **??** illustrates the idea for $\Delta = 3$; the golden balls are shown double circled. The game process is modified as follows: a girl will initiate a swap only on receiving a golden ball (namely, a girl will not perform a swap upon receiving white balls). The original game corresponds to the case where $\Delta = 1$, where every ball is golden. As before, let B_g represent the number of balls received by a girl g (counting both white and golden balls); let $B_{\max}$ represent the maximum of B_g over all the girls g.

Extending Theorem 4.2, we can prove the following result. The proof is given in Appendix C.

Theorem 4.3 *Fix any ball sequence $\tau = b_1, b_2, \ldots, b_K$, any $\Delta \geq 1$ and any $t \geq 6$ Then,*

$$\Pr[B_{\max} \geq 2t(K/n)] \leq 2n2^{-\frac{tK}{\Delta n}}.$$

Moreover, the above inequality is true, if we replace K on both LHS and RHS by any $K' \geq K$.

4.2 Analysis of the SemiRand Algorithm

In this section, we relate the SemiRand algorithm to the games introduced in the previous section and derive an upperbound on the number of coin related messages received by any processor (MaxLoad). We note that the polling related messages and swap related messages are not accounted for in our analysis. These will be considered in the next section, where we present the OptRand algorithm.

Fix any trigger pattern P. Let N denote the total number of coins that are generated (accounting for all the rounds). The number N is completely determined by the trigger pattern P (i.e., it is independent of any randomness used by the algorithm). In each round, exactly n coins are generated and the number of rounds is at most $\log w$. Thus, $N \leq n \log w$.

Let L denote the set of all leaf node positions. Each of the N generated coins gets initiated at a random node from L (on successful polling). Let $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_N$ denote the sequence of leaf node positions where the coins get initiated (where α_i is the leaf node position where coin i gets initiated). The sequence α is a random variable determined by the leaf polling process. The set of all possibilities of α is given by $\Omega = L \times L \times \cdots \times L$, where L is repeated N times. Let us call each element of Ω as an *initiation configuration*. The set Ω represents the sample space over which the N coins get initiated. In other words, the leaf polling process can be simulated by simply picking an initial configuration α uniformly at random from Ω.

Let us now consider the number of coin related messages exchanged, denoted M. The random variable M is a function of the randomly picked initiation configuration α. (The value of M is determined purely by α and the random swaps do not play a role in determining M). We will invoke Markov's inequality and show that for a significant fraction of the initiation configurations, $M = O(n \log w)$.

Fix any $\epsilon > 0$; this will represent our failure probability. For an initiation configuration α, let $M(\alpha)$ denote the number of coin related messages exchanged. Corollary 3.2 shows that the expected number of coin related messages exchanged is $O(n \log w)$. Meaning,

$$\mathbf{E}_{\alpha \in \Omega}[M(\alpha)] \leq cn \log w,$$

for some constant c. By Markov's inequality, we have that

$$\Pr_{\alpha \in \Omega}[M(\alpha) \geq (1/\epsilon)cn \log w] \leq \epsilon.$$

Call an initiation configuration to be *good*, if $M(\alpha) \leq (1/\epsilon)cn \log w$. We have that $(1 - \epsilon)$ fraction of the initiation configurations are good.

Fix any good initiation configuration α and consider the behavior of the SemiRand algorithm when α is the initiation configuration. Let us number the N coins generated from 1 to N. Consider the ith coin. After getting initiated at some leaf node, the coin passes through a fixed sequence of nodes positions, before getting deposited at some node. In this process a sequence of coin related messages are exchanged. Let τ_i denote the sequence of nodes that receive these coin related messages Construct a sequence τ by concatenating the sequences for all the N coins; namely, $\tau = \tau_1 \circ \tau_2 \circ \ldots \circ \tau_N$. Let K be the length of τ. The value K represents the number of coin related messages exchanged (across all $\log w$ rounds), when α is the initiation configuration. Since, α is a good initiation configuration, we have that $K = O(n \log w)$. Let M_x denote the number of coin related messages received by a processor x and let $M_{\max}$ denote the maximum of M_x over all processors x.

Now we will relate the SemiRand algorithm to the extended ball game discussed in the previous section. The node positions and the processors correspond to the seats and girls, respectively. We will use K balls and τ will be the ball sequence. Thus, the balls correspond to the messages. Recall that in the SemiRand algorithm after Δ coin related messages are received on a node position, a swap takes place. We will use this same parameter Δ in the extended ball game.

Now, we invoke Theorem 4.3, by setting $t = 2\Delta$. Recall our assumption that $w \geq n$. Since, $K = O(n \log w)$, we get that with probability at least $(1 - 2/n)$ no processor (i.e., girl) receives more than $O(\log w)$ coin related messages. This is true for any good initiation configuration. Recall that at most ϵ fraction of the configurations are bad. We can now apply the union bound and get that with probability at least $(1 - \epsilon - 2/n)$, no processor receives more than $O(\log w)$ coin related messages. Thus, the failure probability is at most 2ϵ. The discussion is summarized by the following result.

Theorem 4.4 *Consider the* SemiRand *algorithm. There exists a constant c such that the following is true. Fix any* $0 < \epsilon < 1$. *Then,*

$$\Pr[M_{\max} \geq (1/\epsilon)4c\Delta \log w] \leq 2\epsilon.$$

Thus with high probability the MaxLoad is $O(\log w)$.

Remark: The parameter Δ used by the SemiRand algorithm is not very critical. We can set it to, say $\Delta = 6$. The value of Δ will be important in the analysis of the OptRand algorithm.

5 OptRand Algorithm

In this section, we describe the OptRand algorithm. This is designed by modifying the SemiRand algorithm of the previous section. Here, we will take into account all types of messages (coin related, poll related and swap related messages) and present a complete analysis. We will show that with high probability, OptRand has message complexity $O(n \log w)$ and MaxLoad $O(\log w)$.

For most part, the OptRand algorithm is similar to the SemiRand algorithm. However, interesting issues arise when we start accounting for swap related messages. For instance, when a node initiates a swap, it sends a swap information message to its neighbors. A node v may receive a lot of such messages even though it does not receive many coin related messages. Such a node may also become a hot-spot. To overcome the issue, we increment the δ counter also for swap related messages at the source end. However, the swap related messages exchanged at the destination end and the leaf polling messages should not increment the δ counters. The reason is that node positions that receive these later messages cannot be derived from the initiation configuration (these are determined by the randomness used in swapping). Only the messages whose node position can be determined from the initiation configuration incremen the δ counters; these will be accounted for using the ball game. A separate analysis will be performed for the other messages.

The pseudocode for the OptRand algorithm is presented in Figures 1, 2 and 3. For the ease of exposition, the pseudocode is presented from a node position perspective. This code is executed by whichever processor occupies the particular node position at any given instance of time. Suppose a processor x occupying a node position u performs a swap with a processor y occupying a node position v. Once the swap happens, the processor y will continue executing the code of node u (from where x left off) and similarly, the processor x will continue executing the code of node v (from where y left off). We will call u the *source node* of the swap and v the *destination node* of the swap.

The *ProcessEvent* procedure is invoked whenever a message is received. This procedure primarily handles unsolicited events. The solicited events (acknowledgements) are handled within the other procedures explicitly.

We start by describing the process of initiating a coin. Recall that when a processor x generates a coin, the coin has to be initiated at some leaf node. We will employ the same hot-potato-like procedure used in the SemiRand algorithm. To do this, the processor x picks a processor y at random and sends the coin to y using a *LeafPoll* message. Upon receiving the *LeafPoll* message, the

```
ProcessEvent(Msg,Sender)
    switch (Msg type)
      LeafPoll: ProcessLeafPoll(Msg,Sender);
      Coin: ProcessCoin(Msg,Sender);
      SwapReq: ProcessSwapReq(Msg,Sender);
      SwapInfoS: ProcessSwapInfoS(Msg,Sender);
      SwapInfoD:
          Update neighbor information from Msg;
    endswitch
```

Figure 1: Main Event Processing Procedure

processor y works as follows. (See *ProcessLeafPoll* procedure). Let v be the node position occupied by y. Then, y checks if v is a leaf node position. If not, it just forwards the message again to another processor picked uniformly at random. If v is indeed a leaf node position, it either consumes the coin or passes the coin to its parent using a *CoinMsg*, depending on whether v already has a coin or not.

The initiated coin is passed from one node to the other using the *Coin* message, until it gets deposited at some node position. When a node receives a *Coin* message, it follows a procedure similar to the CoinPass algorithm. This is shown in the *ProcessCoin* procedure. First consider the case when the node already has a coin. If the coin is received from the parent, the procedure returns back a *CoinReject* to the parent as the subtree is already full of coins. On the other hand if the coin is received from a child, it is simply forwarded to the parent. Now consider the case when the node does not have a coin. In this case, it tries to push the coin down its subtree, and if it fails, it simply consumes the coin. The coin gets deposited at the current node position.

