Efficient Decentralized Algorithms for the Distributed Trigger Counting Problem *

Venkatesan T. Chakaravarthy[1], Anamitra R. Choudhury[1], Vijay K. Garg[2], and Yogish Sabharwal[1]

[1] IBM Research - India, New Delhi.
{vechakra, anamchou, ysabharwal}@in.ibm.com
[2] University of Texas, Austin.
garg@ece.utexas.edu

November 15, 2011

Abstract

Consider a distributed system with n processors, in which each processor receives some triggers from an external source. The distributed trigger counting (DTC) 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, synchronizers and other distributed settings.

In this paper, we first present a randomized and decentralized algorithm for the DTC problem with message complexity $O(n \log n \log w)$; furthermore, with high probability, no processor receives more than $O(\log n \log w)$ messages. Building on the ideas of this algorithm, we develop an improved algorithm having message complexity $O(n \log w)$; furthermore, with high probability, no processor receives more than O(log w) messages. However, neither of the two algorithms provide any bound on the messages sent per processors. These algorithm assume complete connectivity between the processors. Next, we present a third algorithm having message complexity $O(n \log n \log w)$, wherein no processor exchanges more than $O(\log n \log w)$ messages with high probability; however, there is a negligible failure probability in raising the alert on receiving w triggers. This algorithm only requires that a constant degree tree be embeddable in the underlying communication graph.

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. We will assume that $w \gg n$; for concreteness $w \geq n^2$.

*Preliminary version of the paper appear in the proceedings of ICDCN'11[3] and IPDPS'11[4]

Algorithm	Message Complexity	MaxRcvLoad	MaxMsgLoad	Exact or Approximate
Centralized[8]	$O(n \cdot (\log n + \log w))$	-	-	Exact
LayeredRand	$O(n \log n \log w)$	$O(\log n \log w)$	-	Exact
CoinRand	$O(n(\log w + \log n))$	$O(\log w + \log n)$	-	Exact
ST-Rand	$O(n \log n \log w)$	$O(\log n \log w)$	$O(\log n \log w)$	Approximate

Figure 1: Summary of DTC Algorithms for the complete graph model

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, [10] 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. [8] 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). In the context of synchronizers [2], a distributed system is required to generate the next pulse when all the messages generated in the current pulse have been delivered. Any message in the current pulse can be viewed as a trigger of the DTC problem.

The algorithms for DTC problem have been studied in prior work, where the following natural parameters have been used to measure the performance of an algorithm:

- Message complexity: the total number of messages exchanged amongst all the processors.
- MaxMsgLoad: the maximum number of messages exchanged by any processor in the system. This includes both the messages sent and received by a processor.
- MaxRcvLoad: the maximum number of messages received by any processor in the system.

While message complexity is a natural measure of the performance of an algorithm, the other two measures are also important for many applications. For example, in sensor networks message processing consumes power, which is generally limited. Thus, a high MaxRcvLoad or MaxMsgLoad may reduce the lifetime of a node. Moreover, these two parameters are indicative of the congestion due to the algorithm. Although MaxMsgLoad is better of the two measures, the technique used to provide good bounds for MaxRcvLoad may lead to good algorithms for MaxMsgLoad. Therefore, we consider both these measures.

We will consider two types of algorithms: deterministic and randomized. Furthermore, we will study two kinds of randomized algorithms, exact and *Las Vegas*. An exact algorithm always raises an alert on receiving w triggers, whereas, a Las Vegas algorithm may fail with negligible error probability.

1.1 Related Work

Garg et al. [8] studied the DTC problem under the *complete graph* model, where in it is assumed that any node can communicate directly with any other node. In other words, the interconnection topology is assumed to be a complete graph. They presented an algorithm for the DTC problem with message complexity $O(n \cdot (\log n + \log w))$. In this algorithm, a particular processor acts as a central node and all the other processors communicate with it. We refer to this algorithm as the *centralized algorithm*. They also showed that any deterministic algorithm for the DTC problem must have message complexity $\Omega(n \log(w/n))$. So, the message complexity of the centralized algorithm is near-optimal. However, both the MaxMsgLoad and the MaxRcvLoad of this algorithm are as high as its message complexity.

Prior work [7, 11, 6, 1] has also addressed the DTC problem under a more stringent setup in which the processors are interconnected via a given tree network and a processor can directly communicate only with its neighbors on the tree. Working under this framework, Emek and Korman [7] present two algorithms. The first algorithm is an Las Vegas algorithm with message complexity $O(n(\log\log n)^2 \log w)$. The other algorithm is a deterministic algorithm that has MaxMsgLoad $O((\log n \log w)^2)$. We stress that these bounds apply to any given tree network.

The DTC problem is also related to the distributed controller problem studied in the literature (eg. [1, 11, 6]). We refer to the work of Emek and Korman [7] for a discussion on this problem. Prior work [5, 10, 9] 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`). Mans et al. [12] study the distributed disaster disclosure problem where the goal is to raise an alarm when the size of the connected component of nodes which have sensed an event exceeds some threshold.

1.2 Our Results and Discussion

As mentioned in the related work, two interconnection models have been considered: complete graph model [8] and tree model [7, 11, 6, 1]. The goal of this paper is to derive a comprehensive set of results under the complete graph model. We present four results:

- Our first result is a centralized algorithm with message complexity $O(n \cdot (\log n + \log w))$; this result provides a simpler alternative to the algorithm of Garg et al.[8] with the same message complexity.

- We present a randomized algorithm called LayeredRand, and prove that the message complexity of this algorithm is $O(n \log n \log w)$; moreover, with high probability, the MaxRcvLoad of this algorithm is bounded by $O(\log n \log w)$. (High probability means a probability of success of at least $1 - 1/n^c$, for any fixed $c \geq 1$). However, the MaxMsgLoad of this algorithm may be as high as its message complexity.

- We build on the ideas of the LayeredRand algorithm and develop a randomized algorithm called CoinRand with improved MaxRcvLoad. We show that with high probability, its message complexity is bounded by $O(n \cdot (\log n + \log w))$ and its MaxRcvLoad is $O(\log n + \log w)$. As mentioned earlier, any deterministic algorithm for the DTC problem must have a message complexity of $\Omega(n \log(w/n))$ [8]. This implies MaxRcvLoad must be at least $\Omega(\log(w/n))$. Thus, the MaxRcvLoad of our randomized algorithm is close to the above deterministic lower bound. In particular, for any constant $\epsilon > 0$, in the case where $w \geq n^{1+\epsilon}$, the two bounds differ only by a constant factor.

- Though CoinRand is good in terms of MaxRcvLoad, its MaxMsgLoad can be as large as the message complexity. We present a second randomized algorithm called spanning tree algorithm (denoted ST-Rand) which has a better bound on the MaxMsgLoad. With high probability, its MaxMsgLoad (and hence, also the MaxRcvLoad) is bounded by $O(\log n \log w)$. This algorithm has the additional advantage that it only requires that a constant degree tree be embeddable in the underlying communication graph. It is an approximate algorithm with negligible failure probability. The algorithm assumes that $w \leq 2^{\text{poly}(n)}$, which is a reasonable assumption for all practical scenarios.

We note that the core ideas behind the centralized algorithm are known from prior work [1, 7, 8, 6]. The main purpose of presenting this algorithm is to make the paper self-contained; moreover, the discussion of this algorithm brings out the issues pertaining to the problem and improves the readability of the paper. The ST-Rand algorithm is implicit in the work of Emek and Korman [7]; we present an explicit description for the sake of completeness.

The main contributions of this paper are the two algorithms LayeredRand and CoinRand. While CoinRand algorithm has better MaxRcvLoad, LayeredRand is simpler and has the advantage that the message complexity is always bounded by $O(n \cdot \log n \cdot \log w)$, whereas the CoinRand algorithm satisfies the bound of $O(n \cdot (\log n + \log w))$with only high probability.

To summarize, the CoinRand algorithm provides near-optimal bound for the MaxRcvLoad when the underlying interconnection network is a complete graph and the ST-Rand provides the best known bound for MaxMsgLoad when the underlying interconnection network admits a constant degree tree embedding.

