Adaptive Convergecast by Distributed Topology Switching

Suchetana Chakraborty, Sushanta Karmakar

[1] Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati
Assam, India

{suchetana,sushantak}@iitg.ernet.in

Abstract. *Convergecast in a wireless sensor network is a process in which each sensor node senses the environment and forwards that information to a base station in some way. For correct data gathering using convergecast there should be no data loss and no delivery of redundant data. Sensors can form a spanning tree rooted at the sink (base station) to perform the convergecast in an efficient way. Leaves of the tree can sense and forward data independently. However an internal node forwards data to its parent only after receiving data from all its children. It has been observed that a Breadth-First-Search (BFS) tree is a better choice for convergecast under low system load because the depth of any node from the root is always minimum. However under higher load a Depth-First-Search (DFS) tree may be a better option as the degree of any node in a DFS tree is generally lower than that in a BFS tree. Hence load of each node is lower in case of a DFS tree than that in a BFS tree. Therefore it may be desirable to dynamically switch between a BFS tree and a DFS tree based on load. In this paper we propose a scheme for adaptive convergecast that dynamically switches between a BFS tree and a DFS tree. The switching mechanism remains transparent to the convergecast. Also each convergecast message is correctly delivered to the base station eventually without any loss or redundancy.*

1. Introduction

A sensor is a device which is capable of sensing, processing and transmitting data to other sensors. A group of such sensors form a network which can be used to monitor environment, health, military or critical resources. Sensors have limited processing capacity, memory and they are generally battery powered. Data gathering using convergecast is an important application in sensor networks. In convergecast, each node senses some data, gets data from some other sensors, and fuses them to forward to the base-station. In recent years convergecast has received increasing attention because of significant number of practical applications. One way to achieve an efficient convergecast is to model a sensor network as a tree rooted at the sink. All the leaf nodes of the tree collect and forward data independently, but an intermediate node can forward data to its parent only after receiving from all its children. A correct convergecast process guarantees that all the data from different nodes must reach the sink without any data loss or data redundancy.

1.1. Motivation

The performance of a distributed system depends on its environment which can change with time. For example, average load of a distributed system varies with time as the number of users changes. Therefore it is desirable to design protocols for distributed systems

that can adapt to changing environments. Convergecast generally uses a rooted spanning tree. A fixed data-gathering tree may not be suitable for convergecast under various load conditions. If there exist more than one tree each rooted at the sink node, and each of them is better suited to a particular environment then the system may dynamically switch from one tree to another based on change in environment and thereby can implement an efficient convergecast that adapts to the changes in the environment and thus provides better performance. In this work we consider the existence of two trees, a BFS tree and a DFS tree, both rooted at the sink for data gathering. The convergecast application uses the desired tree based on load. If load is low, it uses a BFS tree as the distance from any node to the root is always minimum in a BFS tree. However for higher load the application uses a DFS tree because the degree of any node is generally lower in a DFS tree than that in a BFS tree. If load changes from low to high, the system switches from the BFS tree to the DFS tree, and similarly vice-versa.

A random switching between these two data-gathering trees would result in an incorrect convergecast due to data redundancy or loss of data. The problems are illustrated in Figure 1 and Figure 2. In the figures a dashed link indicates a communication link between two nodes and an arrow indicates an edge of the tree.

- **Redundant data:** Let at time t convergecast was using the BFS tree of Figure 1(a) and at that moment leaf node 7 has sent data to node 3. Now suppose the system starts using the DFS tree of Figure 1(b) due to a random switching. So at time $t + 1$, node 3 will try to forward the data to its current parent, which has changed from node 1 to node 7. Hence the same data that node 7 had sent to node 3 will come back to node 7 as a duplicate. Thus in general, whenever the parent-child relationship between any two nodes gets reversed due to a random switching, the correctness of the convergecast will be compromised due to redundant data transmission in the network.

- **Data loss:** Suppose the system is using a DFS tree as shown in Figure 2(b). At some time t node 3 has sent data to its parent, node 5. However the data has not yet reached node 5 because of the asynchronous property of network channel. Now a random switch occurs from the DFS tree to the BFS tree (Figure 2(a)) . After the switching node 5 has got two children i.e. node 6 and node 7. As soon as it receives data from both of its children it will fuse and forward to its parent which is node 2. So even if data from node 3 reaches to node 5 eventually, it will not consider it as node 3 is not its child any more. So the data that node 3 sent to node 5 will be lost in this process and will never reach to the sink.