We now describe the process of swapping node positions. Recall that in each node position u, we maintain a counter δ_u to keep track of the number of messages received at u. The procedures *ProcessCoin* and *ProcessLeafPoll* also increment the δ counter. In case the counter crosses the specific threshold Δ, the node initiates a swap procedure (by invoking *InitiateSwap*). Suppose a processor x occupying a node position u initiates a swap. The processor x picks a processor y uniformly at random and sends it a *SwapRequest* message along with the state information of u. Let v be the node position occupied by y. The processor y upon receiving the *SwapRequest*, invokes the *ProcessSwapReq* procedure. The processor y sends out a *SwapInfoD* message to all the three neighbors of v informing them of the new processor x that is taking over the node position v. (Thus, *SwapInfoD* message is used to inform destination-side neighbors about the swap.) The processor y then sends back a *SwapAccept* to x along with state information of v. The processor x upon receiving the *SwapAccept*, copies the state information. Note that at this point (marked by bullets in the *InitiateSwap* and *ProcessSwapReq* procedures), the processors x and y exchange their roles (node positions). The state information for a node position u consists of all the variables that are associated with u such as the identity of the processors that are neighbors of u, the δ_u counter, whether u has a coin or not, etc. Note that the processor x executing the *InitiateSwap* procedure of u, on execution of the statement marked with the bullet would continue execution from the statement marked with the bullet in the *ProcessSwapReq* procedure (of the node position v). Similarly, y suspends executing the code of v and starts executing the code of u. In practice, this can also be encoded as part of the state.

Source-end Swap Processing via Baton Passing: After a swap has successfully taken place, the processor y (now occupying the node position u) sends out a *SwapInfoS* message to the neighbors of u informing them of the processor change. (Thus, the *SwapInfoS* message is used to inform source-side neighbors about the swap). An interesting aspect of the source-end swap processing is that the process can give rise to a chain of more swaps, as discussed next. When the neighbors receive *SwapInfoS* messages, they must increment their δ counters (see procedure *ProcessSwapInfoS*). This may cause the neighbors to hit the Δ threshold and initiate a new swap. In general, a chain of swaps may ensue. However, we would like to avoid simultaneous swaps happening in the system. For this purpose a baton passing mechanism is used. The originator node u now generates a baton. The baton will be passed to other nodes in a sequential manner. Another node can perform a subsequent swap only when it receives the baton. In other words, a neighboring node receiving the *SwapInfoS* message does not immediately initiate swap, even if it hits the threshold Δ. Instead it waits for the *Baton* to come it. Starting at the node u, the baton is passed in a recursive manner. When a node z_1 receives a baton from a node z_2, the node z_1 behaves as follows. The node z_1 checks if its δ counter has hit the Δ threshold. If not, the baton is passed back immediately to the node z_2. On the other hand, suppose the δ counter of z_1 has indeed hit the Δ threshold. In this case, z_1 performs the swap. Then, z_1 informs its neighbors about the swap. Next, the node z_1 will recursively send out the baton by giving the baton to its neighbors. The baton will eventually be returned to z, which it returns back to z_2. This way, the baton will eventually be returned to the originator u, which destroys the baton. (see procedure *ProcessSwapInfoS*). This completes the description of the OptRand algorithm.

Remark 1: Note that while a node is waiting for a message (e.g. in WaitFor(*Baton*)), it may receive a request for a swap. In this case, the node services the swap, i.e., the corresponding processors exchange their node positions. Note that in this case, this node (waiting for *Baton*), is at the receiving end of the swap request and therefore generates *SwapInfoD* messages for its neighbors. This is not a cause for concern since these messages (exchanged at the receiving end of the swap) do not increment the δ counter and therefore, do not cause more swaps.

Remark 2: Note that the *InitiateSwap* procedure does not check δ against the threshold on receiving the baton. This is acceptable since δ has been set to 0 at the start of the procedure and the node may receive the baton back at most 3 times during the execution of this procedure. This can be handled by simply setting Δ to a sufficiently large constant. Also note that the *ProcessSwapInfoS* procedure may perform a swap when $\delta = \Delta + 1$ (instead of $\delta = \Delta$) since it waits for the baton. We will account for this in our analysis.

Remark 3: We now discuss how to set the parameter Δ. The parameter Δ has to large enough to account for the following two phenomena. First, notice that whenever δ_u of a node u hits Δ, u initiates a swap. The process results in generating *SwapInfoD*, *SwapInfoS* and *Baton* messages. The number of such messages is bounded by some constant, for a given swap (discounting the recursive swaps). The parameter Δ has to be large enough so that the number of swaps is minimized; and hence we can get a bound on the total number of messages generated. Secondly, suppose a node u passes the baton to a neighbor. It will immediately reset its counter δ_u to 0. The node u will get back the baton in a *Baton* message. Moreover, a neighbor of u may also perform a swap and send a *SwapInfoS* message to u. For each of the above *Baton* messages and the *SwapInfoS* messages, the node u will increment the counter δ_u. We should ensure that this does not result in δ_u hitting the threshold Δ again. Because, such an occurence may result in the baton being passed indefinitely in the tree. Thus, Δ has to be large enough. Setting Δ to any constant greater than 20 suffices.

Remark 4: As in the case of CoinPass algorithm, when $\hat{w}$ gets below n, we switch to a more straightforward algorithm. This is discussed in Appendix A.

Analysis of OptRand - An Overview:

We will now present an overview of the analysis. The full analysis is described in Appendix D, where Theorem 1.1 is proved. We will classify the messages to a few categories and analyze each category separately.

First consider the *LeafPoll* messages related to the leaf polling process. The number of coins generated is $N = O(n \log w)$. The number of leaf nodes is $n/2$. For each coin, the number of *LeafPoll* messages generated follows the geometric distribution with success probability $p = 1/2$. Let X denote the total number of *LeafPoll* messages for all the N coins put together. Then, X follows the negative binomial distribution. By appealing to a known Chernoff-like bound [12] for the negative binomial distribution, we will show that with high probability, every processor receives only $O(\log w)$ *LeafPoll* messages.

Next consider the messages related to coin passing and swapping. Corollary 3.2 gives us a bound of $O(n \log w)$ for the messages related to the original CoinPass algorithm (namely, *Coin*, *CoinAccept* and *CoinReject*); call these coin-related messages. Regarding the swap process, consider only the messages related to source end of the swap (namely, *SwapInfoS*, *SwapAccept* and *Baton*); call these source-end swap messages. When a node u receives Δ (or $\Delta + 1$) of these two type of messages, it generates some source-end swap messages; a simple accounting shows that the number of these extra messages is at most $\Delta/2$ for our choice of $\Delta = 20$. Thus, we have a process that starts with N messages and for each Δ messages, the process adds at most another $\Delta/2$ messages. We will argue that in any such process, the number of messages can only double. Thus, the total number coin-related and source-end swap messages is $M \leq 2N$. So, $M = O(n \log w)$. We will argue that with high probability, every processor receives about $M/n = O(\log w)$ messages. This will be accomplished by slightly generalizing the extended ball game (of Section 4.1) and analyzing it.

Finally, consider the swap messages related to the destination-end of the swapping (namely, *SwapInfoD* and *SwapRequest*); call these destination-end swap messages. The number of swaps in the algorithm is roughly $M/\Delta = O(n \log w)$. For each such swap, a constant number of destination-end swap messages are generated. So, the total number of destination-end swap messages is $D = O(n \log w)$. For each swap, the destination node is chosen at random. So, each of the above D messages is delivered to a random node (intuitively). By appealing to Chernoff bound, we will argue that with high probability each processor receives about $D/n = O(\log w)$ messages.

6 Open Problems

Our main result (Theorem 1.1) presents a randomized algorithm with MaxLoad $O((1/\epsilon) \log w)$, where ϵ is the failure probability. An interesting open problem is to design an algorithm with improved MaxLoad, with respect to ϵ. Namely, it would be nice to design an algorithm with MaxLoad=$O(\log(1/\epsilon) \log w)$. Going one step further, one can aim for an algorithm with error probability being an inverse polynomial in n; namely, the goal here is to get MaxLoad=$O(d \log w)$ with $\epsilon = 1/n^d$. The ultimate goal of this line of work would be to design a deterministic algorithm with MaxLoad $O(\log w)$. A second line of open problems is concerned with proving lowerbounds for deterministic algorithms. A starting point would be to show that any deterministic algorithm should have MaxLoad $\Omega(\log^2 n)$.

References

[1] Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael E. Saks. Local management of a global resource in a communication network. *J. ACM*, 43(1):1–19, 1996.