A comparison with the work of Emek and Korman [7] is in order. Even though their algorithms provide weaker message bounds, their algorithms can work on any given (connected) inter-connection network. The problem of deriving message bounds similar to CoinRand and ST-Rand for arbitrary interconnection networks is a challenging open problem.

The prior results and our results under the complete graph model are summarized in Figure 1. An unspecified entry in the table indicates that the corresponding bound was not considered and hence, it could be as high as the message complexity of the algorithm. We note that all the bounds in the table refer to the number of messages. Each of these messages can be of size at most $O(\log w)$ bits.

2 A Deterministic Algorithm

For the DTC problem, Garg et al. [8] presented an algorithm with the message complexity of $O(n \log w)$. In this section, we describe a simple alternative deterministic algorithm with the same message complexity. We note that the core ideas behind this algorithm are known from prior work [1, 7, 8, 6]. The main purpose of presenting this algorithm is to make the paper self-contained; moreover, the discussion of this algorithm brings out the issues pertaining to the problem and improves the readability of the paper.

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 to 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*, meaning even though all the w triggers have been received, the system does not detect termination. 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, the system would enter 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 for the next round. 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 strictly less than $\hat{w}$; therefore, 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 efficient in terms of message complexity. However, the master may receive upto $O(n \log w)$ messages and so, the MaxRcvLoad of the algorithm is $O(n \log w)$. In the next section, we present an efficient *randomized* algorithm which simultaneously achieves provably good message complexity and MaxRcvLoad bounds.

3 LayeredRand Algorithm

In this section, we present a randomized algorithm called LayeredRand. Its message complexity is $O(n \log n \log w)$ and with high probability, its MaxRcvLoad is $O(\log n \log w)$.

For the ease of exposition, we first describe our algorithm under the assumption that the triggers are delivered one at a time; meaning, all the processing required for handling a trigger is completed before the next trigger arrives. This assumption allows us to better explain the core ideas of the algorithm. We will relax this assumption and discuss how to handle the concurrency issues in Section 6.

3.1 Algorithm

For the sake of simplicity, we assume that $n = 2^L - 1$, for some integer L. The n processors are arranged in L layers numbered 0 through $L - 1$. For $0 \leq \ell < L$, layer ℓ consists of 2^ℓ processors. Layer 0 consists of a single processor, which we refer to as the *root*. Layer $L - 1$ is called the *leaf*

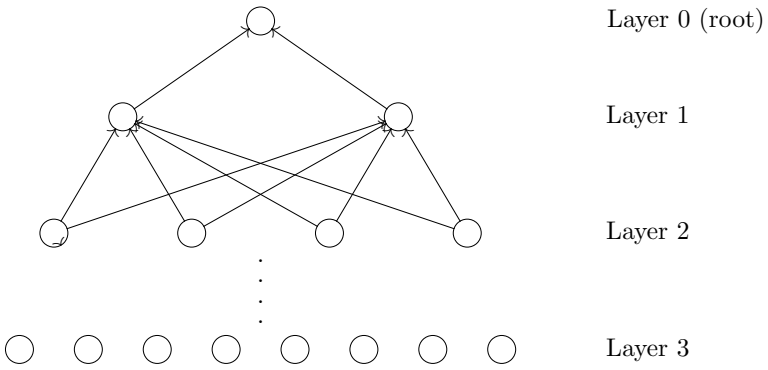

Figure 2: Illustration for LayeredRand

layer. The layering is illustrated in Figure 3, for $n = 15$. Only processors occupying adjacent layers will communicate with each other. We assume that the multi-layered partitioning is fixed apriori and known to all the processors.

The algorithm proceeds in multiple rounds. In the beginning of each round, the system needs to know how many triggers are yet to be received. This can be computed by keeping track of the total number of triggers received in all the previous rounds and subtracting this quantity from w. Let the term *initial value of a round* mean the number of triggers yet to be received at the beginning of the round. We use a variable $\hat{w}$ to store the initial value of any round. In the first round, we set $\hat{w} = w$, since all the w triggers are yet to be received.

We next describe the procedure followed in a single round. Let $\hat{w}$ denote the initial value of this round. For each $1 \leq \ell < L$, we compute a threshold $\tau(\ell)$ for the layer ℓ:

$$\tau(\ell) = \left\lceil \frac{\hat{w}}{4 \cdot 2^{\ell} \cdot \log(n+1)} \right\rceil .$$

Each processor x maintains a counter $C(x)$, which is used to keep track of some of the triggers received by x and other processors occupying the layers below of that of x. The exact semantics $C(x)$ will become clear shortly. The counter is reset to zero in the beginning of the round.

Consider any non-root processor x occupying a level ℓ. Whenever x receives a trigger, it will increment $C(x)$ by one. If $C(x)$ reaches the threshold $\tau(\ell)$, x chooses a processor y occupying level $\ell - 1$ *uniformly at random* and sends a message to y. We refer to such a message as a *coin*. Upon receiving the coin, the processor y updates $C(y)$ by adding $\tau(\ell)$ to $C(y)$. Intuitively, receipt of a coin by y means that y has evidence that some processors below the layer $\ell - 1$ have received $\tau(\ell)$ triggers. After the update, if $C(y) \geq \tau(\ell - 1)$, y will pick a processor z occupying level $\ell - 2$ uniformly at random and send a coin to z. Then, processor y updates $C(y) = C(y) - \tau(\ell - 1)$. Processor z handles the coin similarly. See Figure 2. A directed edge from a processor u to a processor v means that u may send a coin to v. Thus, a processor may send a coin to any processor in the layer above. This is illustrated for the top three layers in Figure 2.

We now formally describe the behavior of a non-root processor x occupying a level ℓ. Whenever x receives a trigger from the external source or a coin from level $\ell + 1$, it behaves as follows:

- If a trigger is received, increment $C(x)$ by one.
- If a coin is received from level $\ell + 1$, update $C(x) = C(x) + \tau(\ell + 1)$.
- If $C(x) \geq \tau(\ell)$,
  - Among the $2^{\ell-1}$ processors occupying level $\ell - 1$, pick a processor y uniformly at random and send a coin to y.

– Update $C(x) = C(x) - \tau(\ell)$.

The behavior of the root is similar to that of the other processors, except that it does not send coins. The root processor r also maintains a counter $C(r)$. Whenever it receives a trigger from the external source, it increments $C(r)$ by one. If it receives a coin from level 1, it updates $C(r) = C(r) + \tau(1)$.

An important observation is that at any point of time, any trigger received by the system in the current round is accounted in the counter $C(x)$ of exactly one processor x. This means that the sum of $C(x)$ over all the processors gives us the exact count of the triggers received in the system so far in this round. This observation will be useful in proving the correctness of the algorithm.

The crucial activity of the root is to initiate an *end-of-round* procedure. When $C(r)$ reaches $\lceil \hat{w}/2 \rceil$ (i.e., when $C(r) \geq \lceil \hat{w}/2 \rceil$), the root declares *end-of-round*. Now, the root needs to get a count of the total number of triggers received by all the processors in this round. Let this count be w'. The processors are arranged in a pre-determined binary tree formation such that each processor x has exactly one parent from the layer above and exactly two children from the layer below. The end-of-round notification can be broadcast to all the processors in a recursive top-down manner. Similarly, the sum of $C(x)$ over all the processors can be reduced at the root in a recursive bottom-up manner. Thus, the root obtains the value w', i.e., the total number of triggers received in the system in this round. The root then updates the initial value for the next round by computing $\hat{w} = \hat{w} - w'$, and broadcasts this to all the processors, again in a recursive fashion. All the processors then update their $\tau(\ell)$ values for the new round. This marks the start of the next round. Notice that in the end-of-round process, each processor receives at most a constant number of messages.

At the end of any round, if the newly computed $\hat{w}$ is zero, we know that all the w triggers have been received. So, the root can raise an alert to the user and the algorithm is terminated.

It is easy to derive a bound on the number of rounds taken by the algorithm. Observe that in successive rounds the initial value drops by a factor of two (meaning, $\hat{w}$ of round $i+1$ is at most half the $\hat{w}$ of round i). Thus, the algorithm takes at most $\log w$ rounds.

