An Efficient Decentralized Algorithm 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 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, synchronizers and other distributed settings. The main result of the paper is a decentralized and randomized algorithm with expected message complexity $O(n \log n \log w)$. Moreover, every processor in this algorithm receives no more than $O(\log n \log w)$ messages with high probability. All the earlier algorithms for this problem have maximum message load of $\Omega(n \log w)$.

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. We note w may be much larger than n. 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,

© Springer-Verlag Berlin Heidelberg 2011

example applications are monitoring the volume of traffic or the number of remote logins. See, for example, [7] 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. [4] 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 [1], 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.

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

– The *message complexity*, i.e., the number of messages exchanged between the processors.
– The MaxRcvLoad, i.e., the maximum number of messages received by any processor in the system.

Garg et al. [4] studied the DTC problem for a general distributed system. They presented two algorithms: a centralized algorithm and a tree-based algorithm. The centralized algorithm has message complexity $O(n \log w)$. However, the MaxRcvLoad of this algorithm can be as high as $\Omega(n \log w)$. The tree-based algorithm has message complexity $O(n \log n \log w)$. This algorithm is more decentralized in a heuristic sense, but its MaxRcvLoad 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 must have message complexity $\Omega(n \log(w/n))$. So, the message complexity of the centralized algorithm is optimal asymptotically. However, this algorithm has MaxRcvLoad as high as the message complexity.

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 assume an asynchronous model of computation and messages. We assume that the messages are guaranteed to be delivered but there is no fixed upper bound on the message arrival time. Also, messages are not corrupted or spuriously introduced. This setting is common in data networks. We also assume that there are no faults in the processors and that the processors do not fail.

Our main result is a decentralized randomized algorithm called LayeredRand that is efficient in terms of both the message complexity and MaxRcvLoad. Its message complexity is $O(n \log n \log w)$. Moreover, with high probability, its MaxRcvLoad is $O(\log n \log w)$. The message complexity of our algorithm is the same as that of the tree based algorithm of Garg et al. [4]. However, the MaxRcvLoad of our algorithm is significantly better than both their tree based and centralized algorithms. It is important to minimize MaxRcvLoad for many applications. For example, in sensor networks where the message processing may

Algorithm	Message Complexity	MaxLoad
Tree-based[4]	$O(n \log n \log w)$	$O(n \log n \log w)$
Centralized[4]	$O(n \log w)$	$O(n \log w)$
LayeredRand	$O(n \log n \log w)$	$O(\log n \log w)$

Fig. 1. Summary of DTC Algorithms

consume limited power available at the node, a high MaxRcvLoad may reduce the lifetime of a node.

Another important aspect of our algorithm is its simplicity. In particular, our algorithm is much simpler than both the algorithms of Garg et al. A comparison of our algorithm with the earlier results is summarized in Fig. 1. Designing an algorithm with message complexity $O(n \log w)$ and MaxRcvLoad $O(\log w)$ remains a challenging open problem.

Our main result is formally stated next. 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.*

Theorem 1. *Fix any trigger pattern. The message complexity of the* LayeredRand *algorithm is* $O(n \log n \log w)$*. Furthermore, there exist constants* c *and* $d \geq 1$ *such that*

$$\Pr[\text{MaxRcvLoad} \geq c \log n \log w] \leq \frac{1}{n^d}.$$

The above bounds hold for any *trigger pattern*, even if fixed by an adversary.

Related work. Most prior work (e.g. [3,7,6]) 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 [2,9,10] and the framework of repeated global computation [5] 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.

2 A Deterministic Algorithm

For the DTC problem, Garg et al. [4] presented an algorithm with the message complexity of $O(n \log w)$. In this section, we describe a simple alternative deterministic algorithm having the same message complexity. The aim of presenting this algorithm is to highlight the difficulties in designing an algorithm that simultaneously achieves good message complexity and MaxRcvLoad bounds.

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. 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

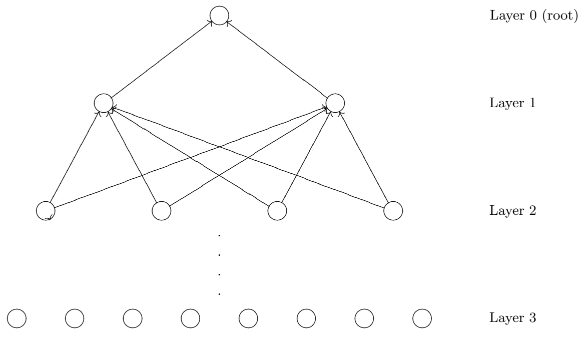

Fig. 2. Illustration for LayeredRand

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 discuss how to handle the concurrency issues in Sect. 5.