1.2. The Problem Definition

A correct convergecast requires that all the data from the different nodes must reach the sink without any data loss or redundancy. Also each node can send data only once. Different topologies of the underlying data-gathering network will be useful in different scenarios. The main objective of the work is to propose a scheme for convergecast that adapts to the load of the network by dynamically switching between different topologies and still guarantees a correct convergecast. Depending on the system load the adaptive convergecast protocol uses either a BFS tree (at lower load) or a DFS tree (at higher load). However a random switching from a BFS tree to a DFS tree or vice-versa can cause the

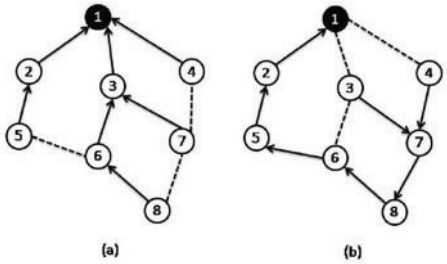

Figure 1. Redundant Data Delivery

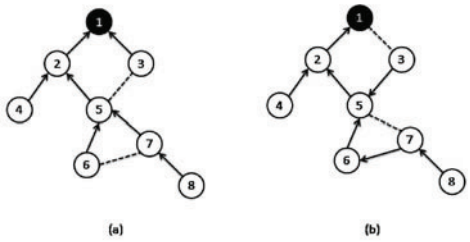

Figure 2. Data Loss

problem of redundant data delivery, data loss, or an indefinite stall of data flow. In this paper a distributed message-passing algorithm has been proposed to switch from a BFS tree to a DFS tree (similarly vice-versa) such that the application layer convergecast remains unaffected. The proposed algorithm performs the switching between a BFS tree and a DFS tree while assuring that all the properties of a correct convergecast are maintained. Even if there is a temporary blocking of data at some particular node, it resumes within a finite amount of time without any data loss. Also the stall is temporary and local to some node.

2. Related Work

Convergecast in wireless sensor network has received significant research attention. [Annamalai et al. 2003] proposed an algorithm for tree construction and TDMA based channel allocation for collision free convergecast to achieve greater efficiency in terms of latency and power consumption. The algorithm constructs a tree level by level and allocates a schedule for each node that specifies the time-slot(s) in which it can transmit data. [Upadhayayula et al. 2003] proposed another algorithm to minimize energy consumption during communication, while ensuring low-latency through faster data transfer and reliability through collision-free transmission. The motivation for improving the latency is by constructing a balanced tree such that it enhances the likelihood of multiple simultaneous transmissions in a given time slot. [Krishnamachari et al. 2002] proposed a data-centric routing scheme and studied the energy-latency trade-off involved in data-aggregation that includes the effect of source-sink placements, communication network topology and density of the network on convergecast. They have shown that the latency of convergecast is proportional to the number of hops between the sink and the furthest source and the formation of an optimal data-gathering tree is a NP-hard problem. So they proposed a heuristic based polynomial time algorithm to create a data-aggregation tree. However none of the aforesaid works proposed an adaptive model of convergecast which can cope with changing environments.

[Heinzelman et al. 1999] proposed a family of adaptive protocols, named SPIN (Sensor Protocols for Information via Negotiation), for data dissemination in wireless sensor network. To eliminate the transmission of redundant data flow in the network, the protocol uses high level data descriptors called meta-data. The performance of SPIN outperforms the traditional approaches like flooding or gossiping using its key features such as meta-data negotiation and resource adaptation. The proposed algorithm is energy-aware but useful only for highly available bandwidth. Also communication cost is comparatively higher.