3.2 Correctness of the Algorithm

We now show that the system will correctly raise an alert to the user when all the w triggers are received. The main part of the proof involves showing that after starting a new round, the root always enters the end-of-round procedure, i.e., the system does not get stalled in the middle of the round, when all the triggers have been delivered.

We denote the set of all processors by $\mathcal{P}$. Consider any round and let $\hat{w}$ be the initial value of the round. Let x be any non-root processor and let ℓ be the layer in which x is found. Notice that at any point of time, we have $C(x) \leq \tau(\ell) - 1$. Thus, we can derive a bound on the sum of $C(x)$:

$$\sum_{x \in \mathcal{P} - \{r\}} C(x) \;\leq\; \sum_{\ell=1}^{L-1} 2^{\ell}(\tau(\ell) - 1) \;\leq\; \sum_{\ell=1}^{L-1} 2^{\ell} \cdot \frac{\hat{w}}{4 \cdot 2^{\ell} \cdot \log(n+1)} \;\leq\; \frac{(L-1)\hat{w}}{4 \cdot \log(n+1)} \;\leq\; \frac{\hat{w}}{4}$$

Now suppose that all the outstanding $\hat{w}$ triggers have been delivered to the system in this round. We already saw that at any point of time, $\sum_{x \in \mathcal{P}} C(x)$ gives the number of triggers received by the system so far in the current round. Thus, $\sum_{x \in \mathcal{P}} C(x) = \hat{w}$. It follows that the counter at the root $C(r)$ satisfies[1] $C(r) \geq 3\hat{w}/4 \geq \lceil \hat{w}/2 \rceil$. But, this means that the root would initiate the end-of-round procedure. We conclude that the system will not get stalled in a dead state.

[1] We note that $C(r)$ is an integer, and hence this holds even when $\hat{w} \geq 2$

The above argument shows that the system always makes progress by moving into the next round. As we observed earlier, the initial value $\hat{w}$ drops by a factor of at least two in each round. So, eventually, $\hat{w}$ must become zero and the system will raise an alert to the user.

3.3 Bound on the Message Complexity

We now prove message bounds.

Lemma 3.1 *The message complexity of the algorithm is $O(n \log n \log w)$.*

Proof: As argued before, the algorithm takes only $O(\log w)$ rounds to terminate.

Consider any round and let $\hat{w}$ be the initial value of the round. Consider any layer $1 \leq \ell < L$. Every coin sent from layer ℓ to $\ell - 1$ means that at least $\tau(\ell)$ triggers have been received by the system in this round. Thus, the number of coins sent from layer ℓ to the layer $\ell - 1$ can be at most $\hat{w}/\tau(\ell)$. Summing up over all the layers, we can get a bound on the total number of coins (messages) sent in this round:

$$\text{Number of coins sent} \quad \leq \quad \sum_{\ell=1}^{L-1} \frac{\hat{w}}{\tau(\ell)} \quad \leq \quad \sum_{\ell=1}^{L-1} 4 \cdot 2^{\ell} \log(n+1) \quad \leq \quad 4 \cdot (n-1) \log(n+1)$$

The end-of-round procedure involves only $O(n)$ messages, in any particular round. Summing up over all $\log w$ rounds, we see that the message complexity of the algorithm is $O(n \log n \log w)$. $\square$

3.4 Bound on the MaxRcvLoad

In this section, we show that with high probability, MaxRcvLoad is $O(\log n \log w)$. We use the following Chernoff bound (see [13]) for this purpose.

Theorem 3.2 (see [13], Theorem 4.4) *Let X be the sum of a finite number of independent $0-1$ random variables. Let the expectation of X be $\mu = \mathbf{E}[X]$. 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' on both sides.*

Lemma 3.3 *There exists a constant c such that*

$$\Pr[\text{MaxRcvLoad} \geq c \log n \log w] \quad \leq \quad n^{-47}.$$

Proof: Let us first consider the number coins received by any processor. Processors in the leaf layer do no receive any coins and so, it suffices to consider the processors occupying other layers.

Consider any layer $0 \leq \ell \leq L - 2$ and let x be any processor in layer ℓ. Let M_x be the random variable denoting the number of coins received by x. In any given round, the number of coins received by layer ℓ is at most $\frac{\hat{w}}{\tau(\ell+1)} \leq 4 \cdot 2^{\ell+1} \log(n+1)$. Thus, the total number of coins received by layer ℓ for all rounds is at most $4 \cdot 2^{\ell+1} \log(n+1) \log w$ because the algorithm takes at most $\log w$ rounds. Each of these coins is sent uniformly and independently at random to one of the 2^{ℓ} processors occupying layer ℓ. Thus, expected number of coins received by x is

$$\mathbf{E}[M_x] \quad \leq \quad \frac{4 \cdot 2^{\ell+1} \log(n+1) \log w}{2^{\ell}} \quad = \quad 8 \log(n+1) \log w$$

The random variable M_x is a sum of independent 0-1 random variables. Applying the Chernoff bound given by Theorem 3.2 (taking $r = 6$), we see that

$$\Pr[M_x \geq 48 \log(n+1) \log w] \quad \leq \quad 2^{-48 \log(n+1) \log w} \quad < \quad n^{-48}.$$

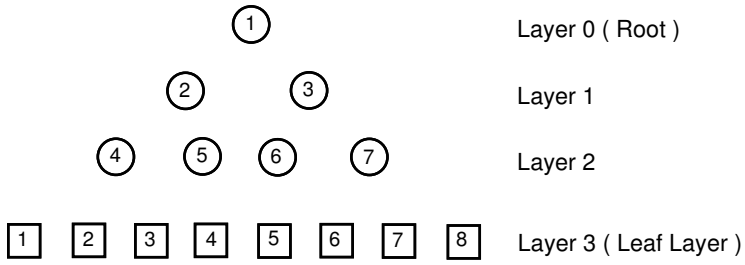

Figure 3: Illustration of the CoinRand Algorithm

Applying the union bound, we see that

$$\Pr[\text{There exists a processor } x \text{ with } M_x \geq 48 \log(n+1) \log w] \quad < \quad n^{-47}.$$

During the end-of-round process, a processor receives at most a constant number of messages in any round. So, the total of these messages received by any processor is $O(\log w)$. □

Remark: Notice that we only provide a bound on MaxRcvLoad and not on MaxMsgLoad. The difficulty in dealing with MaxMsgLoad is that some processor may receive a lot of triggers. Such a processor would send too many messages under the LayeredRand algorithm.

4 CoinRand Algorithm

In this section, we build on the ideas in the LayeredRand algorithm and develop a randomized algorithm called CoinRand, whose message complexity and MaxRcvLoad are, with high probability, $O(n \log w)$ and $O(\log w)$ respectively. This is an exact algorithm and assumes that the interconnection network is a complete graph.

As in the case of LayeredRand, we first describe our algorithm under the assumption that the triggers are delivered one at a time; meaning, all the processing required for handling a trigger is completed before the next trigger arrives. We will discuss how to handle the concurrency issues in Section 6.

4.1 Algorithm

For the ease of exposition, we assume that $n = 2^L$, for some integer L. The algorithm and analysis can easily be extended to the general case. The algorithm works in two phases. In this section, we focus on describing the first phase, which is the main phase of the algorithm. The second phase handles a boundary case and is described in a later section (Section 4.6).

The n processors are arranged in $(L+1)$ layers numbered 0 through L. For $0 \leq \ell \leq L$, layer ℓ consists of 2^ℓ nodes. All the processors appear as a node in the bottom-most layer (layer L), which is called the *leaf layer*. The other $(L-1)$ layers are called the internal layers and they put together contain $(n-1)$ nodes. Of the n processors, an arbitrary processor is omitted and the remaining $(n-1)$ processors are used to fill the internal layers. Thus, all the processors (except one) play dual roles in the algorithm: as a leaf-node and as an internal node. Layer 0 consists of a single node, which we refer to as the *root*. The layering is illustrated in Figure 3, for $n = 8$. All the eight processors appear in the leaf layer and the first seven processors are used to fill the internal nodes. We assume that the layering information is fixed apriori and known to all the processors.