[2] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition.* MIT Press, McGraw-Hill Book Company, 2000.

[4] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. In *SODA*, 2008.

[5] Yuval Emek and Amos Korman. Brief announcement: new bounds for the controller problem. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 340–341, New York, NY, USA, 2009. ACM.

[6] R. Garg, V. K. Garg, and Y. Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *20th Int. Conf. on Supercomputing (ICS)*, 2006.

[7] Vijay K. Garg and Joydeep Ghosh. Repeated computation of global functions in a distributed environment. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):823–834, 1994.

[8] L. Huang, M. Garofalakis, A. Joseph, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. In *ICDCS*, 2007.

[9] R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD Conference*, 2006.

[10] Amos Korman and Shay Kutten. Controller and estimator for dynamic networks. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 175–184, New York, NY, USA, 2007. ACM.

[11] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge Univ. Press, 2005.

[12] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, New York, 1993.

[13] Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

[14] Gerard Tel. Distributed infimum approximation. In Lothar Budach, Rais Gatic Bakharajev, and Oleg Borisovic Lipanov, editors, *FCT*, volume 278 of *Lecture Notes in Computer Science*, pages 440–447. Springer, 1987.

A The Case of $\hat{w} < n$

Here, we present a simple algorithm (called PlainTree) to handle the case when $\hat{w}$ gets below n in the CoinPass and OptRand algorithm.

A.1 The PlainTree Algorithm

The n processors are arranged in the form of a binary tree, as in the case of the CoinPass algorithm. The PlainTree algorithm differs from CoinPass algorithm in the following ways. First, a coin is generated for each trigger. Secondly, once the coin is initiated in some leaf node, it passes up the tree without getting deposited anywhere along path, until it reaches the root node; once the coin reaches the root node, it is deposited there. Thus, every coin is deposited at the root node. The root node keeps count of the number of coins received. Once the count reaches $\hat{w}$, the protocol is terminated and the user is sent an alert.

It is easy to see that each coin passes though $O(\log n)$ nodes (the depth of the tree). Thus the message complexity is $O(\hat{w} \log n) = O(n \log n)$.

A.2 Applying PlainTree Algorithm within the OptRand Algorithm

Consider the OptRand algorithm and the scenario where $\hat{w}$ gets below n. In this case, we switch to the PlainTree algorithm. However, we modify the PlainTree algorithm to include swapping. Namely, a δ counter is maintained with each node position. When the δ counter of a node position u hits the threshold Δ, the processor x occupying u picks a processor y random and the two swap their positions.

The analysis of PlainTree algorithm with swapping is very similar to the overall analysis of the OptRand algorithm. It can be shown that the MaxLoad is $O(n \log n)$.

B Analysis of the Expectation

In this section, we analyze the expected number of messages exchanged in the CoinPass algorithm and prove Theorem 3.1. The proof goes as follows. We first introduce a *bottom-up* random process that captures the essence of the CoinPass algorithm. We will derive an upperbound on the cost of incurred in this process. However, the bottom-up process is cumbersome to analyze directly. So, we shall introduce an easy-to-analyze *top-down* random process and derive an upperbound on the cost incurred by this process. We will show that the two processes are equivalent and their costs differ only by a factor of two. Theorem 3.1 is proved putting together the above pieces.

B.1 Proof of Theorem 3.1

Let $\ell = \log((n+1)/2$. The tree has 2^ℓ leaf nodes. Recall that the signature sig_u of a leaf node u is an ℓ-bit string. Let $\sigma = s_1, s_2, \ldots, s_n$ be a sequence of n signatures, where each $s_i \in \{0,1\}^\ell$; we call σ an *initiation sequence*. In other words, an initiation sequence specifies a sequence of leaf nodes represented by their signatures.

Bottom-up Process: Given an initiation sequence σ, the bottom-up processes works as follows. It considers each signature s_i and simulates the CoinPass algorithm by initiating a new coin at at the leaf node u whose signature is s_i. After getting initiated at the node u, the coin passes through a sequence of internal nodes, before getting deposited at some node v. The *bottom-up cost* of the ith coin is defined to be the number of internal nodes in the above sequence (including u and v). The *bottom-up cost* of the sequence σ is the sum of the bottom-up costs of the n coins. Let $\mathrm{BU}_\ell(\sigma)$ denote the bottom-up cost of the sequence σ.

Choose an initiation sequence S uniformly at random (namely, each signature s_i is chosen independently and uniformly at random). Let $B(n) = \mathbf{E}[\mathrm{BU}_\ell(S)]$ denote the expected bottom-up cost of the random sequence S. Thus, $B(n)$ is a function of n, where n denotes both the number of nodes in the tree and number of coins initiated.

Now consider the CoinPass algorithm. In any round, exactly n coins are generated. These are then initiated at n leaf nodes, which are chosen at random uniformly and independently (The nodes where these coins get generated is determined by the given trigger pattern. However, where these coins get initiated is random). The coins pass through the internal nodes; passing a coin from one node to another can be accomplished using a constant number of messages. Thus, the expected number of messages is each round is $O(B(n))$.

Proposition B.1 *Consider the* CoinPass *algorithm. In each round, the expected number of messages is at most $cB(n)$, for some constant c, i.e., the expected number of messages in $O(B(n))$.*

By the above proposition, it suffices to derive an upperbound on $B(n)$. We now introduce the top-down process and then show that it is equivalent to the bottom-up process. In the bottom-up process, the path travelled by coins start at the leaf level. The top-down processes will do the opposite: the paths will start at the root level and go down the tree. We will design the top-down process in such a way that the two processes are equivalent. Meaning, for any initiation sequence σ, the nodes where each coin gets deposited will be the same in both the process. Furthermore, we will define the top-down cost of σ in such a way that the bottom-up cost of σ is at most twice its top-down cost. This way, deriving an upperbound on the top-down costs will gives an upperbound on the bottom-up costs. The top-down process is defined next.

Top-down Process: Given an initiation sequence $\sigma = s_1, s_2, \ldots, s_n$, the process considers each sequence s_i and works as follows. We generate a new coin and give it to the root node. The coin will pass through the nodes and get deposited at some node of the tree. Suppose a node u gets the coin. If both the children of u have coins, the coin gets deposited at u. If only one of the children of u has the coin, the coin is given to the node that does not have a coin. If both the children of u do not have coin: compute the bit $b = s_i[\mathit{level}(u)]$. If $b = 0$, give the coin to the left child; if $b = 1$, give the coin to the right child. If $b = 0$, we say that the left-child is the *sig-specified* child at u; similarly, if $b = 1$, we say that the right child is the *sig-specified* child of u. Notice that the coin may or may not be given to the sig-specified child. If the coin is given to the child which is not sig-specified, we say that the path of the coin *bends* at u. This completes the description of the process. The top-down cost of the ith coin is defined as follows. As the coin passes down the tree, if the path does not bend at any node, then its cost is defined to be $\ell - \mathit{level}[u] + 1$, where u is node where it got deposited. On the other hand, suppose the path bends at some node. Let u be the first node where such a bending takes place. Then, the cost is $\ell - \mathit{level}[u] + 1$. The top-down cost of the sequence σ is defined to be the sum of the top-down costs of all the n coins. Let $\mathrm{TD}_\ell(\sigma)$ denote the top-down cost of σ.

Choose an initiation sequence S uniformly at random. Let $T(n) = \mathbf{E}[\mathrm{TD}_\ell(S)]$ denote the expected top-down cost of the random sequence S. Thus, $T(n)$ is a function of n, where n denotes both the number of nodes in the tree and number of coins initiated.

It is not hard to see that the two processes are equivalent. Namely, for any initiation sequence σ, the node where each coin gets deposited is the same in both the processes. Furthermore, the bottom-up cost of σ is at most twice of the top-down cost of σ. This can be seen as follows. A coin can get deposited in two ways. We will analyze both the cases. Consider a coin and let z be the node where it got initiated. In the bottom-up process, a coin travels up the tree and reaches a node u that does not have a coin.

- Suppose both children of u have coins. In this case, the coin gets deposited at u. The bottom-up cost of the coin is $\ell - \mathit{level}[u] + 1$ (the distance between node u and the node z). Now, consider the top-down process: the coin will get deposited at the same node u and will incur the same cost.

- Suppose some children of u does not have a coin. In this case, the coin will be passed down to a coin-less child of u and get deposited at some node v. The bottom-up cost of the coin is the distance from z to u plus the distance from u to v; namely, the cost is $d_1 + d_2$, where $d_1 = \ell - \mathit{level}[u] + 1$ and $d_2 = \mathit{level}[v] - \mathit{level}[u]$. Notice that $d_2 \leq d_1$ and so the bottom-up cost is at most $2d_1$. Now consider the top-down process. The path of the coin will start from the root and reach the node u, without bending anywhere. Then, for the first time, it will bend at u. Next, proceeding further down the tree, the coin will reach v. The top-down cost is exactly d_1. We see that the bottom-up cost is at most twice the top-down cost.