[Liu and van Renesse 2000] worked on the adaptivity and coordination among running protocols. They have built a hybrid protocol that can make smooth adaptation at run-time. The algorithm works in three steps and has small switching overhead and is scalable and delay-efficient compared to the traditional two-phase commit protocol. However the protocols used are abstract in nature, homogeneous and application specific. In another paper, [Liu et al. 2001] designed a generic switching protocol which assures to preserve some of the communication properties like reliability, total order, integrity, no replay, confidentiality etc. The switching protocol runs below the application layer and is transparent. However the properties considered are application specific and thus have limited applicability in general. [Mocito and Rodrigues 2006] have proposed an adaptive protocol that is able to dynamically switch between different total-order algorithms. The assurance is that the flow of application messages do not stop even during the switching. To achieve smooth transition, all nodes need to agree on the point in the message flow where they switch. Also, the proposed protocol does not allow concurrent adaptation and offers low overhead as long as there is enough network bandwidth to support the transmission of data in parallel during the reconfiguration. Furthermore this is applicable only for broadcast.

[Yacoab and Sundaram 2010] proposed an adaptive traffic-aware data aggregation technique for wireless sensor network. In the proposed scheme, a traffic monitoring agent is used to monitor the load and a multipath structured tree is constructed in which nodes are selected based on their residual energy level. If the total traffic load of the system is less than a threshold value, then the structured lossless aggregation is applied, otherwise the aggregation technique is adaptively changed to structure-free lossy aggregation. [Chen et al. 2008] proposed an adaptive data-gathering scheme for clustered WSN. The objective was to shift the burden of computation from ordinary sensor nodes to the resource-rich sink node through proper adjustment of aggregation ratio and reporting frequency. The spatial and temporal aggregation degree is adaptive to the dynamic state of WSN via the interaction between the sink node and the clusterheads.

[Karmakar and Gupta 2007] designed a distributed protocol switching algorithm for broadcast applications that switches between a BFS tree and a DFS tree for adapting to the system load. At low load a BFS tree is used as it reduces the broadcast delay. However at higher load a DFS tree is used to since the degree of node in a DFS tree is generally lower than that of a BFS tree. In the proposed algorithm, the switching is done adapting to the network load as well as without affecting the application layer broadcasting. The algorithm also ensures the correct delivery of each broadcast packet and guarantees that some spanning tree of the graph is always maintained even at the time of switching. However, this problem gets more challenging when the application is convergecast. Data packets are forwarded from the leaf nodes to the root in case of convergecast. If the switching occurs in the reverse direction, then a lot of co-ordinations among the nodes are needed to achieve a successful switching while assuring the correctness of the algorithm. The challenge also lies in the fact that for convergecast there are multiple initiators of the data-gathering process i.e. multiple leaf nodes in the system.To the best of our knowledge there is no algorithm for adaptive convergecast that switches between two trees. In this paper we have proposed a distributed algorithm for load-adaptive convergecast using tree switching.

3. System Model

Let there be n number of sensor nodes placed randomly in the environment that is being monitored. The random distribution of the sensor nodes are represented by a communication graph $G(V, E)$ where V represents the set of sensor nodes and E be the set of communication links between the nodes. $N(v)$ denotes the neighbor set for any node v. The switching algorithm is used to switch between a BFS tree and a DFS tree (or vice-versa) depending on the load of the system. The BFS tree and the DFS tree are assumed to be precomputed for a given network topology using some standard algorithm and both are rooted at the sink node. The computation model is assumed to be asynchronous and the network is static. So the per hop message delay is finite but unbounded. Also it is assumed that the channel is reliable and FIFO as well as there is no node or link failure in the network.

4. Algorithm Design

4.1. Tree Switching Algorithm

The task of switching between a BFS tree and a DFS tree has two components. The first component is about *when* to switch. This component detects the change in environment that triggers the switching. In this work we assume that there exists an oracle at the sink node that decides the time when the switching should start. The second component deals with *how* to switch from one tree to another. In this paper we concentrate on this component. The proposed algorithm is a distributed message passing algorithm which works in two phases. Let T and T' denote the BFS and DFS tree respectively.

- **First Phase:** For a switch from tree T to T', sink node (root) passes a *TOKEN* to its children of new tree T'. As the *TOKEN* traverses downwards, the virtual new tree T' is constructed from the root to the leaves. However the convergecast still follows the old tree T.
- **Second Phase:** Leaf nodes start passing back the *TOKEN* to their respective parents for the switched tree T' through the return paths. Each node thus on receiving *TOKEN* back from all its children makes the link to its parent permanent, which is an edge of the new tree T'. Now the data packets start flowing through these newly built paths. In this way the switching from T to T' occurs gradually from the bottom of the tree towards the root.

