**The Report Committee for Juan Manuel Casas Jr.**
**Certifies that this is the approved version of the following report:**

# Distributed Trigger Counting Algorithms

**APPROVED BY**
**SUPERVISING COMMITTEE:**

**Supervisor:**

Vijay K. Garg

Bruce McCann

# Distributed Trigger Counting Algorithms

**by**

**Juan Manuel Casas Jr., B.S.**

**Report**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

**The University of Texas at Austin**

**December 2010**

# Dedication

To my brothers and sisters

# Acknowledgements

I would like to kindly thank my Graduate Studies Committee (Dr. Vijay K. Garg and Dr. Bruce McCann) for providing guidance and for allocating their valuable time to this endeavor. I wholeheartedly thank my family and friends for their encouragement and support. I am grateful to my employer IBM and CLEE for providing the opportunity to accomplish my academic aspirations.

December 2010

# Abstract

## Distributed Trigger Counting Algorithms

by

Juan Manuel Casas Jr., MSE

The University of Texas at Austin, 2010

Supervisor: Vijay Garg

A distributed system consists of a set of N processor nodes and a finite set of communication channels. It is frequently described as a directed graph in which each vertex represents a processor node and the edges represent the communication channels. A *global snapshot* of a distributed system consists of the local states of all the processor nodes and all of the in-transit messages of a distributed computation. This is meaningful as it corresponds to the global state where all the local states and communication channels of all the processor nodes in the system are recorded simultaneously. A classic example where snapshots are utilized is in the scenario of some failure where the system can restart from the last global snapshot. This is an important application of global snapshot algorithms as it forms the basis for fault-tolerance in distributed programs and aids in serviceability as a distributed program debugging mechanism. Another important application includes checkpointing and monitoring systems where a set of continuous

global snapshots are employed to detect when a certain number of triggers have been received by the system.

When the distributed system is scaled in terms of an increase in the number of processor nodes and an increase in the number of expected triggers the message complexity increases and impacts the total overhead for the communication and computation of the global snapshot algorithm. In such a large distributed system, an optimal algorithm is vital so that the distributed application program that is employing the snapshots does not suffer from performance degradation as the size of the distributed system continues to grow over time. We are interested in global snapshot algorithms that offer lower bound message complexity and lower bound MaxLoad messages for large values of N processor nodes and large values of W expected triggers. In this report we study and simulate the Centralized, Grid based, Tree Based, and LayeredRand global snapshot algorithms then evaluate the algorithms for total number of messages (sent and received) and MaxLoad messages (sent and received) for the trigger counting problem in distributed computing. The report concludes with simulation results that compare the performance of the algorithms with respect to the total number of messages and MaxLoad messages required by each algorithm to detect when the number of W triggers have been delivered to the distributed system.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1: Introduction

Distributed systems have become increasingly important in science and medicine by facilitating the necessary computing power for problem solving distributed applications. A complex mathematical computation is executed in the form of a sophisticated distributed algorithm that can leverage the computing power of all the processor nodes in the distributed system. The distributed algorithm is designed to perform parallel computations and arrive to a solution at a faster rate. Current projects that utilize distributed systems include SETI@home (Search for Extra Terrestrial Intelligence) and Folding@home (Protein Folding and Molecular Dynamics) as well as scientific applications for climate prediction and earth quake detection. The distributed application is initiated as a moderate size distributed system and over time it may grow to become a large complex set of interconnected processor nodes. It is essential to maintain a lower bound number of messages required by the underlying snapshot computation to effectively scale with the size of the distributed system. Therefore it is important to develop efficient global snapshot algorithms that deliver optimal performance as the number of processor nodes in the distributed systems increases.

The first published work describing a global snapshot algorithm was introduced by Chandy and Lamport in 1985 [CL85] as a solution to the fundamental problem in distributed computing. Their work described a protocol for capturing the state of a distributed system by using control messages and various processor node and communication channel states to record a global state of the system. The approach is such that a processor node can record its own state as well as the messages it sends and receives, but nothing else. Naturally a processor node must rely on information recorded by other processor nodes about their own state when performing a computation to

determine a global system state. One obvious problem is that the processor nodes don't record their local states at exactly the same time since they don't share a common system clock. The challenge then is to develop algorithms that are designed to capture all the processor node and channel states in order to determine the systems global state.