We will now formally show that the two processes are equivalent. In order to formalize the claim, we define the notion of a *coin configuration.* A coin configuration simply specifies the subset of nodes on which coins are present. A coin configuration is said to be *proper*, if the following is true: for any node u, if u possesses a coin then both the children of u posses coins. If every node possesses a coin, then the configuration is said to be *full*; similarly, if no node possesses a coin the configuration is said to be *empty.* Let $\sigma = s_1, s_2, \ldots, s_n$ be an initiation sequence. Consider executing the top-down processes on this sequence. To start with, we have an empty configuration and processing each signature produces a new configuration with one more coin. Let C_0 be the empty initial configuration and let C_i denote the configuration produced after processing the ith coin. Notice that C_n is the full configuration. The sequence $C_0, C_1, \ldots, C_n$ is called the top-down configuration sequence of S. The notion of bottom-up configuration sequence is defined similarly. We are now ready to formally state the equivalence claim. The lemma is proved in Section B.2

Lemma B.2 *For any initiation sequence S, the top-down and bottom-up configuration sequences are the same. Moreover, the bottom-up costs of S is at most twice the top-down cost of S, i.e.,* $\mathrm{BU}_\ell(S) \leq 2\mathrm{TD}_\ell(S)$.

The lemma below follows immediately.

Lemma B.3 $B(n) \leq 2T(n)$.

Now, we derive the expected cost for the top-down process and prove the following result.

Lemma B.4 $T(n) = O(n)$.

Proof: We will prove the lemma by establishing a recurrence relation for $T(n)$ in terms of $T((n-1)/2)$. Let L and R denote the left and right subtrees (i.e., the subtrees rooted at the left and right children of the root node). For a signature $s \in \{0,1\}^\ell$, let $\mathit{trunc}(s)$ be the string obtained by deleting the first bit of s; thus, s is a signature string at length $\ell - 1$. We say that a signature $s \in \{0,1\}^\ell$ is *left-going*, if starts with bit 0; it is said to *right-going*, if it starts with bit 1. Let $\sigma = s_1, s_2, \ldots, s_n$ be any initiation sequence at length ℓ. Consider the ith signature s_i and the coin corresponding to it. Suppose s_i is left-going. Let u be the node where the ith coin gets deposited. The node u can also be determined by simulating the top-down process on the left-subtree L with $\mathit{trunc}(s_i)$ as the input signature. The cost incurred by the ith coin in the original tree is the same as the cost incurred by the coin the latter process. A similar claim is true for right-going signatures. The above claims are true, until the left and the right subtree do not become full. Now, suppose the left subtree gets full (i.e., all nodes in the subtree have coins). Consider a subsequent signature s_i, which is left-going. Then, the path of the corresponding coin will bend at the root node itself.

After that, the coin will travel through the right subtree and get deposited in some node v in the right subtree. Observe that the node v can be determined by simulating the top-down process on the right subtree with $\mathit{trunc}(s_i)$ as the input signature. The cost of the coin is $\ell + 1 = \log(n+1)$, since bending happens at the root node itself. A similar scenario happens if the right subtree gets full.

To formalize the discussion so far, let us construct two initiation sequences σ_1 and σ_2 as follows. Ignore the last signature s_n of σ. Scan through the first $(n-1)$ signatures in the sequence σ. For each signature s_i: if s_i is left-going, add it to σ_1; if s_i is right-going, add it σ_2. However, we will ensure that neither σ_1 nor σ_2 gets more than $(n-1)/2$ signatures. Namely, once σ_L gets $(n-1)/2$ signatures, the subsequent signatures are always added to σ_2; similarly, once σ_R gets $(n-1)2$ signatures, then the subsequent signatures are added to σ_1. The sequences σ_1 and σ_2 are initiation sequences of length $(n-1)/2$ and their constituent signatures of length $\ell - 1$. We will call the above process as truncation. We say that a left-going signature is *bounced*, if it gets added to σ_2; similarly, a right-going signature is *bounced*, if it gets added to σ_1. Let $\text{Bounce}(\sigma)$ denote the number signatures that are bounced. Then, by our discussion above, we get that

$$\begin{aligned}
\text{TD}_\ell(\sigma) &\leq \text{TD}_{\ell-1}(\sigma_1) + \text{TD}_{\ell-1}(\sigma_2) \\
&\quad + (\ell+1)\cdot \text{Bounce}(\sigma) + (\ell+1) \\
&= \text{TD}_{\ell-1}(\sigma_1) + \text{TD}_{\ell-1}(\sigma_2) \\
&\quad + (\ell+1)\cdot [1 + \text{Bounce}(\sigma)]
\end{aligned} \tag{1}$$

Any bouncing signature incurs a cost of $(\ell+1)$ and hence, we have the term $(\ell+1)B(\sigma)$. An extra $(\ell+1)$ is added to account for the cost of the last signature s_n (whose coin always gets deposited at the root node). We get only an inequality, since the RHS is not a tight analysis of the scenario: for a left-going coin which got bounced, we have accounted for its cost in the term $(\ell+1)\cdot\text{Bounce}(S)$; additionally, $\text{TD}_{\ell-1}(\sigma_2)$ will also add a cost for this coin. (Similarly, the RHS includes extra costs for right-going coins which get bounced).

Choose an initiation sequence S of length n at random. Let $X = \text{TD}_\ell(S)$ be the random variable denoting the top-down cost of S. Then, $T(n) = \mathbf{E}[X]$ is the expectation of X. Let S_1 and S_2 be the sequences constructed by applying the truncation process to S. Let X_1 and X_2 be random variables denoting $\text{TD}_{\ell-1}(S_1)$ and $\text{TD}_{\ell-1}(S_2)$, respectively. Let $B = \text{Bounce}(S)$ be the random variable denoting the number of bounced signatures in S. Notice that S_1 and S_2 are random initiation sequences of length $(n-1)/2$. We have $\mathbf{E}[X_1] = T((n-1)/2)$ and $\mathbf{E}[X_2] = T((n-1)/2)$.

From Equation 1, we get that

$$X \leq X_1 + X_2 + (\ell+1)(1+B).$$

By the standard properties of expectation, we have hat

$$\mathbf{E}[X] \leq \mathbf{E}[X_1] + \mathbf{E}[X_2] + (\ell+1)(1+\mathbf{E}[B]).$$

It follows that

$$\begin{aligned}
T(n) &= \mathbf{E}[X] \\
&\leq 2T\left(\frac{n-1}{2}\right) + (\ell+1)(1+\mathbf{E}[B])
\end{aligned} \tag{2}$$

We now derive a bound for $\mathbf{E}[B]$. The random variable B can be viewed in a different way by considering the following coin tossing experiment. Toss a fair coin n' times independently, where $n' = n - 1$. Let H and T be the random variables denoting the number of heads and tails, respectively. Both have expectation $n'/2$. Let Z be a random variable denoting the expected deviation; namely, define $Z = |H - (n'/2)|$ (equivalently, $Z = |T - n'/2|$). Then, B and Z have the same distribution. Thus, $\mathbf{E}[B] = \mathbf{E}[Z]$. The random variable H follows the binomial distribution. Its expectation is $\mu_H = n'/2$; its variance is $\sigma_H^2 = n'/4$; its standard deviation is $\sigma_H = \sqrt{n'}/2$. The random variable Z can be rewritten as $Z = |H - \mu_H|$. It is a well known fact that for any random variable Y,

$$\mathbf{E}[|Y - \mu_Y|] \leq \sigma_Y,$$

where μ_Y and σ_Y are the expectation and standard deviation of Y, respectively. Applying the above fact to the random variable Z, we get that

$$\mathbf{E}[B] = \mathbf{E}[Z] \leq \sqrt{n'}/2.$$

Applying the above bound and the value of ℓ in Equation 2, we get that

$$T(n) \leq 2T\left(\frac{n-1}{2}\right) + \log(n+1) \cdot \left[1 + \frac{\sqrt{n}}{2}\right] \tag{3}$$

It is not hard to solve the above recurrence relation; for instance, we can appeal to the master theorem of recurrence relations [3]. We can show that $T(n) \leq cn$, for some constant c. $\square$

We get Theorem 3.1 by combining Proposition B.1, Lemma B.3 and Lemma B.4.

B.2 Proof of Lemma B.2

The proof is by induction over the length of the initiation sequence S.

Clearly, for $S = \phi$, the configuration sequences C_0, C'_0 are the same as they are both the empty configurations. Moreover $\mathrm{BU}_\ell(S) = \mathrm{TD}_\ell(S) = 0$.