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 layer*. The layering is illustrated in Fig. 2, for $n = 15$. Only processors occupying adjacent layers will communicate with each other.

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 $1 \le \ell < L$, 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 $\tau(\ell - 1)$ triggers. After the update, if $C(y) \ge \tau(\ell - 1)$, 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 Fig. 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 the figure.

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 $C(x) \ge \tau(\ell)$,
    - 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 $C(r) \ge \lceil \hat{w}/2 \rceil$), 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.

4 Analysis of the LayeredRand Algorithm

Here, we prove the correctness of the algorithm and then prove message bounds.

4.1 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 $C(x)$:

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

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 enter a dead state.

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

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.

4.2 Bound on the Message Complexity

Lemma 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:

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

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 $O(n \log n \log w)$. □

4.3 Bound on the MaxRcvLoad

In this section, we show that with high probability, the MaxRcvLoad is bounded by $O(\log n \log w)$. We use the following Chernoff bound (see [8]) for this purpose.

Theorem 2 (see [8], 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 2. $\Pr[\text{MaxRcvLoad} \geq c \log n \log w] \quad \leq \quad n^{-47}$*, for some constant c.*

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 found in layer ℓ. Let M_x be the random variable denoting the number of coins received by x. As discussed before, the algorithm takes at most $\log w$ rounds. In any given round, the number of coins received by layer ℓ is at most $\frac{\hat{w}}{\tau(\ell+1)} \leq 4 \cdot 2^{\ell+1} \log n$. Thus, the total number of coins received by layer ℓ is at most $4 \cdot 2^{\ell+1} \log n \log w$. 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

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

The random variable M_x is a sum of independent 0-1 random variables. Applying the Chernoff bound given by Theorem 2 (taking $r = 6$), we see that

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

Applying the union bound, we see that

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

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 $O(\log w)$. □

5 Handling Concurrency

In this section, we discuss how to handle the concurrency issues. 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.

5.1 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 Sect. 3).

5.2 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 Sect. 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.

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 enters 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. Thus, end-of-round processing for different rounds cannot be in progress simultaneously.

5.3 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 equals 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 Sect. 4.1 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 Conclusions

We have presented a randomized algorithm to the DTC problem which reduces the MaxRcvLoad of any node from $O(n \log w)$ to $O(\log n \log w)$ with high probability. The ultimate goal of this line of work would be to design a deterministic algorithm with MaxRcvLoad $O(\log w)$.

References

1. Awerbuch, B.: Complexity of network synchronization. J. ACM 32(4), 804–823 (1985)
2. Chang, E.: Echo algorithms: Depth parallel operations on general graphs. IEEE Trans. Software Eng. 8(4), 391–401 (1982)
3. Cormode, G., Muthukrishnan, S., Yi, K.: Algorithms for distributed functional monitoring. In: SODA (2008)
4. Garg, R., Garg, V.K., Sabharwal, Y.: Scalable algorithms for global snapshots in distributed systems. In: 20th Int. Conf. on Supercomputing, ICS (2006)
5. Garg, V., Ghosh, J.: Repeated computation of global functions in a distributed environment. IEEE Trans. Parallel Distrib. Syst. 5(8), 823–834 (1994)
6. Huang, L., Garofalakis, M., Joseph, A., Taft, N.: Communication-efficient tracking of distributed cumulative triggers. In: ICDCS (2007)
7. Keralapura, R., Cormode, G., Ramamirtham, J.: Communication-efficient distributed monitoring of thresholded counts. In: SIGMOD Conference (2006)
8. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge Univ. Press, Cambridge (2005)
9. Segall, A.: Distributed network protocols. IEEE Transactions on Information Theory 29(1), 23–34 (1983)
10. Tel, G.: Distributed infimum approximation. In: Lupanov, O.B., Bukharajev, R.G., Budach, L. (eds.) FCT 1987. LNCS, vol. 278, pp. 440–447. Springer, Heidelberg (1987)