In their algorithm, each processor node records its own state and the two processor nodes that share a communication channel cooperate in recording the state of the channel. The requirement is that the two processor nodes coordinate to form a valid and meaningful state for achieving the global snapshot. The global state recording algorithm is superimposed onto the distributed application to be executed concurrently while not interfering with the underlying computation. The global snapshot algorithm will send various control messages to the processor nodes and the nodes are required to handle these messages and perform the necessary operations to record states for the snapshot computation. Furthermore, the processor node must ensure that the underlying computation does not stall as a result of processing the control messages. Thus, the logic in the algorithm allows for each processor node to maintain a set of invariants that assist the processor nodes in coordinating the snapshot for their local state as well as the state of the communication channels.

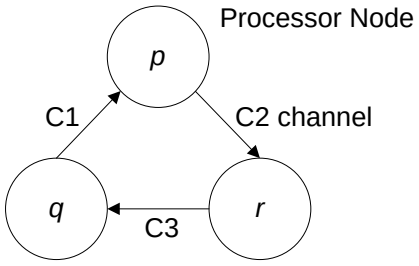


Figure 1: A distributed system consisting of three processor nodes *p*, *q*, and *r* and channels c1, c2, and c3.

To illustrate an example of a distributed system and the global state for a snapshot, consider the simple system consisting of only two processor nodes *p* and *q*, and two channels c and c'. The possible events that may occur at a processor node include: internal events, message send events, and message receive events. In this example we assume that the channel state can be recorded without any delay. This allows us to gain an intuitive understanding of the sequence of events and their relationship as it pertains to the instant when the state for channel c and the node states for *p* and *q* are recorded. Initially the global state of the distributed system begins with each processor node and channel having an initial state and empty sequence. An event for this system is defined as a single token arriving at a processor node. When an event occurs, the state of the processor node and channel change and this corresponds to transitions in the global system state. Specifically, an internal event changes the state of the processor node at which the internal event occurred. Furthermore a send or receive event changes the state of the processor node as well as the state of the channel on which the message was sent or received.

Consider the case when the token arrives at node *p* and the recorded local state at *p* indicates that the token is in *p*. The snapshot algorithm will record this as a valid global state with token in *p* and no state transitions occurring at processor node *q* or channel c. Next, consider the subsequent event where *p* sends the token to *q* via communication channel c. In this scenario, the state of *p* and c change to reflect the send message event, but while the message is in transit there will be no state change at node *q*. Hence the global snapshot algorithm must correctly recognize that a message is in flight from processor node *p* to *q* via channel c. This is an important property of the snapshot algorithm as recording only the local states of the processor nodes while the token is in flight will incorrectly compute that no token exists in the system. Therefore, when

computing the global system state, the snapshot algorithm must account for all nodes and channel states and coordinate the transition sequences to form a consistent global state.

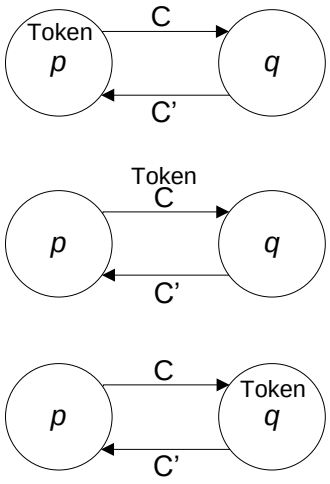


Figure 2: A two processor node distributed system showing token transition from node *p* to node *q* over channel c.

# Chapter 2: Related Work

The acceptance and popularity grew for the seminal paper which Chandy and Lamport introduced for computing a meaningful global snapshot. The published work quickly piqued the interest of researchers as they set out to explore the algorithm for possible improvements. Soon after, variations of Chandy and Lamport's algorithm followed with the first being introduced by Spezialetti and Kearns [SK89] which describes an improvement in efficiency where their algorithm optimizes concurrent initiation of snapshot collection as well as improved efficiency for distributing the recorded snapshot. Many such improved variations followed including: incremental snapshots [Ven93], wave synchronization [JMH89], and vector clock [Mat93] based algorithms.