Now, suppose the claim holds for any given configuration sequence of length k, Then we show that it also holds for any configuration sequence of length $k+1$. Let the initiation sequence be $S = s_1, s_2, \ldots, s_k, s_{k+1}$. Let $C_0, C_1, \ldots, C_{k+1}$ be the configuration sequence for the bottom-up process after processing the first k coins (signatures). Similarly, let $C'_0, C'_1, \ldots, C'_{k+1}$ be the configuration sequence for the top-down process after processing the first k coins (signatures). Let $S_k = s_1, s_2, \ldots, s_k$ be the initiation sequence consisting of the first k signatures of S. By induction, after processing the first $k-1$ coins, the configuration sequences are same. Therefore, $C_k = C'_k$. Now let us analyze where the k^{th} coin settles down for the bottom-up and the top-down processes. Let sig be the signature of the k^{th} coin. Note that the bits of sig define a unique path from the root (say r) to a leaf node (say u) wherein we proceed to the left (resp. right) child at level i if $\mathrm{sig}[i] = 0$ ($\mathrm{sig}[i] = 1$ resp.). Note that in case of the bottom-up process, this leaf node is the initiating node.

First we consider the case where in the bottom-up process, the coin is passed from u upwards and settles down on some node, say v, without going down. Clearly in C_k, v cannot have a coin and both children of v must already have coins on them. Moreover, in C_k, no ancestor of v on the path from v to the root can have a coin on it – otherwise it would violate the invariant that a node with a coin cannot have a coinless child. Note that the top-down process follows the same path specified by signature until it is either forced to bend or it settles down. Since C'_k is same as C_k by induction, we have that v is coinless and both its children have coins. Therefore the coin must come and settle at v. It is easy to see that the cost incurred in both the processes is the same $\ell - level[v] + 1$. By induction, we have that $\mathrm{BU}_\ell(S_k) \leq 2\mathrm{TD}_\ell(S_k)$. Therefore,

$$\begin{aligned}\mathrm{BU}_\ell(S) &= \mathrm{BU}_\ell(S_k) + (\ell - level[v] + 1)\\ &\leq 2\mathrm{TD}_\ell(S_k) + (\ell - level[v] + 1)\\ &\leq 2\mathrm{TD}_\ell(S).\end{aligned}$$

Next we consider the case where in the bottom-up process, the coin is passed from u upwards until some node, say v, and is then passed downwards until it finally settles at some node, say w. Clearly in C_k, v does not have a coin, the child of v on the path towards u has a coin and the other child of v does not have a coin. Moreover no ancestor of v has a coin, as before. The cost incurred is $(\ell - level[v] + 1) + (level[w] - level[v] + 1) \leq 2(\ell - level[v] + 1)$ since u is a leaf node and has maximum depth. Let us now analyze the top-down process for the coin in this case. Again, the top-down process follows the same path specified by signature until it is either forced to bend or it settles down. Since C'_k is same as C_k by induction, we observe that the coin will be first forced to bend at v. Note that when a coin goes downwards in the bottom-up process, it uses the same algorithm as the top-down process. Therefore, after bending from v, the coin will follow the same path as followed by the coin from v in the bottom-up process. Therefore, the coin will settle down at the same node w as in the bottom-up process. The cost incurred in the top-down process is $(\ell - level[v] + 1)$ as that is the first bending node in its path. By induction, we have that $\mathrm{BU}_\ell(S_k) \leq 2\mathrm{TD}_\ell(S_k)$. Therefore,

$$\begin{aligned}\mathrm{BU}_\ell(S) &\leq \mathrm{BU}_\ell(S_k) + 2(\ell + 1 - level[v])\\ &\leq 2\mathrm{TD}_\ell(S_k) + 2(\ell - level[v] + 1)\\ &\leq 2\mathrm{TD}_\ell(S)\end{aligned}$$

C Analyzing the Ball Games

In this section, we first analyze the basic ball game (with $\Delta = 1$) and prove Theorems 4.1 and 4.2. Then, we shall analyze the extended ball game (with arbitrary Δ) and prove Theorem 4.3.

C.1 Proof of Theorem 4.1 and 4.2

We will first prove Theorem 4.1. We will do this by directly counting the number of possibilities that could lead to the event of interest.

Let $R_1, R_2, \ldots, R_K$ be random variables that denote the girls chosen at random in the K rounds. Let $\tau = b_1, b_2, \ldots, b_K$ be the ball sequence fixed by the adversary. Then, we have $Y_i = \pi_{i-1}(b_i)$. The random variable π_i is obtained from π_{i-1} by swapping the position of the girls Y_i and R_i in π_{i-1}. The relationship between these variables is shown in Figure **??**.

Define a function $swap : \Pi \times S \times G \to \Pi$ that captures the swapping process of the game. Given a seating arrangement σ, a seat s and a girl g, we swap the positions of the girls g' and g, where $g' = \sigma(s)$ is the girl occupying the seat s under σ; the resulting seating arrangement $\bar{\sigma}$ is the output of $swap(\sigma, s, g)$.

Claim C.1 *Fix any seat $\bar{s} \in S$ and any seating arrangement $\bar{\sigma}$. There are exactly n* good *pairs $\langle \sigma, g \rangle$ satisfying: $swap(\sigma, \bar{s}, g) = \bar{\sigma}$.*

Proof: Let $g' = \bar{\sigma}(\bar{s})$. Imagine that we are trying to construct a good pair $\langle\sigma, g\rangle$ and we will count the number of ways in which we can accomplish this task. In σ, we can make any one of the n girls to sit in seat $\bar{s}$. For $1 \leq i \leq n$, consider the option of setting $\sigma(\bar{s}) = g_i$. In this case we must do the following: (i) set $g = g'$; (ii) set $\sigma(s') = g'$, where s' is the seat occupied by g_i in $\bar{\sigma}$; (iii) omitting s' and $\bar{s}$, for all the other seats, use the same seating arrangement as $\bar{\sigma}$. See Figure 4. Notice that the pair (σ, g) constructed above is a good pair and that this is the only way of constructing such a pair, given that we have decided to set $\sigma(\bar{s}) = g_i$. We see that we can construct exactly n good pairs. □

Claim C.2 *Fix any seat $\bar{s} \in S$, any seating arrangement $\bar{\sigma}$ and any girl $\bar{g}$. There exists exactly one* good *pair $\langle\sigma, g\rangle$ satisfying: swap$(\sigma, \bar{s}, g) = \bar{\sigma}$ and $\sigma(\bar{s}) = \bar{g}$.*

Proof: Let $g' = \bar{\sigma}(\bar{s})$. Imagine that we are trying to construct a good pair $\langle\sigma, g\rangle$ and we will count the number of ways in which we can accomplish this task. We must do the following: (i) set $g = g'$; (ii) set $\sigma(\bar{s}) = \bar{g}$; (iii) set $\sigma(s') = g'$, where s' is the seat occupied by $\bar{g}$ in $\bar{\sigma}$; (iv) omitting s' and $\bar{s}$, for all the other seats, use the same seating arrangement as $\bar{\sigma}$. See Figure 5. Notice the pair $\langle\sigma, g\rangle$ is a good pair and that this is the only way to construct a good pair. □

Now we show that each π_{i-1} is uniformly distributed.

Claim C.3 *For any $0 \leq i \leq K-1$, the random variable π_i is uniformly distributed; that is, for any σ, $\Pr[\pi_i = \sigma] = 1/n!$.*

Proof: The claim is proved by induction on i. The claim is trivially true for $i = 0$, since π_0 is chosen uniformly at random. Now, consider π_i. By induction hypothesis π_{i-1} is uniformly distributed. Recall that R_i is a random variable denoting the random girl chosen at round i. The pair $\langle\pi_{i-1}, R_i\rangle$ has $n \times n!$ possibilities and each of these possibilities occur with the equal probability of $1/(n \times n!)$. By Claim C.1, of these possibilities, exactly n of them can lead to $\pi_i = \sigma$. Thus,

$$\Pr[\pi_i = \sigma] = \frac{n}{n \times n!} = \frac{1}{n!}.$$

□

It follows immediately that $Y_1, Y_2, \ldots Y_K$ are all uniformly distributed.

Claim C.4 *For any $1 \leq i \leq K$, the random variable Y_i is uniformly distributed; that is, for any girl g, $\Pr[Y_i = g] = 1/n$.*

Proof: We get $Y_i = g$, if $\pi_{i-1}(b_i) = g$. The random variable π_{i-1} has $n!$ possibilities. Of these possibilities, exactly $(n-1)!$ possibilities σ satisfy $\sigma(b_i) = g$. By Claim C.3, π_{i-1} takes each of these $(n-1)!$ possibilities with probability $1/n!$. It follows that $\Pr[Y_i = g] = (n-1)!/n! = 1/n$. □

Now we will prove an independence claim for π_{K-1}.