Therefore it is evident that at some point of time data gathering is using T' at the bottom levels of the tree and T at the the upper ones. However the algorithm assures that convergecast process does not get affected in any way and also the tree properties are maintained all the time. Three types of messages are exchanged among the nodes to complete the switching process.

- Each node sends *TOKEN* message to all its children for the switched tree and thus making a virtual construct of the switched tree. Also each node receives back the *TOKEN* from all its children in second phase to make sure that the paths from the leaves to that particular node have already been permanent. The *TOKEN* is generated by the root to initiate the switching and finally consumed back by the root itself completing the process.

- When a node wants to change its parent pointer due to switching, it first sends a *CANCEL* message to its old parent. Also when an intermediate node becomes a leaf node for the switched tree, it sends *CANCEL* to all its old children from whom it used to receive data packets. Upon receiving *CANCEL* from a child, a node removes that particular node from its child set. Again upon receiving *CANCEL* from parent, a node starts buffering incoming application data.
- A node sends *ACKC* to its parent as an acknowledgment to the *CANCEL* message. Upon receiving *ACKC* from all the expected neighbors, a node can guarantee that it is not going to receive any more data packets from any of those neighbors and hence changes its parent pointer for the sake of switching.

Each node has a set a of variables whose descriptions are as follows:

- p_{curr} : Parent variable for the current tree T.
- p_{new} : Parent variable for the switched tree T'.
- C_{curr} : Set of children for the current tree T.
- C_{new} : Set of children for the switched tree T'.
- $Counter1$: Integer variable, keeps the track of the cardinality of the set C_{new}.
- $Counter2$: Integer variable, keeps the track of the cardinality of the set C_{curr}.
- $BLOCK$: Boolean variable, if FALSE then the system dequeues application messages from the Buffer.

$Counter1$ and $Counter2$ are initialized to 0 and p_{new} to NULL. p_{curr} and p_{new} is NULL for the root node. Root node initiates the algorithm by sending *TOKEN* to all v where $v \in C_{new}$ and when it receives back the *TOKEN* from all v where $v \in C_{new}$ then the algorithm terminates. The formal description of the algorithm for BFS to DFS switching or vice-versa is given in Figure 3, 4 and 5

Let each convergecast message coming from the application layer be denoted by M. The tree switching algorithm runs as a middleware layer below the application layer. So upon receiving M from the application layer, the switching algorithm, based on local computation, either forwards it or starts buffering. If $BLOCK = FALSE$ then it simply forwards to its current parent, otherwise it starts buffering. Each node has a buffer to store incoming data from application layer. The buffer is implemented as a queue. FRONT and REAR denote the first and last pointers of the queue respectively and both are initialized to 0. The formal algorithm for handling application messages is given in Figure 6.

4.2. Correct Delivery of Convergecast Messages

The problem of application data loss due to switching can be easily avoided through duplicate transmission. However this may increase the latency and traffic overhead of the system. Hence in this work we assume that a node does not send the same convergecast message to its parent more than once. With this condition, only the proper design of the switching algorithm can guarantee a correct convergecast. The illustration in Figure 7 is for the switching from a BFS tree to a DFS tree to show that the switching gets completed successfully as well as the properties of the convergecast are also satisfied.

- **No Redundant Data:** For T node 3 is the parent of node 4, but in T' this parent-child relationship gets reversed. So duplicate data may generate as a consequence. But on receiving *CANCEL* from node 3, node 4 starts buffering data. Receiving

```
ON RECEIVING TOKEN FROM U
 1  if u ∈ C_new
      then
 2        C_curr ← u
 3        Counter1 ← Counter1 − 1
 4        if Counter1 = 0
            then
 5              if p_curr ≠ p_new
                  then
 6                    Send (CANCEL,p_curr)
 7                    p_curr ← p_new
 8              endif
 9        endif
10        Send (TOKEN,p_new)
11        if BLOCK = TRUE
            then
12              BLOCK ← FALSE
13        endif
14    else
15        p_new ← u
16        Send (TOKEN,v) where v ∈ C_new
17        Counter1 ← |C_new|
18        if C_new = φ
            then
19              if C_curr = φ
                  then
20                    if p_curr ≠ p_new
                        then
21                          Send (CANCEL,p_curr)
22                    endif
23                else
24                    Send (TOKEN,p_new)
25              endif
26          else
27              Send (CANCEL,v) where v ∈ C_curr
28              Counter2 ← |C_curr|
29        endif
30  endif
```