These early variations of Chandy and Lamport's baseline work focused on improving the algorithm to reduce overall size and complexity and these were important steps towards refinement. In more recent work the focus has turned to measuring the performance of the global snapshot algorithms when scaling the system to a large number of processor nodes and a large number of expected triggers. Garg *et al* proposed scalable algorithms and studied their performance [GGS06] of the global snapshot by deploying the distributed system on the IBM BlueGene/L massive parallel supercomputer which could scale up to 65,536 processor nodes. Their implementation is developed using the ANSI-C language and the set of experiments utilize a distributed application that is scaled to 32, 64, 128, 256, and 512 processor nodes. Each node in the distributed system sends and receives a specified number of application messages and 40,000 W trigger messages to random processor nodes in the system. By using the MPI library for inter-process communication the send and receive events are blocking calls in the sense that

the processor node logic allows for control messages to be processed even when there are application messages available. This ensures that the global snapshot algorithm will successfully terminate when the total number of expected triggers are delivered to the system. They show that their proposed global snapshot algorithms scale much better than existing algorithms and generalize the problem to be solved as the *distributed trigger counting* (DTC) problem. Additionally, this work also describes the lower bound for the number of point-to-point control messages required by any algorithm to detect termination.

Garg *et al* further study the DTC problem in large distributed systems where N processor nodes receive some triggers from an external source [CSG10]. The LayeredRand DTC algorithm is described as a scalable global snapshot algorithm which terminates with a successful completion by raising an alert when a user specified number of W triggers is received by the system. The main result from their study is a decentralized and randomized algorithm with an expected message complexity $O(n \log n \log w)$ and a MaxLoad $O(\log n \log w)$ receive messages with high probability where all earlier algorithms have $\Omega(n \log w)$ MaxLoad.

The main contribution presented in this report is the first implementation of the LayeredRand algorithm and a performance evaluation comparing the scalability with a centralized algorithm and the Tree Based algorithm described in [GGS06]. The distributed system is modeled and simulated in Java as an abstraction of processor nodes in an asynchronous message-passing system that does not have a shared clock. The Java distributed application program spawns a set of N threads that represent the processor nodes in our distributed system. Each processor node creates a set of TCP socket connections that form the communication channels for sending and receiving application messages, control messages, and trigger messages. Each algorithm is measured for the

total number of sent and total number of received message involved in the DTC global snapshot algorithm and also evaluate the MaxLoad for sent and received messages.

# Chapter 3: DTC Algorithms Overview

The distributed system in our simulation is modeled to reflect the scalable experimental model given in [GGS06] where the Grid Based and Tree Based algorithms are detailed. To establish a proper foundation for our experiments, we begin by describing a simple solution to the *distributed trigger counting* problem in the form of a Centralized algorithm. This is followed with an overview of the scalable DTC algorithms that are studied for our simulation.

### CENTRALIZED ALGORITHM

A centralized approach offers the simplest algorithm for determining when the distributed system has received all W trigger messages with $O(W)$ message complexity. The processor nodes are organized in a star topology or the classic client/server model where one node in the system will act as the root node server while all other processor nodes act as end clients. The algorithm begins with all nodes being initialized and ready to receive trigger or application messages. The root server node maintains a count of the total expected triggers. When this count reaches zero then the root server node knows that the system has received all W triggers and will raise an alert. The only messages involved are those which are sent by the end clients to inform the root node that it has received a trigger message. The root node will process this message and decrement the expected total trigger count. If the root node itself receives a trigger message then it need not send a message to any node and instead simply decrements its total trigger count to account for the trigger message it received. The centralized algorithm is effective for a distributed system with a small number of nodes, but its performance degrades as the total number of nodes grows to a large scale and the number of expected triggers increases.

## GRID BASED ALGORITHM

The grid-based algorithm logically organizes the processor nodes to form a grid-like structure of the distributed system. This algorithm requires $O(\sqrt{N})$ messages per processor node with each message requiring a size of $O(\sqrt{N})$ integers. It is similar to previous algorithms in that it also exhibits O($N$) space complexity, but differs from other algorithms with respect to the size of each message. Although each message is larger than existing algorithms, the grid-based algorithm improves on the total overhead of communication for the snapshot computation by reducing the total number of messages required for the global snapshot algorithm to determine that all W triggers have been received by the system. The key to the grid-based algorithm is twofold; first each node only needs to keep track of the number of triggers that have been sent to it and second a node can use a single message to communicate the number of messages it has sent to other processor nodes.