Claim C.5 *The random variables π_{K-1} and $Y_1, Y_2, \ldots, Y_{K-1}$ are mutually independent: fix any $\bar{\sigma} \in \Pi$ and $a_1, a_2, \ldots, a_{K-1} \in G$. Then,*

$$\Pr\left[\begin{array}{c} \pi_{K-1} = \bar{\sigma} \text{ and} \\ \wedge_{i=1}^{K-1}(Y_i = a_i) \end{array}\right] = \frac{1}{(n!)n^{K-1}}$$

Proof: Refer to Figure **??**. Notice that the entire random process is driven by the choice of π_0 and $R_1, R_2, \ldots R_{K-1}$. The number of possibilities is $(n!) \times n^{K-1}$. Each of these possibilities is taken with equal probability. Let us count the number of possibilities under which we can get $\pi_{K-1} = \bar{\sigma}$ and $Y_1 = a_1, Y_2 = a_2, \ldots, Y_K = a_k$. Let $\sigma_{K-1} = \bar{\sigma}$. We will appeal to Claim C.2 repeatedly. By the claim, there exists exactly one good pair $\langle \sigma, g \rangle$ such that $\mathit{swap}(\sigma, b_{K-1}, g) = \sigma_{K-1}$ and $\sigma(b_{K-1}) = a_{K-1}$. Let $\langle \sigma_{K-2}, \bar{g}_{K-1} \rangle = \langle \sigma, g \rangle$. Thus, to get $\pi_{K-1} = \sigma_{K-1}$ and $Y_{K-1} = a_{K-1}$, we must have $\pi_{K-2} = \sigma_{K-2}$ and $R_{K-1} = \bar{g}_{K-1}$. Thus, to get $\pi_{K-1} = \sigma_{K-1}$ and $Y_{K-1} = a_{K-1}$, there is only one possibility for π_{K-2} and R_{K-1}. Let us apply the claim and the same argument again. We get that, to get $\pi_{K-2} = \sigma_{K-2}$ and $Y_{K-2} = a_{K-2}$, there is only one possibility for π_{K-3} and R_{K-2}. Continuing backwards, we conclude that there is only one possible way of setting π_0 and $R_1, R_2, \ldots, R_{K-1}$. The claim is proved. □

Claim C.6 *The random variable $Y_1, Y_2, \ldots, Y_K$ are mutually independent. That is, for any $a_1, a_2, \ldots, a_K$,*

$$\Pr[(Y_1 = a_1) \wedge \cdots \wedge (Y_K = a_K)] = \frac{1}{n^K}.$$

Proof: Let Π' denote the set of all seating arrangements σ such that $\sigma(b_K) = a_K$. The cardinality of Π' is exactly $(n-1)!$. We get $Y_K = a_K$, if $\pi_{i-1} \in \Pi'$. By Claim C.5, for each $\sigma \in \Pi'$, the probability that $\pi_{i-1} = \sigma$ and $Y_1 = a_1$, $Y_2 = a_2$, …, and $Y_{K-1} = a_{K-1}$ is $1/((n!)n^{K-1})$. Summing up over all $\sigma \in \Pi'$, we get that

$$\Pr[(Y_1 = a_1) \wedge \cdots \wedge (Y_K = a_K)] = \frac{(n-1)!}{(n!)n^{K-1}} = \frac{1}{n^K}.$$

□

Theorem 4.1 follows from Claims C.4 and C.6. We will now prove Theorem 4.2. We will use the following version of the Chernoff bound [11].

Theorem C.7 (see [11], Theorem 4.4) *Let $X_1, X_2, \ldots, X_K$ be 0-1 independent random variables with $\Pr[X_i = 1] = p_i$ and let X be their sum. Let $\mu = \mathbf{E}[X] = \sum_i p_i$ be the expectation of X. Then, for any $r \geq 6$,*

$$\Pr[X \geq r\mu] \leq 2^{-r\mu}.$$

Moreover, for any $\mu' \geq \mu$, the inequality is true, if we replace μ by μ' in both LHS and RHS.

We will now prove Theorem 4.2. Fix any girl $g \in G$. Let $Z_1, Z_2, \ldots, Z_K$ be 0-1 random variables such that $Z_i = 1$, if g gets the ith ball. By Theorem 4.1, for any i, $\Pr[Z_i = 1] = 1/n$. Moreover, the same theorem shows that these random variables are independent. Recall that B_g is a random variable denoting the number of balls received by a girl g. Then, B_g is the sum of $Z_1, Z_2, \ldots, Z_K$. Define $\mu = \mathbf{E}[B_g] = K/n$. Then, by the Chernoff bound (Theorem C.7), we have that

$$\Pr[B_g \geq t(K/n)] =\leq 2^{-\frac{tK}{n}}$$

Theorem 4.2 is obtained by applying the union bound over all the girls. The "moreover" part of the theorem can be derived by applying the Chernoff bound by setting the parameter r is a slightly different manner.

C.2 Proof of Theorem 4.3

In this section, we analyze the extended ball game and prove Theorem 4.3.

Let P and Q denote the number of golden and white balls in the ball sequence τ, respectively. So, $P+Q=K$. Let S denote the number of swaps that happened in the game. For a girl g, let P_g and Q_g represent the number of golden and white balls received by g, respectively; let $B_g = A_g + B_g$ denote the total number of balls received by g. Our goal is derive an upperbound on B_g. When a girl g receives a golden ball, she picks a girl g' at random and the two swap positions. We say that g *initiates* the swap and g' *receives* the swap. For a girl g, let S_g denote the number of swaps in which g participated; let S_g^1 and S_g^2 denote the number of swaps initiated and received by g, respectively; so, $S_g = S_g^1 + S_g^2$.

Consider a girl g. Notice that between any two swaps in which g participates, she receives at most Δ balls (golden and white balls together). Thus,

$$B_g \leq (S_g + 1)\Delta. \tag{4}$$

So, it suffices to derive an upperbound on S_g. We will derive upperbounds on S_g^1 and S_g^2 separately.

Consider any seat s and its occurences in the sequence τ. Notice that for every golden ball, there are at least $(\Delta - 1)$ white balls preceding it. Thus, $Q \geq (\Delta-1)P$. Since $P+Q=K$, we get that

$$P \leq \frac{K}{\Delta}. \tag{5}$$

Construct a sequence $\bar{\tau}$ by projecting only the golden balls in τ. Namely, scan through τ and delete any white balls; the resulting sequence is denoted $\bar{\tau}$. Length of $\bar{\tau}$ is P. Imagine playing the basic ball game on the sequence $\bar{\tau}$ by performing swaps for every ball. Clearly, Theorem 4.2 applies to this scenario. From the proof of the above theorem, we see that the random variable P_g follows the binomial distribution with expectation $\mu = P/n$. Applying Equation 5 and Chernoff bound (Theorem C.7), we get that for any girl g

$$\Pr\left[P_g \geq \frac{tK}{\Delta n}\right] \leq 2^{-\frac{tK}{\Delta n}}$$

The girl g initiates a swap for every golden ball she receives and so $S_g^1 = P_g$. Thus, we get that for any girl g,

$$\Pr\left[S_g^1 \geq \frac{tK}{\Delta n}\right] \leq 2^{-\frac{tK}{\Delta n}} \tag{6}$$

A swap is initiated for every golden ball. Thus the number of swaps is $S = P$. For a girl g and a particular swap, the probability that g is at the receiving end of the swap is $1/n$. So, each girl is expected to receive P/n swaps. By invoking the Chernoff bound (Theorem C.7) and Equation 5, we derive that

$$\Pr\left[S_g^2 \geq \frac{tK}{\Delta n}\right] \leq 2^{-\frac{tK}{\Delta n}} \tag{7}$$

Consider any girl g. Notice that $S_g = S_g^1 + S_g^2$. So, Equations 6 and 7 imply that

$$\Pr\left[S_g \geq \frac{2tK}{\Delta n}\right] \leq 2 \times 2^{-\frac{tK}{\Delta n}}$$

From the above inequality and Equation 4, we get that

$$\Pr\left[B_g \geq \frac{2tK}{n}\right] \leq 2 \times 2^{-\frac{tK}{\Delta n}}$$

Theorem 4.3 can now be derived by appealing to the union bound. The "moreover" part of the theorem can be derived by applying the Chernoff bound by setting the parameter r is a slightly different manner.

D Analysis of OptRand: Proof of Theorem 1.1

We will classify the messages into a few categories and analyze each category separately. The categories are as follows:

1. Leaf Polling Messages: This category contains only the *LeafPoll* message.
2. Main Messages: This category contains *Coin* message and the two source-send swap related messages, namely *SwapInfoS* and *Baton*.
3. Destination-end Swap Messages: This category contains *SwapInfoD* and *SwapRequest*.
4. Miscellaneous Messages: This category contains *SwapAccept*, *CoinAccept* and *CoinReject*.