The algorithm proceeds in multiple rounds as in LayeredRand. In the beginning of each round, the system needs to know how many triggers are yet to be received. We shall compute

this by maintaining a counter at each processor that stores the number of triggers received by the processor in the current round. We use a variable $\hat{w}$ to store the initial value of any round. In the first round, we set $\hat{w} = w$, since all the w triggers are yet to be received.

We now describe the working of any particular round. Each processor computes a *threshold value* $\tau = \lceil \hat{w}/4n \rceil$. Note that the threshold value is same for all processors in contrast to LayeredRand algorithm. Every processor x also maintains a counter $C(x)$, which is used to keep track of the number of the triggers received by x in the current round. Whenever x receives a trigger from the external source, it increments $C(x)$ by one. If $C(x)$ reaches the threshold τ, x chooses a processor y occupying the layer $(L-1)$ *uniformly at random* and sends a message to y. We refer to such a message as a *coin*. In addition, the processor x resets its $C(x)$ value to zero. Upon receiving the coin, processor y works as follows. If this is the first coin received by y, it just keeps the coin with itself. On the other hand, if y already possesses a coin, it chooses a processor z occupying the layer $(L-2)$ *uniformly at random* and passes the coin to z. Processor z handles the coin similarly.

The behavior of the root node is different from that of the other internal nodes. Upon receiving a coin, the root node initiates an *end-of-round* procedure. Now, the root needs to get a count of the total number of triggers received by all the processors in this round. Let this count be w'. Notice that $w' = C_{\text{sum}} + \tau \times N_{\text{coins}}$, where C_{sum} is the sum of $C(x)$ over all the processors and N_{coins} is the total number of coins possessed by the internal nodes. The root can compute the count w' via a simple broadcast and upcast procedure in a pre-determined binary tree. The end-of-round notification can be broadcast to all the processors in a recursive top-down manner. Similarly, the values C_{sum} and N_{coins} can be reduced at the root in a recursive bottom-up manner. Thus, the root obtains the value w', i.e., the total number of triggers received in the system in this round. The root then updates the initial value for the next round by computing $\hat{w} = \hat{w} - w'$, and broadcasts this to all the processors, again in a recursive fashion. All the processors then update their τ values for the new round. The processors delete any coin that they possess. This marks the start of the next round. Notice that in the end-of-round process, each processor exchanges at most a constant number of messages.

Now, suppose the newly computed $\hat{w}$ is less than n. This is a boundary case and needs a slight modification to the procedure described so far. In this scenario, the algorithm enters the second phase. Essentially, the modification is that the number of internal layers is reduced. The details of the second phase are described in Section 4.6.

4.2 First Phase of CoinRand : Proof of Correctness

In this section, we prove the *correctness* of the first phase of the algorithm: meaning, we show that either (i) the algorithm raises an alert and terminates; or (ii) enters the second phase. In other words, we argue that the algorithm never stalls in a deadlock in the first phase even after all the w triggers are received. After proving the correctness, we then derive bounds on the number of messages exchanged and the maximum number of messages received by any processor.

The correctness claim is proved by showing that after starting a new round, the root always enters the end-of-round procedure.

Consider any round and let $\hat{w}$ be the initial value of the round. The root invokes the end-of-round procedure upon receiving a coin. We now prove that the maximum number of triggers that can be received by the system without the root receiving a coin is at most $\hat{w} - 1$. The extreme case is as follows: every non-root internal node has a coin and every processor x has $C(x) = \tau - 1$. Under this situation, no matter which processor receives the next trigger, a coin will be generated by that processor and propagated all the way up to the root, leading to the invocation of the end-of-round

procedure.

In this extreme case, the total number of coins in the system is $n-2$. Recall that each coin represents the receipt of τ triggers. The sum of the $C(x)$ values of all processors x in the leaf layer is $n \cdot (\tau - 1)$. Therefore, the total number of triggers received so far in this round is:

$$w' = (n-2)\tau + n \cdot (\tau - 1)$$

We shall argue that $w' < \hat{w}$. To see this, let us consider two cases. The first case is where $n \leq \hat{w} \leq 4n$. In this case $\tau = 1$ and so, $w' = n - 2 < \hat{w}$ (since we are in the first phase). The second case is where $\hat{w} > 4n$. In this case:

$$w' = (n-2)\tau + n \cdot (\tau - 1) \leq 2n\tau \leq 2n(\frac{\hat{w}}{4n} + 1) < \hat{w}.$$

This means that the system can receive at most $\hat{w} - 1$ triggers without the root receiving a coin. It follows that the root will always initiate the end-of-round procedure at least when all the $\hat{w}$ triggers are received. We conclude that the system will never get stalled in a dead state.

The above argument shows that the system always makes progress by moving into the next round. The initial value $\hat{w}$ decreases in each round. If $\hat{w}$ falls below n, we proceed to the second phase of the algorithm as given in Section 4.6.

4.3 Bound on the Number of Rounds

We now derive a bound on the message complexity for the first phase of the CoinRand algorithm. First, we derive a bound on the number of rounds taken by the algorithm.

As shown above the value of $\hat{w}$ decreases in each round and the algorithm enters the second phase when $\hat{w}$ drops below n. The number of rounds taken by the first phase depends on the rate at which $\hat{w}$ value decreases in each round. In the worst case, in a particular round, the root can invoke the end-of-round procedure after only L coins have been generated. In this case, after receiving only τL triggers, the algorithm may enter the next round. Thus, the $\hat{w}$ value may reduce only by a small amount and thereby, the algorithm may take a large number of rounds to enter the second phase. Here we show that such a scenario is highly unlikely. We achieve this by proving that with sufficiently high probability, the $\hat{w}$ value drops by a constant factor in every round.

We say that a round is *successful*, if the root receives a coin only after at least $n/4$ coins are generated and propagated. Notice that in any successful round, the system receives at least $(n/4) \cdot \tau$ triggers. We have

$$w' = (n/4) \cdot \tau \geq \hat{w}/16.$$

Thus the initial value reduces by a (constant) factor of $15/16$ at the end of any successful round.

We proceed to show that any round is successful with probability at least $1/2$. Consider the following probabilistic process similar to the coin propagation mechanism of CoinRand. Fix any $1 \leq \ell \leq L-1$ and let $m = 2^\ell$. We insert and propagate $m/2$ coins as follows. Each coin is thrown at a processor x chosen uniformly at random from layer ℓ. If this is the first coin received by x, it keeps the coin with itself; otherwise, it chooses a processor y uniformly at random from layer $\ell - 1$ and passes the coin to y. The processor y handles the coin similarly. If the coin propagates all the way up to the root, the root keeps the coin to itself. Let $P(\ell)$ denote the probability that the root receives at least one coin in the above random process. Below, we prove that for any $1 \leq \ell \leq L-1$, $P(\ell) \leq 1/2$. In particular, this implies that $P(L-1) \leq 1/2$. This establishes that in the CoinRand algorithm, the probability that the root receives a coin before $n/4$ coins are

generated and propagated is at most 1/2. In other words, the probability that a round is successful is at least 1/2.

We now show that $P(\ell) \leq 1/2$, for all $1 \leq \ell \leq L-1$. First, we establish a recurrence relation on $P(\ell)$, given below.

Lemma 4.1 *For any* $4 \leq \ell \leq L-1$,

$$P(\ell) \leq (3/4)^{m/4} + P(\ell - 1),$$

where $m = 2^\ell$.