Figure 3. On receiving *TOKEN* from u

ACKC confirms node 3 that no more data is going to come from node 4. Before receiving *ACKC*, node 3 forwards all data through its old parent node 1, and after receiving *ACKC*, to its new parent node 4. So no data sent from node 4 to node 3

```
ON RECEIVING CANCEL FROM U
1  if u = p_curr
       then
2          BLOCK ← TRUE
3          Send (ACKC,u)
4  endif
5  if u = C_curr
       then
6          C_curr ← {C_curr − u}
7  endif
```

Figure 4. On receiving *CANCEL*

```
ON RECEIVING ACKC
1  Counter2 ← Counter2 − 1
2  if Counter2 = 0
       then
3          Send (CANCEL,p_curr)
4          Send (TOKEN,p_new)
5  endif
```

Figure 5. On receiving *ACKC*

```
ON RECEIVING M
1  Buffer[REAR] ← M
2  REAR ← REAR + 1
3  if BLOCK = FALSE
       then
4          Send (M, p_curr)
5          FRONT ← FRONT + 1
6  endif
```

Figure 6. On receiving application message

can come back to node 4 as duplicate.

- **No Data Loss:** Node 3 had two children, node 4 and node 7 in T, whereas it is a leaf node in T'. Data sent from a child could get lost if its parent does not exist any more due to a random switch. However node 3 first makes sure through *ACKC* that no more data is going to come from its current child and then only it can remove the link to its current parent. All the data received before receiving *ACKC* is thus forwarded through the old path.
- **No Indefinite Stall:** A node starts buffering data after receiving either a *TOKEN* from its C_{new} or a *CANCEL* from its p_{curr}. Also a node informs its p_{curr} through *CANCEL* message before changing its parent variable. Thus a node does not wait indefinite time (network stall) expecting data from its child.

4.3. Proof of Correctness

Lemma 4.3.1 *Each node receives TOKEN exactly* $|C_{new}|+1$ *times, one from its* p_{new} *and each one from* u *where* $u \in C_{new}$.

Proof The root node initiates the switching algorithm by sending *TOKEN* to all u where $u \in C_{new}(root)$ independently. At the second phase of the algorithm leaf nodes send back *TOKEN* to their respective p_{new} independently. Each intermediate node receives *TOKEN*

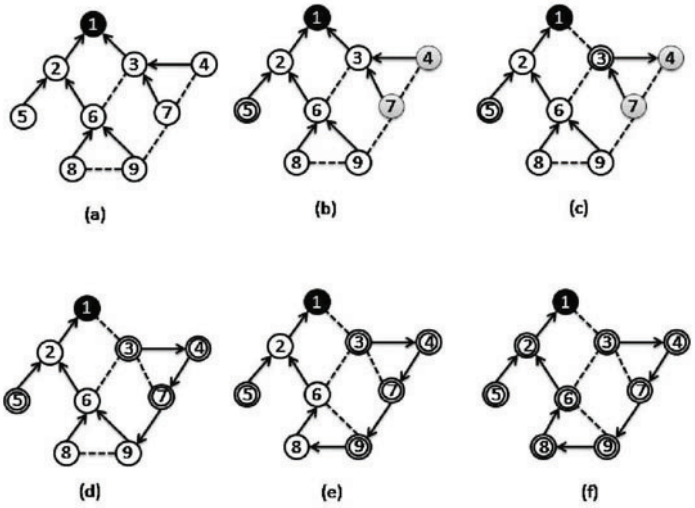

Figure 7. Sample Run of the Switching Algorithm

from its p_{new} when the *TOKEN* traverses through the forward paths from the root node to the leaves and from each u where $u \in C_{new}$ when the *TOKEN* traverses through the backward paths from the leaves to the root node.

Lemma 4.3.2 *The COUNTER1 value at each node eventually becomes 0 and the node sends back TOKEN to its* p_{new}.