The reader can verify that we have accounted for all the different types of messages. The main messages category includes the ones that increment the δ counter. The main messages category is named so, because, in comparison to the other three categories, this one is harder to analyze.

Let M^* be a random variable denoting the total number messages exchanged in the system. Let M^L, M^m, M^D and M^O be the random variables that denote that the number of leaf polling messages, main messages, destination-end swap messages and miscellaneous messages exchanged in the system, respectively. Thus, $M^* = M^L + M^m + M^D + M^O$. We will show that the expectation of M^L, M^m, M^D and M^O are all $O(n \log w)$. This would show that the expected message complexity is $O(n \log w)$.

Let M_x^* denote the total number of messages received by a processor x. For a processor x, let M_x^L, M_x^m, M_x^D, and M_x^O denote the number of messages received by x, in each of the four categories. Let $M_{\max}^L$, $M_{\max}^m$, $M_{\max}^D$ and $M_{\max}^O$ denote the maximum over all processors x, the number of messages received by x in the four categories.

We have that MaxLoad is the maximum of M_x^*, over all processors x. Notice that MaxLoad is at most $M_{\max}^L + M_{\max}^m + M_{\max}^D + M_{\max}^O$. We will analyze each category separately and derive a bound for $M_{\max}^L$, $M_{\max}^m$, $M_{\max}^D$ and $M_{\max}^O$. We will show that with probability at least $(1-\epsilon)$ each of the above quantities is bounded by $O((1/\epsilon)\log w)$. It would follows that MaxLoad has the same property.

D.1 Leaf Polling Messages

The number of coins generated is $N \leq n \log w$. Consider the ith coin. During the leaf polling process, the coin is passed through a sequence of processors, before it lands on a processor occupying a leaf node position. Let us call passing the coin from one node to another node as a trial. Each trial results in one *LeafPoll* message being sent. A trial is successful when the processor receiving the coin is occupying a leaf node position. The number of leaf nodes $L = (n+1)/2$. The probability that a trial is successful is $p = (n+1)/(2n) \geq 1/2$. Let X_i be a random variable denoting the number of trials conducted for the ith coin. This random variable follows the geometric distribution with success probability $p \geq 1/2$.

Let $P = \sum_i X_i$ be the total number of trials for all the N coins. The random variable P follows the negative binomial distribution. The expectation of P is $\mathbf{E}[P] = N/p \leq 2N$ For each trial, a *Leaf Poll* message is generated. So, $M^L = P$ and the result below follows.

Lemma D.1 $\mathbf{E}[M^L] \leq 2n \log w$.

We appeal to a known Chernoff-like bound for negative binomial distributions (see [12], Section A.1.2) and derive the following result.

$$\Pr[P \geq 6n \log w] \leq 2^{-n \log w} \leq 1/(2n). \tag{8}$$

Consider the case where $P \leq 6n \log w$. In each of these P trials a randomly chosen processor receives a *LeafPoll* message. Thus, we expect that each processor receives about P/n of these messages. Appealing to the Chernoff bound (Theorem C.7) and the union bound, we get the following claim:

$$\Pr[M^L_{\max} \geq 36 \log w] \leq n 2^{-36 \log w} \leq 1/(2n), \tag{9}$$

provided $P \leq 6N$. (To get the last inequality, recall our assumption that $w \geq n$).

Combining Equations 8 and 9, we get the following lemma.

Lemma D.2 *There exists a constant c such that*

$$\Pr[M^L_{\max} \geq 36 \log w] \leq 1/n.$$

D.2 Container Game

Here, we describe a ball game (called the container game) and present its simple analysis. This game is useful in analyzing the main messages category.

The game is driven two input-specified parameters Δ and λ, with $\lambda \leq \Delta/2$. We have a container with N balls. Th game proceeds in multiple iterations. In each iteration, we pick Δ balls from the container and drop them outside the container; then, we add λ new balls into the container. The game stops when the number of balls in the container gets below Δ. Let N' denote the total number of balls (those outside the container, plus the ones inside the container), when the game ends. Our goal is to derive a bound on N' in terms of N and Δ.

Fix any Δ. Let $B(N)$ denote the total number of balls at the end of the game, if we start with N balls. We can write a recurrence relation for $B(n)$. In a single iteration, we remove Δ balls from the container and add λ balls. So, the number of balls in the container after the iteration is $N - (\Delta - \lambda)$. This later quantity is at most $N - (\Delta/2)$, since we assumed $\lambda \leq \Delta/2$. On the other hand, Δ balls gets added outside the container (because, we drop outside, the Δ balls removed from the container). This leads to the following recurrence relation.

$$B(N) \leq B[N - (\Delta/2)] + \Delta.$$

The boundary condition is that $B(N) = N$, if $N < \Delta$. We get the following proposition by solving the recurrence relation.

Proposition D.3 $B(N) \leq 2N$.

D.3 Bounding the Number of Main Messages

Throughout this section, let us focus only on the main category messages. Of the three messages in this category, let us isolate the *Coin* message. Let M^C be the random variable denoting the number of *Coin* messages exchanged. Corollary 3.2 shows that the expectation of M^C is $O(n \log w)$.

Whenever a node u receives Δ (or $\Delta + 1$ – see below) of the main category messages, it initiates a swap. As a consequence, u sends and receives a few main category messages. Let us count these: (i) u sends 3 *SwapInfoS* messages to its 3 neighbors; (ii) u sends 3 *Baton* messages to its 3 neighbors; (iii) u receives 3 *Baton* messages from its 3 neighbors (when the neighbors return the baton to u). Thus, 9 main category messages are exchanged whenever u receives Δ main category messages. Let $\lambda = 9$.

The above process can be summarized as follows. The process starts with M^C main category messages (*Coin* messages) and for each Δ of these messages, λ messages are generated. Now, let us relate the above process to the container game. We start with M^C messages (=balls). For each Δ messages, λ new messages get generated. (The old messages cannot lead to the generation of any new messages; so, this is similar to dropping balls outside the container). Recall that we set $\Delta = 20$ and so, $\lambda \leq \Delta/2$. By appealing to Proposition D.3, we see that the total number of messages generated can be at most $2M^C$. Since, $M^C = O(n \log w)$ we get the following lemma.

Lemma D.4 $\mathbf{E}[M^m] = O(n \log w)$.

The case of $\Delta + 1$: A careful pass over the pseudocode reveals that under a particular peculiar scenario, a node v may initiate a swap when its counter δ_v equals $\Delta + 1$, (instead initiating the swap when δ_v equals Δ; this is the common scenario). To see this, suppose v has $\delta_v = \Delta - 1$. Now, suppose a neighbor u initiates a swap. Then, u will send a *SwapInfoS* message to v. Upon receiving the message v will increment δ_v; so δ_v now equals Δ. But, v cannot initiate a swap, because it does not possess the baton. It will wait for the baton. Eventually, u will send the baton via a *Baton* message. Upon receiving this message, v will again increment δ_v; so δ_v now equals $\Delta + 1$. Having the baton in hand, v can initiate a swap. Thus, v may initiate a swap when δ_v is $\Delta + 1$.

The above case is not of concern in Lemma D.4, because of the following reason. In the container game, given a choice between Δ and $\Delta + 1$, we will always choose to remove Δ balls (instead of $\Delta + 1$ balls).

D.4 Bounding $M^m_{\max}$

We will use an analysis similar to the one used for the SemiRand algorithm. We will invoke Markov's inequality and appeal to the extended ball game. However, in the current setup, a swap may happen either when the δ counter reaches Δ or $\Delta+1$. In the ball game setting, this corresponds to the scenario where the adversary who designed the ball sequence has the liberty to declare either make the Δth or $(\Delta+1)$th ball as golden. Towards this end, we first slightly generalize the extended ball game.

Generalized Ball Game: This game is similar to the extended ball game and is driven by a parameter Δ. The only difference is as follows. In the new game, the adversary is given the liberty to mark either the Δth or $(\Delta + 1)$th ball as golden. Formally, consider a ball sequence $\tau = b_1, b_2, \ldots, b_K$ fixed by an adversary. Consider any seat $s \in S$. The adversary can mark any appearance of s as golden. However, the difference between the positions of two consecutive golden balls (corresponding to s) must be either Δ or $\Delta + 1$. (In the extended ball game, only the choice of Δ was allowed). The rest of the game is not unchanged. Let $B_{\max}$ denote the maximum number of balls received by any girl. We get the following lemma by extending Theorem 4.3,

Lemma D.5 *Consider the generalized ball game. Fix any ball sequence $\tau = b_1, b_2, \ldots, b_K$, any $\Delta \geq 1$ and any $t \geq 6$ Then,*

$$\Pr[B_{\max} \geq 2t(K/n)] \leq 2n2^{-\frac{tK}{(\Delta+1)n}}.$$

Moreover, the above inequality is true, if we replace K on both LHS and RHS by any $K' \geq K$.