A node state is designated by a color white, red, or black that corresponds to the local state of the processor node, but not to the state of its channels. This color scheme invariant notation is used for all the algorithms and denotes three states that a processor node can have. If the node is color white then it has not recorded its local state and is able to process application messages or trigger messages if no control messages are available. If the node is color red then it has recorded its local state but not the state of incoming channels. The node is color black if it has recorded both its local state and all the in transit messages for its channels. The algorithm is initiated with all nodes in the grid being color white and then the system begins processing any application and trigger messages that arrive. An event is detected when a trigger message arrives at a node at which point the node may initiate some control messages as dictated by the DTC algorithm logic inherent in each processor node. Three main components are described

in the DTC algorithm to compute the number of triggers that the system has received. The first component is a predefined spanning tree in the distributed system that is used by any processor node to broadcast a message that it has initiated the global snapshot. This broadcast message is received by all other nodes in the system and is considered an event that should turn the processor node color red. The first component is standard for DTC algorithms [GGS06] and is included for completeness.

```
initiate() // enabled only if (color=white)
        take local checkpoint;
        color = red;
        sent "init" to all processes connected by spanning tree edges

On receiving "init" message or a control message on channel c
        if (color = white)
                take local checkpoint;
                color = red;
                send "init" to all spanning tree channels except c
```

Figure 3: Common Initiation Component for DTC algorithms

The second component is for each process to determine the total number of triggers that have been received by the system. To do so the grid-based algorithm requires that each processor node maintain information (vector of integers) about messages it has sent to other processor nodes and uses this information to compute the total number of trigger received by the system. A marker is sent on every communication channel belonging to the node and when this marker is received by another processor node, it indicates to the receiving node that it also must turn red if it is not already red. Thus to record the state of the channel, a node begins recording all the messages it receives on its channel after turning color red. This component concludes with every processor node knowing the total number of triggers that have been received.

The third and final component of the algorithm is meant to turn all processor nodes color black as well as detecting when this has occurred. Recall that a black color node indicates that the node has recorded its local state and all channel states. The algorithm utilizes the spanning tree to perform a converge cast to determine when all nodes in the tree have turned black.

### TREE BASED ALGORITHM

The tree based algorithm has a similar initiation for the first component and reduces the vector integer information that is required by the grid snapshot algorithm. Recall that in the grid-based algorithm, a vector of integers is maintained at each processor node and by reducing the number of integers maintained by each node, the tree based approach improves the snapshot algorithm as it reduces the overhead to $O(1)$ integers instead of $O(N)$. The tree based algorithm organizes processor nodes in a binary tree with communication channels existing from parent to direct children. Unlike the grid-based algorithm which requires a node to maintain the number of individual messages sent to other nodes, the tree based algorithm requires each processor node to maintain a deficit which denotes the total number of triggers expected by the node minus the number triggers received. The deficit is maintained in the form of tokens such that when a trigger message arrives at a processor node a deficit token is consumed.

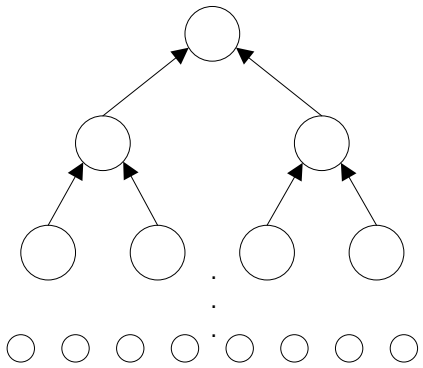

Figure 4: Tree-based algorithm processor node topology and communication channels from children to parent.