Proof Each leaf node sends back *TOKEN* to its p_{new} independently as their COUNTER1 value is 0 always. For any intermediate node u, COUNTER1 value is decremented when u receives *TOKEN* back from each v where $v \in C_{new}$. So the COUNTER1 becomes 0 when the *TOKEN* is received from all v where $v \in C_{new}$ and as a consequence it sends back *TOKEN* to its p_{new}.

Lemma 4.3.3 *When an intermediate node becomes a leaf node for the switched tree, then eventually its* C_{curr} *becomes EMPTY.*

Proof When an intermediate node u becomes a leaf of the switched tree, then all v where $v \in C_{curr}$ send *TOKEN* to their respective p_{new} by Lemma 4.3.2. Also each v sends *CANCEL* to its p_{curr} u. On receiving *CANCEL* from a node v where $v \in C_{curr}$, node u removes v from C_{curr}. Thus C_{curr} becomes empty eventually.

Theorem 4.3.4 *The switching algorithm will be eventually terminated.*

Proof Depending on the network load, the root node takes the decision of switching and starts the switching procedure by sending *TOKEN* to each node $v \in C_{new}$. Each node u on receiving *TOKEN* from its p_{new}, forwards it to all $v \in C_{new}$. The variable $Counter1$ at u tracks the cardinality of the set C_{new}. This phase of the algorithm terminates if $C_{new} = \phi$ for the node u; or in other words u is a leaf node. At the second phase each leaf node u sends back the *TOKEN* message to a node v where $v = p_{new}$. Thus the link between u and v becomes permanent according to Lemma 4.3.2. Let $Ncount(l)$ be the nodes at level l of the tree. Further, let all the links between $Ncount(l+1)$ and

$Ncount(l)$ be already permanent. Then according to Lemma 4.3.2, the COUNTER1 value at each u where $u \in Ncount(l)$ becomes 0 eventually. They sent back the *TOKEN* to v where $v \in Ncount(l-1)$ and $v = p_{new}(u)$. Then all the links between $Ncount(l)$ and $Ncount(l-1)$ become permanent. Thus a node u at level l does not transmit any more messages to a node at level $(l+1)$, or at level $(l-1)$ once all the edges incident on u have become the part of the new tree. This phase of the algorithm terminates producing a switched tree correctly when the root at level 0 receives back *TOKEN* from all its children. The variable $Counter1$ at the root becomes 0 as a consequence. Thus there exists no more control messages in the system and all the nodes and edges become stable at this point. Hence the algorithm terminates successfully.

Theorem 4.3.5 *There will always be a path from any node to the root node at the time of switching.*

Proof When a node u needs to change its parent due to switching, it has to be assured that there exists a path from $p_{new}(u)$ to the root. Now a node u sends *TOKEN* to a node v where $v = p_{new}(u)$ to make a permanent link only when it has already received *TOKEN* from v through forward *TOKEN* traversal. That means node v had a path to the root. v could change its path to the root only after receiving *TOKEN* from all $w \in C_{new}(v)$. Node u sending *TOKEN* back to its p_{new}, v implies that the path from v to the root has not been changed yet. This implies that there exists a path from any node to the root always, irrespective of the switching.

Theorem 4.3.6 *No cycle will be created in the system due to switching.*

Proof All the leaf nodes u, whether it is newly created or already existing one, independently send back *TOKEN* to its p_{new} to build up a permanent link. According to Theorem 4.3.5 that node v, where $v = p_{new}$ is a part of the new tree and hence has a path to the root. A cycle can only form if $v \in C_{curr}(u)$ or there was a path $P = \{v, v+1, ..., u-1, u\}$ from v to node u according to old tree construct. From Lemma 4.3.3, $C_{curr}(u)$ will become empty eventually breaking the cycle. Now suppose node v made a link to a node x, where $x \in P$ and also $x = p_{new}(v)$ making a cycle $C = \{v, v+1, ..., x, ..., u-1, u, v\}$. But there must be at least one node y where $y = x$ or x is a descendant of y such that there exists a path from node y to the root. This is because the *TOKEN* has already traversed from the root to the node v through node y along the forward path. So node y will eventually make a permanent link to its parent along the path to the root and thus breaking the cycle C. Thus no cycle will be formed due to the switching.