The rest of the analysis is very similar to that of the SemiRand algorithm (Section 4.2). The only difference is that the analysis of the SemiRand algorithm appeals to the extended ball game and Theorem 4.3. Instead, here we make use of the generalized ball game and invoke Lemma D.5. Using this approach, we can establish the following lemma.

Lemma D.6 *There exists a constant c such that for any $\epsilon > 0$,*

$$\Pr[M^m_{\max} \geq (1/\epsilon)4c(1+\Delta)\log w] \leq 2\epsilon.$$

D.5 Destination-end Swap Messages

By Lemma D.4, expectation of M^m is $O(n \log w)$. Whenever a node u receives Δ main category messages, it initiates a swap process. Thus, the number of swaps S is at most M^m/Δ. Thus, expectation of S is $\mathbf{E}[S] = O(n \log w)$. For each such swap, u sends a *SwapRequest* to a random node v. In the swap process, each of the three neighbors of v receive a *SwapInfoD* message. Thus, a total of four destination-end swap messages are generated, per swap. We see that the total number of destination-end swap messages is at most $4S$. The below follows immediately.

Lemma D.7 $\mathbf{E}[M^D] = O(n \log w)$.

Fix an initiation configuration α. Observe that number of swaps S is purely determined by the initiation configuration. Let $S(\alpha)$ denote the number of swaps performed, if α is used as the initiation configuration. Recall that $\mathbf{E}[S] = O(n \log w)$; namely, $\mathbf{E}[S] \leq cn \log w$, for some constant c. Applying the Markov's inequality, we get that

$$\Pr_{\alpha}[S(\alpha) \geq (1/\epsilon)cn\log w] \leq \epsilon. \tag{10}$$

Denote $\lambda = (1/\epsilon)cn \log w$. Let us call an initiation configuration α to be *good*, if $S(\alpha) \leq \lambda$. Fix a good configuration α. Then, at most λ swaps take place. Consider any swap. It is initiated by some node u and received by some random node v. This process generates four destination-end swap messages. A processor x receives one of these messages, if the node occupied by x is v or one of the three neighbors of v. So, the probability that the processor x will receive a message is $4/n$. It follows that M^D_x follows the binomial distribution with expectation $\mu = (4/n)\lambda \leq (1/\epsilon)4c \log w$. We can now appeal to the Chernoff bound(Theorem C.7) and derive the following claim (setting $r = 6$). For any processor x,

$$\Pr[M^D_x \geq 24c \log w] \leq 2^{-24c \log w}$$

Recall our assumption that $w \geq n$. We get the following claim by applying the union bound over all the n processors.

$$\Pr[M^D_{\max} \geq 24(1/\epsilon)c \log w] \leq 1/n^{24c-1} \leq 1/n,$$

provided α is a good configuration. Combining the above claim with Equation 10 gives us the following result.

Lemma D.8 *There exists a constant c such that for any $\epsilon > 0$,*

$$\Pr[M^D_{\max} \geq 24(1/\epsilon)c \log w] \leq 1/n + \epsilon \leq 2\epsilon.$$

D.6 Miscellaneous Messages

First consider, *CoinAccept* and *CoinReject* messages. For each *Coin* message exchanged, two of these messages are generated. We know that $\mathbf{E}[M^m] = O(n \log w)$ and number of *Coin* messages is included in M^m. It follows that expected number of *CoinAccept* and *CoinReject* messages is $O(n \log w)$. Now, consider the *SwapRequest* message. One such message is exchanged for each swap. In the previous section, we saw that the number of expected swaps is $O(n \log w)$. Put together the expected number of miscellaneous messages is $O(n \log w)$.

Lemma D.9 $\mathbf{E}[M^O] = O(n \log w)$.

Now, let us derive a bound on $M^O_{\max}$. Consider a processor x. For each *CoinAccept* and *CoinReject* message received by x, it must also have received a *Coin* message. The *Coin* messages received by x are accounted for in M^m_x. Similarly, for each *SwapAccept* message received by x, x must have initiated a swap. The number of swaps initiated by x is at most M^m/Δ. It follows that $M^O_{\max} \leq 3M^m_x$. Therefore, $M^O_{\max} \leq 3M^m_{\max}$. From Lemma D.6, we get the claim below.

Lemma D.10 *There exists a constant c such that for any $\epsilon > 0$,*

$$\Pr[M^O_{\max} \geq (1/\epsilon)12c(1 + \Delta) \log w] \leq 2\epsilon.$$

```
ProcessLeafPoll(Msg,Sender)
    if (I am not a leaf node)
        Pick a processor $y$ uniformly at random;
        Send LeafPoll to $y$;
    else
        $\delta++$;
        Associate signature with coin;
        if ($\delta \geq \Delta$) InititateSwap(true);
        if (haveCoin==true)
            send Coin to parent;
            waitFor(CoinAccept);
        else set haveCoin=true;
        endif
    endif
```

```
ProcessCoin(Msg,Sender)
    $\delta++$;
    if ($\delta \geq \Delta$) InititateSwap(true);
    if (haveCoin==true)
        if (Sender==parent)
            return CoinReject to parent;
        else /* sender is child */
            send CoinAccept to Sender;
            send Coin to parent;
            waitFor(CoinAccept);
        endif
    else /* Don't have a coin */
        send CoinAccept to Sender;
        if (Sender==child)
            send Coin to otherChild;
            waitFor(CoinAccept or CoinReject);
            if (CoinReject) set haveCoin=true;
        else /* Sender is parent */
            Determine child from signature;
            send Coin to child;
            waitFor(CoinAccept or CoinReject);
            if (CoinReject)
                send Coin to otherChild;
                waitFor(CoinAccept or CoinReject);
                if (CoinReject) set haveCoin=true;
            endif
        endif
    endif
```

Figure 2: Coin Processing Procedures

```
InitiateSwap(flag)
    set GenerateBaton=flag;
    set $\delta = 0$;
    Pick a processor $y$ uniformly at random;
    Send *SwapRequest* to $y$ with state;
    • WaitFor(*SwapAccept*) & copy state;

    Send *SwapInfoS* to neighbors;
    if (GenerateBaton==true)
        Generate *Baton*;
        Send *Baton* to left child;
        WaitFor(*Baton*); $\delta++$;
        Send *Baton* to right child;
        WaitFor(*Baton*); $\delta++$;
        Send *Baton* to Parent;
        WaitFor(*Baton*); $\delta++$;
        Destory *Baton*;
    else /* I was passed Baton */
        if (Sender $\neq$ left child)
            Send *Baton* to left child;
            WaitFor(*Baton*); $\delta++$;
        endif
        if (Sender $\neq$ right child)
            Send *Baton* to right child;
            WaitFor(*Baton*); $\delta++$;
        endif
        if (Sender $\neq$ parent)
            Send *Baton* to parent;
            WaitFor(*Baton*); $\delta++$;
        endif
    endif
```

```
ProcessSwapReq(Msg,Sender)
    Send *SwapInfoD* to neighbors;
    • Send *SwapAccept* to Sender with
            state and copy state from Msg;
```

```
ProcessSwapInfoS(Msg,Sender)
    $\delta++$;
    Update neighbor information from Msg;
    waitFor(Baton); $\delta++$;
    if ( $\delta \geq \Delta$ ) InitiateSwap(false);
    return *Baton* to Sender;
```

Figure 3: Swap Handling Procedures

$$\begin{array}{r|c|c|c|c|c|c|c|c|c|c|c|}
\cline{2-12}
\sigma : & - & - & - & - & g_i & - & - & g' & - & - & - \\
\cline{2-12}
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\bar{s}} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{s'} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\[1em]
\cline{2-12}
\bar{\sigma} : & - & - & - & - & g' & - & - & g_i & - & - & - \\
\cline{2-12}
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\bar{s}} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{s'} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{}
\end{array}$$

Figure 4: Illustration for Claim C.1

$$\begin{array}{r|c|c|c|c|c|c|c|c|c|c|c|}
\cline{2-12}
\sigma : & - & - & - & - & \bar{g} & - & - & g' & - & - & - \\
\cline{2-12}
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\bar{s}} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{s'} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\[1em]
\cline{2-12}
\bar{\sigma} : & - & - & - & - & g' & - & - & \bar{g} & - & - & - \\
\cline{2-12}
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\bar{s}} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{s'} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{}
\end{array}$$

Figure 5: Illustration for Claim C.2