Proof:

Let E be the event that the root receives at least one coin; so $P(\ell) = \Pr[E]$. We consider two cases: (i) at the end of the process, layer ℓ has greater than $m/4$ coins; (ii) at the end of the process, layer ℓ has at most $m/4$ coins. Let E_1 represent the former event and let E_2 represent the latter event. Then,

$$\Pr[E] = \Pr[E_1] \cdot \Pr[E \mid E_1] + \Pr[E_2] \cdot \Pr[E \mid E_2].$$

Suppose the event E_1 occurred. Then, at most $m/4$ coins have propagated to layer $\ell - 1$. And hence, $\Pr[E \mid E_1] \leq P(\ell - 1)$. Clearly, $\Pr[E_1] \leq 1$ and $\Pr[E \mid E_2] \leq 1$. Therefore,

$$\Pr[E] \leq P(\ell - 1) + \Pr[E_2].$$

Let us compute the probability $\Pr[E_2]$. Let F denote the set of all nodes in layer ℓ that contain a coin at the end of the process. The event E_2 means that $|F| \leq m/4$. Hence, $\Pr[E_2] = \Pr[|F| \leq m/4]$. Let Z be the set of all 2^ℓ nodes in layer ℓ. Let $\mathcal{S}$ denote the collection of all subsets of Z of cardinality $m/4$ (i.e., $\mathcal{S} = \{S \subseteq Z \ : \ |S| = m/4\}$). The premise $|F| \leq m/4$ means that there exists a subset $S \in \mathcal{S}$ such that $F \subseteq S$. Thus,

$$\Pr[|F| \leq m/4] = \Pr[\exists S \in \mathcal{S} \ : \ F \subseteq S].$$

Consider any set $S \in \mathcal{S}$ and suppose $F \subseteq S$. This means that each one of the $m/2$ coins got inserted into some node in S. For any particular coin, the probability that it got inserted into some node in S is 1/4. Hence, $\Pr[F \subseteq S] = (1/4)^{m/2}$. Applying the union bound, we get that

$$\Pr[\exists S \in \mathcal{S} \ : \ F \subseteq S] \leq |\mathcal{S}| \left(\frac{1}{4}\right)^{m/2}.$$

The collection $\mathcal{S}$ contains $\binom{m}{m/4}$ sets. Recall that for any integers $a, b \geq 1$,

$$\binom{a}{b} \leq \left(\frac{ae}{b}\right)^b.$$

(See for instance [14]). Thus, $\binom{m}{m/4} \leq (4e)^{m/4}$. It follows that

$$\begin{aligned} |\mathcal{S}| \left(\frac{1}{4}\right)^{m/2} &\leq (4e)^{m/4} \left(\frac{1}{4}\right)^{m/2} \\ &\leq \left(\frac{e}{4}\right)^{m/4} \\ &\leq \left(\frac{3}{4}\right)^{m/4}. \end{aligned}$$

We have proved the lemma. □

We now show that $P(\ell) \leq 1/2$, for all $1 \leq \ell \leq L-1$.

Corollary 4.2 *For any $\ell \leq L-1$, $P(\ell) \leq 1/2$. In particular, $P(L-1) \leq 1/2$.*

Proof: We prove the corollary by solving the recurrence relation given by Lemma 4.1. Notice that $P(1) = 0$ (because only one coin will be inserted, which will settle down in layer 1 and will not reach the root). Similarly $P(2) = 0$. Consider the base case of $P(3)$. Here, we have four layers numbered 0 to 3. Four coins get inserted in layer 3. The first three coins will get deposited in some node in the bottom three layers and only the fourth coin has a chance of reaching the root. For the latter event to happen, each one of the three bottom layers should get one of the first three coins. Then, the fourth coin must get propagated through the nodes that contain the first three coins. The probability of the last of event is $(1/8) \cdot (1/4) \cdot (1/2) = 1/64$. Thus, $P(3) \leq 1/64$. Now consider any $\ell \geq 4$. The recurrence relation show that

$$\begin{aligned} P(\ell) &= P(3) + \sum_{j=2}^{\ell-2} (3/4)^{2^j} \\ &\leq (1/64) + \sum_{j \geq 2} (3/4)^{2^j} \\ &= (1/64) + (3/4)^4 + (3/4)^8 + \sum_{j \geq 4} (3/4)^{2^j} \\ &\leq (1/64) + (3/4)^4 + (3/4)^8 + 2(3/4)^{16} \\ &\leq 1/2 \end{aligned}$$

The corollary is proved. □

Recall that a round is said to be successful, if the root receives a coin only after at least $n/4$ coins are generated and that in any successful round, the initial value decreases by a factor of at least $15/16$. The above corollary shows that any round is successful with probability at least $1/2$. We use this observation to show that with high probability, the first phase takes only $O(\log w)$ rounds. We use the following Chernoff bound (see [15], Theorem A.1.5) for this purpose.

Theorem 4.3 ([15]) *Consider tossing an unbiased coin (with probability of head and tail being $1/2$) multiple times. Let X be the random variable[2] denoting the number of tosses before observing r heads. Then, expectation of X is $\mathbf{E}[X] = 2r$. Furthermore, for any $\epsilon \geq 3$,*

$$\Pr[X > (2+\epsilon)r] \leq e^{-\epsilon r/4}.$$

We now prove a bound on the number of rounds executed in the first phase.

Lemma 4.4 *There exists a constant a such that probability that the first phase takes more than $a \log w$ rounds is at most $1/n^4$.*

Proof: In any successful round, $\hat{w}$ decreases by a factor of $15/16$. For $\hat{w}$ to get below n, we need at most $c \log w$ successful rounds, for some constant $c \geq 1$. We have already shown that any round is successful with probability at least $1/2$. Let X be the random variable denoting the number of rounds till we get $c \log w$ successful rounds. Expectation of X is $\mathbf{E}[X] \leq 2r$, where $r = c \log w$. Note that $w \geq n$. Setting $\epsilon = 16$ and invoking Theorem 4.3, we get that $\Pr[X \geq 18c \log w] \leq 1/n^4$. □

[2]The random variable X follows the negative binomial distribution with parameter $1/2$

4.4 Bound on the Message Complexity

In this section, we show that with high probability, at most $O(n \log w)$ messages are exchanged in the first phase. Let us first derive an upper bound on the maximum number of messages that can be exchanged in any round. Clearly, at most $n-1$ coins can be generated before the end-of-round procedure is invoked. A coin that gets deposited at a layer ℓ must travel from the leaf-layer to the layer ℓ. This involves $L-\ell$ message exchanges. As there are 2^ℓ nodes in layer ℓ, summing up over all the layers, we get a bound on the total number of messages exchanged in a particular round:

Number of messages exchanged in a particular round

$$\begin{aligned} &\leq \sum_{\ell=0}^{L-1}(L-\ell)\cdot 2^\ell = \sum_{\ell=1}^{L}\sum_{j=0}^{L-\ell} 2^j \\ &\leq \sum_{\ell=1}^{L} 2^{L+1-\ell} \leq 2^{L+1} = 2n \end{aligned}$$

The end-of-round procedure involves $2n$ messages. So the total number of messages exchanged in any particular round is at most $4n$.

Lemma 4.4 shows that with high probability, the number of rounds is at most $a \log w$, for some constant a. Combining the lemma with the above derivation, we get that

$$\Pr[N_{\text{msgs}} \geq 4an \log w] \leq 1/n^4,$$

where N_{msgs} is the total number of messages exchanged in the first phase of the CoinRand algorithm.

4.5 Bound on the MaxRcvLoad

In this section, we show that with high probability, no processor receives more than $O(\log w)$ messages in the first phase. We shall use the Chernoff bound stated in Theorem 3.2 for this purpose.

Lemma 4.5 *There exists a constant a such that*

$$\Pr[\textit{Failure}] \leq 2/n^4,$$

where Failure is the event that some processor receives more than $24a \log w$ messages in the first phase.

Proof: From Lemma 4.4, we know that with probability at least $(1-1/n^4)$, the number of rounds is at most $a \log w$, for some constant a. Let us first consider the scenario where the number of rounds is at most $a \log w$. Consider any processor x present in some internal layer ℓ. Let X be the random variable denoting the number of messages received by x in the first phase. The number of nodes in layers ℓ and above is given by $\sum_{j=0}^{\ell} 2^j \leq 2^{\ell+1}$. Thus at most $2^{\ell+1}$ coins pass through the layer ℓ in any particular round. The propagation of each of these coins leads to some processor y in layer ℓ receiving a message, where y is chosen uniformly at random. Since there are 2^ℓ processors in layer ℓ, the processor x receives each one of these messages with probability $1/2^\ell$. Hence, the expected number of messages received by x in any round is 2. The total number of messages expected to be received by x in the first phase is $\mathbf{E}[X] = 4a \log w$. The variable X is binomially distributed and so, we can apply the Chernoff bound (taking $r = 6$ in Theorem 3.2)

$$\Pr[X \geq 24a \log w] \leq 2^{-24a \log w} \leq n^{-24}.$$

Applying the union bound, we get that

$$\Pr[\text{MaxRcvLoad} \geq 24a \log w] \leq n^{-23}.$$

The above bounds hold under the scenario where the number of rounds is at most $a \log w$. The probability that the above scenario fails to happen is at most $1/n^4$. Summing up the two error probabilities, we get the claim. □

4.6 Second Phase of the CoinRand Algorithm

The first phase terminates when the initial value $\hat{w}$ drops below n and the algorithm enters the second phase, described in this section. The procedure followed in the second phase is the same as that of the first phase, except that the number of internal layers changes over rounds.