Theorem 4.3.7 *Each convergecast message will be eventually delivered to the root node.*

Proof When an intermediate node u becomes a leaf of the switched tree, first it assures that all v where $v \in C_{curr}$ is blocked by sending *CANCEL* to them and changes its parent after receiving *ACKC* as response. So all the data received from $v \in C_{curr}$ are forwarded to its p_{curr} using the old link and no more data is received after receiving *ACKC* from all $v \in C_{curr}$. Thus no data is lost in this case.

When a node u needs to change its parent due to switching, first it informs the node v where $v = p_{curr}(u)$ by sending *CANCEL* message. No more data is sent to v after that. All the future data are forwarded using the link to the node w where $w = p_{new}(u)$. By Theorem 4.3.5 there exists a path from w to the root. So no data will be lost during

the switching process.
From the above discussion and Theorem 4.3.5 and Theorem 4.3.6 it can be proved that each convergecast message will be delivered to the root eventually.

Theorem 4.3.8 *Each convergecast message will be delivered exactly once.*

Proof Data redundancy occurs when parent-child relationship for any two nodes gets reversed due to the switching. When a node $v \in C_{curr}(u)$ becomes p_{new} of a node u, it is guaranteed that no data will be received after node u has sent *CANCEL* message to $v \in C_{curr}$. All the data received before will be forwarded through the old link. So no data sent from $v \in C_{curr}$ to u will be forwarded back from u to v where $v = p_{new}(u)$. Therefore no duplicate data will be generated during the switching process. Also from the Theorem 4.3.7 each message will be eventually delivered to the root. So it is implied that each convergecast message will be delivered exactly once.

Theorem 4.3.9 *There will be no indefinite stall in data gathering process.*

Proof When an intermediate node u becomes a leaf of the switched tree, there may arise a temporary stall in the data gathering process until $C_{curr}(u)$ becomes empty. From Lemma 4.3.3 it is observed that C_{curr} of such an intermediate node becomes NULL eventually. So there will be no indefinite blocking in data gathering process.

Theorem 4.3.10 *The message complexity of the switching algorithm is* $O(|E|)$.

Proof Let n and E be the total number of nodes and edges in the network and d_v be the degree of node v. So v sends d_v number of *TOKEN* messages and receives the same in reverse direction. Hence the total number of *TOKEN* messages is, $M_T = 2 \times \sum_{v \in V} d_v = 4|E|$. In the worst case scenario there will be $(n-2)$ number of nodes whose parents can be changed and there will be always less than n number of nodes which have been converted to leaf nodes from intermediate nodes due to the switching process. So there will be $O(n)$ numbers of *CANCEL* messages in worst case. Similarly there will be $O(n)$ number of *ACKC* messages. Hence the worst case message complexity for the switching between a DFS and a BFS tree = $4|E| + O(n) + O(n) = O(|E|)$.

Theorem 4.3.11 *The algorithm runs in* $O(D)$ *steps, where* D *denotes the diameter of the network.*

Proof Let D be the diameter of the network. At the first phase, *TOKEN* is forwarded to all the nodes across the levels of the tree, starting from the root node towards the leaves. Then the *TOKEN* is traversed back from the leaves to the root at the second phase. When the root receives back the *TOKEN*, the algorithm terminates. So, as a whole it takes at most $2 \times D$ steps to complete the switching process, where D ranges from $\log_2 n$ to n and n is the total number of nodes in the network.

Theorem 4.3.12 *The per node space complexity of the algorithm is* $O(\delta)$ *where* δ *is the maximum degree among all the nodes in the network.*

Proof Each node keeps the information about the set of current children as well as the set of children for the switched tree and the current parent pointer. The total number of neighbors for a node can be represented by its degree δ. As we have considered the static nature of the network, any node will need to keep the information for $2 \times \delta$ number of nodes in the worst case. So the per node space complexity is $O(\delta)$.