Let w_0 be the initial value when the algorithm entered the second phase (where $w_0 < n$). The second phase also works in multiple rounds. Let $\hat{w}$ denote the initial value of a round. Let k be the unique integer such that $n/2^k \leq \hat{w} < n/2^{k-1}$.

As before, the n processors are arranged in layers, the leaf layer containing all the n processors. However, instead of L internal layers, we now maintain only $(L-k)$ internal layers; for $0 \leq \ell \leq (L-k-1)$, layer ℓ consists of 2^ℓ processors. Thus each processor appears at most once as an internal node. The processor at layer 0 is the root processor. As before, communication happens only between processors occupying adjacent layers.

The coins are communicated in a similar way as in the first phase. The only difference is that the threshold value is always set as 1. Meaning, a processor will generate and propagate a new coin for every trigger that it receives. When the root receives a coin, it calls the end-of-round procedure and computes the $\hat{w}$ for the next round. Depending on the $\hat{w}$ value range, the number of layers is adjusted for the next round.

At the end of any round in the second phase, if the newly computed $\hat{w}$ is zero, we know that all the w triggers have been received. So, the root can raise an alert to the user and the algorithm is terminated. This completes the description of the algorithm.

Proof of Correctness: We now argue that the algorithm correctly raises an alert when all the w_0 triggers are received by the system. The correctness proof is similar to that of the first phase. Consider any round and let $\hat{w}$ be the initial value of the round. Let k be the unique integer such that $n/2^k \leq \hat{w} < n/2^{k-1}$; so there are $(L-k)$ internal layers, with the bottom-most internal layer containing $n/2^{k+1}$ processors. In the extreme case, each processor in the internal layers has exactly one coin before the round terminates. This implies the maximum number of triggers received by the system before the end-of-round is

$$\sum_{\ell=0}^{L-k} 2^\ell = n/2^k - 1 < \hat{w}.$$

Thus the system will never get stalled in a dead state.

Bound on the Message Complexity and MaxRcvLoad**:** Consider any round and let $\hat{w}$ be the initial value of the round. Let k be the integer such that $n/2^k \leq \hat{w} < n/2^{k-1}$. Thus the bottom-most internal layer has $n/2^{k+1}$ processors. By arguments similar to that of Corollary 4.2, the probability that the root invokes the end-of-round procedure before receiving $n/2^{k+2}$ triggers is at most $1/2$. Since $n/2^k \leq \hat{w} < n/2^{k-1}$,

$$\Pr[\text{round ends before receiving } \hat{w}/8 \text{ triggers}] \leq 1/2.$$

We call a round to be *good* if it terminates only after $\hat{w}/8$ or more triggers have been received by the system. Thus in any good round, the initial value decreases by a factor at least 7/8. Hence, for completion of CoinRand algorithm, $c \log w_0$ good rounds are sufficient, for some constant $c \geq 1$.

Following arguments similar to Section 4.4, we can show that there exists a constant a such that with probability at least $(1 - 1/n^4)$, the second phase takes at most $a \log n$ rounds. Similar to Lemma 4.5, we can show that no processor receives more than $O(\log n)$ messages in the second phase with probability at least $(1 - 2/n^4)$.

Section 4.4 shows that the message complexity of the first phase is $O(n \log w)$ with probability at least $(1 - 1/n^4)$; the argument above shows that the message complexity of the second phase is $O(n \log n)$ with probability at least $(1 - 1/n^4)$. Combining the two results and applying the union bound, we get that with probability at least $(1 - 2/n^4)$, the message complexity of the whole algorithm is $O(n(\log w + \log n)$. Similarly, combining Lemma 4.5 and the corresponding analysis for the second phase, we get that with probability at least $(1 - 4/n^4)$, no processor receives more than $O(\log w + \log n)$ messages.

Theorem 4.6 *There exists a constant c such that*

$$\Pr[\textit{Message Complexity} > cn(\log w + \log n)] \leq \frac{2}{n^4}$$
$$\Pr[\text{MaxRcvLoad} > c(\log w + \log n)] \leq \frac{4}{n^4}.$$

It is easy to see that by appropriately choosing the constants, the error probability can be brought down to $1/n^{\tilde{c}}$ for any constant $\tilde{c}$.

5 ST-Rand Algorithm

In this section, we present a randomized algorithm called ST-Rand for the DTC problem. The ST-Rand algorithm is implicit in the work of Emek and Korman [7]; we present an explicit description for the sake of completeness.

The main characteristic of this algorithm is that with high probability, its MaxMsgLoad is at most $O(\log n \log w)$. However, this is an approximate algorithm which works under the reasonable assumption that $w \leq 2^{n^{O(1)}}$. For ease of exposition, we discuss the algorithm and its analysis for the case where $w \leq 2^n$.

Let Δ be any positive integer. We say that a communication interconnection network is Δ-*ST-Embeddable*, if the network has spanning tree in which every vertex has degree at most Δ. For any fixed constant Δ, the ST-Rand algorithm works on any Δ-ST-Embeddable tree and provides the $O(\log n \log w)$ bound on MaxMsgLoad. For the ease of exposition, we will describe the algorithm for the case of $\Delta = 2$, i.e., we consider any network with a Hamiltonian cycle so that a ring can be emebedded in the network. The case of arbitrary Δ is derived via a simple modification of the algorithm.

5.1 Algorithm

We arrange the n processors in a one dimensional ring, so that each processor can communicate with its adjacent neighbors. The algorithm proceeds in multiple rounds. Let $\hat{w}$ denote the initial value of any round, i.e., the number of triggers yet to be received by the system. For the first round, we set $\hat{w} = w$. Each processor x maintains a counter $C(x)$, which is used to keep track of

the number of the triggers received by x since the beginning of the current round. The counter is reset to zero in the beginning of any round.

The behavior of any processor x is as follows. Whenever x receives a trigger, it increments $C(x)$. Then, with probability $(8 \log n/\hat{w})$, the processor x invokes a COLLECTION subroutine, described next [3].

The COLLECTION subroutine: The goal of the subroutine is to compute the total number of triggers received so far in the current round by the system. Let this count be w', which is given by $w' = \sum_x C(x)$. This count w' can easily be computed by performing a reduce operation over the communication ring. Meaning, the processor x sends its $C(x)$ value to its right neighbor y. The processor y adds its own $C(y)$ to $C(x)$ and sends the sum to its right neighbor. In general, any processor z, upon receiving the sum, adds its own $C(z)$ to the sum and passes it to its right neighbor. This way, the originating processor x can get the value w'.

Then, processor x behaves as follows. If $w' \geq \lceil \hat{w}/2 \rceil$, processor x invokes the end-of-round procedure. Here, processor x updates the initial value for the next round by computing $\hat{w} = \hat{w} - w'$. The new $\hat{w}$ is notified to all the processors via point to point messages along the ring. Notice that the COLLECTION subroutine and the end-of-round procedure involves only constant number of messages per processor.

At the end of any round, if the newly computed $\hat{w} \leq 0$, we know that all the w triggers have been received. So, processor x can raise an alert to the user and the algorithm is terminated.

Remark: Notice that the use of a ring for communication is not crucial to the algorithm. Any interconnection network that allows for executing the COLLECTION and end-of-round notification procedures with only a constant number of message exchanges per processor is sufficient for our purpose. In particular, it would suffice if a constant degree spanning tree is embeddable is embeddable in the network (i.e., Δ-ST-Embeddable networks, for some fixed constant Δ).

5.2 Bound on the Failure Probability

There is a possibility that the system may fail to raise the alert upon receiving w triggers. In this section, we derive bounds on the failure probability of the ST-RAND algorithm. Let us first figure out a bound on the number of rounds taken by the algorithm. Observe that in successive rounds the initial value drops by a factor of two. Thus, the algorithm takes at most $\log w$ rounds.

Consider any particular round and let $\hat{w}$ be the initial value of the round. Let us compute the probability that the algorithm enters a deadlock and fails to proceed to the next round, even after all the $\hat{w}$ triggers are delivered. Consider the system after $\hat{w}/2$ are delivered to the system. Notice that the algorithm fails to raise an alert on $\hat{w}$ triggers only when the COLLECTION subroutine is not invoked for any of the remaining $\hat{w}/2$ triggers. Recall that upon receiving a trigger, any processor invokes the COLLECTION subroutine with probability $(8 \log n/\hat{w})$. Thus, the probability that the COLLECTION subroutine is not called for any of the remaining $\hat{w}/2$ triggers is at most $(1 - 8 \log n/\hat{w})^{\hat{w}/2}$.
So,

$$\begin{aligned} &\Pr[\text{deadlock happens in any round}] \\ &\leq (1 - 8 \log n/\hat{w})^{\hat{w}/2} \leq e^{-4 \log n} = 1/n^4. \end{aligned}$$

As argued before, the algorithm takes only $O(\log w)$ rounds to terminate. Applying the union

[3] Here, we assume that $\hat{w} > 8 \log n$. For otherwise, we can use the trivial centralized algorithm and obtain the stated message bounds

bound, we see that

$$\Pr[\text{There exists a round which enters deadlock}] \leq (\log w)/n^4 \leq 1/n^3.$$

The last inequality follows from the assumption that $w \leq 2^n$.

5.3 Bound on the MaxMsgLoad

Here, we derive an upper bound on the number of messages sent and received by any processor. First, let us consider the case when there is no failure, i.e., every round terminates within $\hat{w}$ triggers received by the system, where $\hat{w}$ is the initial value of the round. From Section 5.2, this happens with probability at least $1 - (1/n^3)$.

Consider any round. Let M be the random variable denoting the number of times the Collection subroutine is invoked in this round. Then, M is binomially distributed with expectation

$$\mathbf{E}[M] \leq \frac{8 \log n}{\hat{w}} \cdot \hat{w} = 8 \log n$$

Thus, applying the Chernoff bound (given by Theorem 3.2 – taking $r = 6$), we see that

$$\Pr[M \geq 48 \log n] \leq 2^{-48 \log n} \leq n^{-48}.$$

Since there are at most $\log w$ rounds, applying the union bound, we see that

$$\Pr[\text{There exists a round with } M \geq 48 \log n] \leq (\log w)/n^{48} \leq 1/n^{47}.$$

The last inequality follows from our assumption that $w \leq 2^n$.

The Collection subroutine and the end-of-round procedure involves only a constant number of messages being sent and received per processor. Therefore,

$$\Pr[\text{MaxMsgLoad} \geq a \log n \log w] \leq 1/n^{47},$$

where a is some constant.

6 Handling Concurrency

In the previous sections, we assumed that the triggers are delivered one at a time; meaning, all the processing required for handling a trigger is completed before the next trigger arrives. Here, we relax this assumption and discuss how to handle the concurrency issues. We first discuss handling of concurrency issues for the LayeredRand algorithm in detail. We then discuss the changes for the CoinRand and the ST-Rand algorithms. All triggers and coin messages received by a processor can be placed into a queue and processed one at a time. Thus, there is no concurrency issue related to triggers and coins received within a round. However, concurrency issues need to be handled during an end-of-round. Towards this goal, we slightly modify the LayeredRand algorithm. The core functioning of the algorithm remains the same as before; we mainly modify the end-of-round procedure by adding some additional features (such as counters and queues). The rest of this section explains these features and the end-of-the round procedure in detail. We also prove correctness of the algorithm in the presence of concurrency issues.

6.1 Processing Triggers and Coins

Each processor x maintains two FIFO queues - a *default* queue and a *priority* queue. All triggers and coin messages received by a processor are placed in the default queue. The priority queue contains only the messages related to the end-of-round procedure, which are handled on a priority basis. In the main event handling loop, a processor repeatedly checks for messages in queues. It first examines the priority queue and handles the first message in that queue, if any. If there is no message there, it examines the default queue and handles the first message in that queue (if any).

Every processor also maintains a counter $D(x)$ that keeps a count of triggers directly received and processed by x, since the beginning of the algorithm. The triggers received by x that are in the default queue (not yet processed) are not accounted in $D(x)$. The counter $D(x)$ is incremented every time the processor processes a trigger from the default queue. This counter is never reset. It is maintained in addition to the counter $C(x)$ (which gets reset in the beginning of each round).

Every processor x maintains another variable, RoundNum, that indicates the current round number for this processor. Whenever x sends a coin to some other processor, it includes its RoundNum in the message. The processing of triggers and coins is done as before (as in Section 3).

6.2 End-of-round Procedure

Here, we describe the end-of-round procedure in detail, highlighting the modifications. The procedure consists of four phases. The processors are arranged in the form of a binary tree as before.

In the first phase, the root processor broadcasts a *RoundReset* message down the tree to all nodes requesting them to send their $D(x)$ counts. In the second phase, these counts are reduced at the root using *Reduce* messages; the root computes the sum of $D(x)$ over all the processors. Note that, unlike the algorithm described in Section 3, here the root computes the sum of $D(x)$ counters, rather than the sum of $C(x)$ counters. We shall see that this is useful in proving correctness. Using the sum of $D(x)$ counters, the root computes the initial value $\hat{w}$ for the next round. In the third phase, the root broadcasts this value $\hat{w}$ to all nodes using *Inform* messages. In the fourth phase, each processor sends an acknowledgement *InformAck* back to the root and enters the next round.

We now describe the four phases in detail, that will help in proving correctness of the algorithm. **First Phase:** In this phase, the root processor initiates the broadcast of a *RoundReset* message by sending it down to its children. A processor x on receiving *RoundReset* message, does the following:

- At this point, the processor suspends processing of the default queue until the end-of-round processing is completed. Thus all new triggers are queued up without being processed. This ensures that the $D(x)$ value is not modified while end-of-round procedure is in progress.
- If x is not a leaf processor, it forwards the *RoundReset* message to its children; if it is a leaf-processor, it initiates the second phase as described below.

Second Phase: In this phase, the $D(x)$ values are sum-reduced at the root from all the processors. The second phase starts when a leaf processor receives a *RoundReset* message, in response to which it initiates a *Reduce* message containing its $D(x)$ value and passes it to its parent. When a non-leaf processor has received *Reduce* messages from all its children, it adds up the values in these messages to its own $D(x)$ and sends a *Reduce* message to its parent with this sum. Thus, the root collects the sum of $D(x)$ over all the processors. This sum w' is the total numbers of triggers received in the system so far. Subtracting w' from w, the root computes the initial value $\hat{w}$ for the next round. If $\hat{w} = 0$, the root raises an alert and terminates the algorithm. Otherwise, the root initiates the third phase.

Third Phase: In this phase, the root processor broadcasts the new $\hat{w}$ value by sending an *Inform* message to its children. A processor x on receiving the *Inform* message, performs the following:

- It computes the threshold $\tau(\ell)$ value for the new round, where ℓ is the layer number of x.
- If x is a non-leaf processor, it forwards the *Inform* message to its children; if x is a leaf processor, it initiates the fourth phase as described below.

Fourth Phase: In this phase, the processors send an acknowledgement upto the root and enter the new round. The fourth phase starts when a leaf processor x receives an *Inform* message. After performing the processing for the *Inform* message, it performs the following actions:

- It increments RoundNum. This signifies that the processor has entered the next round. After this point, the processor does not process any coins from the previous rounds. Whenever the processor receives a coin generated in the previous rounds, it simply discards the coin.
- $C(x)$ is reset to zero.
- It sends an *InformAck* to its parent.
- The processor x resumes processing of the default queue. This way, x will start processing the outstanding triggers (if any).