5. Discussion

Our goal was to investigate a correct convergecast in spite of dynamic switching between the two precomputed trees. In a dynamic scenario, one or more nodes may fail, be added to the network and/or removed from the network. Let us assume that one node crashes. In this case, both the BFS and DFS tree become invalid. Also there will be loss of convergecast messages. Also crash of an intermediate node may stall the convergecast. One approach to the problem may be to recompute the BFS and DFS tree to take care of the failure. However this may be expensive. Also this will cause a global freeze of the convergecast process in the entire network. Another approach may be to locally repair the failures (if possible) and allow the convergecast to take place even during failure recovery. Our goal is to minimize the loss of messages.

The switching has two components. They are *when to switch*, and *how to switch*. In this work we have assumed that there is an oracle at the root node to decide when to initiate the switching. The idea is that the root will monitor the load of the network. This can be done by piggybacking the load information of each node on the convergecast message. Finally, the root can calculate the average load on a per-node basic. If the average load is higher than a threshold then it may initiate a switching to the other tree which is better for higher load. In this work we have concentrated on the direction of how to switch in an efficient way to establish correct convergecast in spite of switching.

6. Conclusion

In this paper, we have proposed a distributed tree switching algorithm for load adaptive convergecast. The algorithm guarantees that even if a switching occurs between the underlying topologies, the application layer convergecast remains unaffected. We have shown that each convergecast packet is eventually delivered to the sink without any data loss or redundancy even during the switching. The switching algorithm eventually terminates. The traditional approach for switching between two protocols is a method similar to the two-phase-commit(2PC) protocol. In two phase commit based switching, a coordinator first broadcasts a “prepare” message, and all the other processes pause their work and send back acknowledgments. Each process is buffering messages from its own application at this point. After receiving back all the acknowledgments, the coordinator broadcasts a “switch” message. All the processes resume working using the new configuration upon receiving that “switch” message. In this approach, all the nodes get involved in switching at the same time blocking the whole system and therefore increasing the delay. Application can not proceed during the switching. It is not scalable also. In the proposed algorithm, the switching starts at all the leaf nodes of the tree and the switching wave moves upward. It is already proved in Theorem 4.3.9 that the temporary blocking of the application data is confined only to the local neighborhood of a new leaf node which was an intermediate node before. So there is no global freeze and the system is available even during the switching. The algorithm runs within $O(D)$ steps, where D is the diameter of the network and the message complexity is $O(|E|)$. Thus in comparison to the traditional method the proposed algorithm is more scalable and has small switching overhead and is expected to perform better in real systems. It will be interesting to investigate the problem when the trees used for the switching are not pre-computed.

References

Annamalai, V., Gupta, S., and Schwiebert, L. (2003). On tree-based convergecasting in wireless sensor networks. In *Proceedings of IEEE Wireless Communication and Networking Conference*, pages 1942–1947.

Chen, H., Mineno, H., and Mizuno, T. (2008). Adaptive data aggregation scheme in clustered wireless sensor networks. *Computer Communications*, 31:3579–3585.

Heinzelman, W. R., Kulik, J., and Balakrishnan, H. (1999). Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 174–185, New York, USA. ACM.

Karmakar, S. and Gupta, A. (2007). Adaptive broadcast by distributed protocol switching. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 588–589.

Krishnamachari, B., Estrin, D., and Wicker, S. B. (2002). The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 575–578, Washington, DC, USA. IEEE Computer Society.

Liu, X. and van Renesse, R. (2000). Fast protocol transition in a distributed environment (brief announcement). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 341, New York, NY, USA. ACM.

Liu, X., van Renesse, R., Bickford, M., Kreitz, C., and Constable, R. (2001). Protocol switching: Exploiting meta-properties. In *Proceedings 21st International Conference on Distributed Computing Systems Workshops*, pages 37–42, Mesa, AZ , USA.

Mocito, J. and Rodrigues, L. (2006). Run-time switching between total order algorithms. In *Euro-Par International 2006 Parallel Processing, Lecture Notes in Computer Science*, volume 4128/2006, pages 582–591, Dresden, Germany. Springer Berlin / Heidelberg.

Upadhayayula, S., Annamalai, V., and Gupta, S. (2003). A low-latency and energy-efficient algorithm for convergecast in wireless sensor networks. In *Proceedings of IEEE Wireless Communication and Networking Conference*, pages 1942–1947.

Yacoab, M. M. and Sundaram, V. (2010). An adaptive traffic aware data aggregation technique for wireless sensor networks. *American Journal of Scientific Research*, pages 64–77.