When a non-leaf node receives *InformAck* messages from all its children, it performs the same processing as above. When the root processor has received *InformAck* messages from all its children, the system is said to have entered the new round.

We note that it is possible to implement the end-of-round procedure using three phases. However, the fourth phase (of sending acknowledgements) ensures that at any point of time, the processors can only be in two different (consecutive) rounds. Moreover, when the root receives the *InformAck* messages from all its children, all the processors in the system are in the same round. As a result, end-of-round processing for different rounds cannot be in progress at the same time.

6.3 Correctness of Algorithm

We now show that the system correctly raises an alert to the user when all the w triggers are delivered. The main part of the proof involves showing that after starting a new round, the root always enters the end-of-round procedure. Furthermore, we also show that system does not incorrectly raise an alert to the user before w triggers are delivered.

We say that a trigger is *unprocessed*, if the trigger has been delivered to a processor and is waiting in its default queue. A processor is said to be in round k, if its RoundNum variable is k. A trigger is said to be processed in round k, if the processor that received this trigger is in round k when it processed the trigger.

Consider the point in time t when the system has entered a new round k. Let $\hat{w}$ be the initial value of the round. Recall that in the second phase, the root computes $w' = \sum_{x \in \mathcal{P}} D(x)$ and sets $\hat{w} = w - w'$, where $\mathcal{P}$ is the set of all processors. Notice that in the first phase, all processors suspend processing triggers from the default queue. The trigger processing is resumed only in the fourth phase after the RoundNum is incremented. Therefore, no more triggers are processed in the round $k-1$. It follows that w' is the total number of triggers that have been processed in the (previous) rounds $k' \leq k-1$. Thus, any triggers processed in round k will be accounted in the counter $C(x)$ of some processor x. This observation leads to the following argument.

We now show that the root initiates the end-of-round procedure upon receiving at most $\hat{w}$ triggers. Suppose all the $\hat{w}$ triggers have been delivered and processed in this round. Furthermore, assume that all the coins generated and sent in the above process have also been received and

processed. Clearly, such a state will happen at some point in time, since we assume a reliable communication network. At this point of time, we have $\sum_{x \in \mathcal{P}} C(x) = \hat{w}$.

At any point of time after t, we have $\sum_{x \in \mathcal{P}-\{r\}} C(x) \leq \hat{w}/4$, where $\mathcal{P}$ is the set of all processors and r is the root processor. The claim is proved using the same arguments as in Section 3.2 and the fact that the processors discard the coins generated in previous rounds.

From the above relations, we get that $C(r) \geq 3\hat{w}/4 \geq \lceil \hat{w}/2 \rceil$. The root initiates the end-of-round procedure whenever $C(r)$ crosses $\lceil \hat{w}/2 \rceil$. Thus, the root will eventually start the end-of-round procedure. Hence the system never gets stalled in the middle of the round. Clearly, the system raises an alert on receiving w triggers.

We now argue that the system does not raise an alert before receiving w triggers. This follows from the fact that $\hat{w}$ for a new round is calculated on the basis of $D(x)$ counters.

The analysis of message complexity and MaxRcvLoad are unaffected.

6.4 Handling concurrency in CoinRand

As the end-of-round procedure is the same in the CoinRand algorithm, concurrency is handled in the same way in this algorithm as it is done in the LayeredRand algorithm. Thus, the same counters, queues and phases are used in CoinRand as well. The correctness of the algorithm can be shown in a similar manner combining the techniques of Section 6.3 with the arguments in Sections 4.2 and 4.6.

6.5 Handling concurrency in ST-Rand

Concurrency handling in the ST-Rand algorithm is much simpler than the LayeredRand and CoinRand algorithms. The simplicity stems from the fact that each node only communicates with its right neighbor. The queues (default and priority queues) and counters ($C(x)$ and $D(x)$) are maintained as before. However, the collection procedure along with the end-of-round procedure now only require two phases. The first phase is a culmination of the first two phases and the second phase is a culmination of the last two phases of the modified concurrency handling procedure described for the LayeredRand algorithm.

First Phase: In the first phase, the initiator of the Collection operation sends a *Reduce* message to its right neighbor with a value of 0. A processor x on receiving *Reduce* message, does the following:

- At this point, the processor suspends processing of the default queue until the Collection operation is completed. Thus all new triggers are queued up without being processed. This ensures that the $D(x)$ value is not modified while the Collection operation is in progress.
- It adds up the value in the received message to its own $D(x)$ and sends a *Reduce* message to its right neighbor with this sum.

This phase completes when the initiator receives back the *Reduce* message. Thus, the initiator eventually collects the sum of $D(x)$ over all the processors. This sum w' is the total numbers of triggers received in the system so far.

Second Phase: The initiator then initiates the second phase. It determines if an end-of-round procedure is to be performed or not, i.e., if $w' \geq \lceil \hat{w}/2 \rceil$ or not. It then sends an *Inform* message to its right neighbor with a value of 1 if an end-of-round is to be performed and a value 0 if an end-of-round is not to be performed. A processor x on receiving the *Inform* message, performs the following:

- If the message indicates end-of-round,

  - It increments RoundNum. This signifies that the processor has entered the next round.
  - It computes the threshold value for the new round.
  - $C(x)$ is reset to zero.
- It forwards the *Inform* message to its right neighbor.
- The processor x resumes processing of the default queue. This way, x will start processing the outstanding triggers (if any).

This phase completes when the initiator receives back the *Inform* message. If the end-of-round was performed, the system is said to have entered the new round.

The only other issue that remains to be resolved is what happens when two or more processes simultaneously initiate the COLLECTION procedure. In this case, all but one of the COLLECTION procedures are aborted. This is done by giving preference to the COLLECTION procedure initiated by the processor with a smaller id (unique processor number or MPI rank). Hence, this requires the id of the processor initiating COLLECTION is also sent along with the *Reduce* message. As the COLLECTION procedures are performed over the same ring proceeding in the same direction, the node initiating the COLLECTION procedure will be the first node to receive the *Reduce* message for a different COLLECTION procedure. Thus, the resolution can be done by the COLLECTION initiating nodes themselves. This resolution is done as follows. When a node that has initiated a COLLECTION procedure, with id i, receives a *Reduce* message for a COLLECTION procedure initiated by a node with id j, it compares i with j. If $i < j$, it discards the newly received *Reduce* message. If on the other hand $j < i$, it processes the message as it would handle a fresh *Reduce* message and forwards it; the other nodes receiving the message also process and forward it accordingly. It is easy to see that the first phase will complete only on one of the nodes initiating the COLLECTION procedure; the one with the smallest id. Note that whenever a node processes a *Reduce* message, it updates the sum counter in the message with its $D(x)$ counter and all other triggers wait in the default queue unprocessed by the processor. Hence, when the first phase completes, then the root correctly has the count of w', the number of triggers processed by the system and all triggers unaccounted in w' lie in the default queue of the processors. The correctness can then again be shown along the same lines as that of LAYEREDRAND by combining the techniques of Section 6.3 with the arguments in Section 5.

References

[1] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks. Local management of a global resource in a communication network. *J. ACM*, 43(1):1–19, 1996.

[2] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[3] V. Chakaravarthy, A . Roy Choudhury, Y. Sabharwal, and V. Garg. An efficient decentralized algorithm for the distributed trigger counting problem. In *ICDCN*, 2011.

[4] V. Chakaravarthy, A. Roy Choudhury, and Y. Sabharwal. Improved algorithms for the distributed trigger counting problem. In *IPDPS*, 2011.

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

[6] Y. Emek and A. Korman. New bounds for the controller problem. In *Proceedings of the 23rd International symposium on distributed computing*, 2009.

[7] Y. Emek and A. Korman. Efficient threshold detection in a distributed environment: extended abstract. In *PODC '10: Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010.

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

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

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

[11] A. Korman and S. 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, 2007.

[12] B. Mans, S. Schmid, and R. Wattenhofer. Distributed disaster disclosure. In *SWAT*, 2008.

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

[14] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

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