The second component of the tree based algorithm is meant to compute the total deficit in the system. This can be determined using the converge cast on the spanning tree in $O(N)$ messages. A processor node located on the lowest leaf layer of the binary tree will consume all its deficit tokens before it initiates a search for tokens. Processor nodes in non-leaf layers of the tree will also initiate this search and additionally may initiate operations to obtain tokens in order to maintain properties that the tree based algorithm requires. This is the third component of the tree based algorithm which is based on computing when the sum of all deficits is zero as it indicates that all W triggers have been received by the system.

The tree based algorithm also includes colors to denote the states of the channels in the system. A processor node channel is green if it has strictly greater than $W_{max}$ deficit tokens where $W_{max}$ is the largest number of deficit tokens that any processor node can have and is recomputed for each round performed by the snapshot algorithm when determining if all triggers have been received. A processor node channel is yellow if it has greater than zero and less then $W_{max}$ tokens. A processor node channel is orange if it has no deficit tokens and has received one or more trigger messages and such a channel state causes the node to initiate a search for tokens.

The token search is initiated and performed when the node transitions from color green to yellow invoking the action where it sends a swap message to its children in an effort to maintain the first invariant property, namely that a yellow node cannot have a green child. The search can also occur when a node transitions from yellow to orange causing it to send a split message requesting half of its parent's tokens in an effort to maintain the second invariant property that an orange node must turn yellow. The third invariant property is that the root node is green and thus if the root node turns yellow and stays yellow then by the previous two properties the entire tree must be yellow. Once the entire tree is yellow then the root node can initiate the next round of the algorithm by first computing the total deficit tokens in the system. The tree based algorithm performs a converge cast to determine the total number of triggers received and accounted for by the processor nodes in the system and this is used to derive a new deficit token count. The new deficit token count is then distributed to the processor nodes for the subsequent round in the algorithm. The main logic to search for tokens is given in the figure below for completeness.

```
On turning from green to yellow
        if any child is green
                send ("swap", tokens) to that child;
        else if root node
                reset for the next round;

On turning from yellow to orange
        send ("split", tokens) to the nearest green ancestor;
```

Figure 5: The Tree-based Algorithm for DTC

### LAYEREDRAND ALGORITHM

The LayeredRand algorithm is introduced in [CSG10] and is described as a deterministic algorithm with message complexity $O(n \log n \log w)$ with high probability and a MaxLoad $O(\log n \log w)$. This algorithm uses a tree topology for the logical organization of the processor nodes and makes use of the tree layers with the root node acting as a master node at layer zero. This algorithm is similar to the tree-based algorithm in that it works in multiple rounds. Unlike the tree-based algorithm where communication channels are limited to a parent and its direct children, the LayeredRand algorithm provides each processor node at layer L of the tree with a communication channel to every processor node at layer L – 1 of the tree.

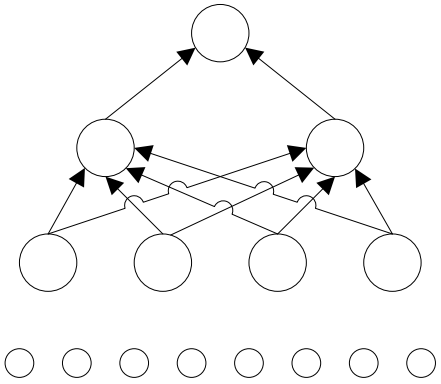

Figure 6: LayeredRand algorithm processor node topology and communication channels from each child in layer L to each parent in layer L - 1.

The algorithm initializes each processor node for the first round with a trigger deficit *threshold* value which is a function of the current rounds' deficit trigger value and the tree layer in which the node resides. This layer threshold value corresponds to the number of deficit triggers yet to be received in the round by any processor node in that layer. Each node maintains a counter C to keep track of the number of triggers it has received as well as some triggers that have been received by the tree layer below it. Each

time a node receives a trigger message it will increment C by one and when C exceeds the current round threshold then the processor node will sends a message to a random node in the tree layer above it. This message serves to inform the fact that the sending node has received more triggers than the computed threshold for its layer. The node that receives the inform message will increment its own C by an amount equal to that of the sending nodes layer threshold value. The root node initiates an end of round procedure when its own C counter value exceeds half of its layer threshold value.

Before the next round of the algorithm is started, the total number of triggers received by the system is computed using the converge cast and then subtracted from the total number of expected W triggers. This evaluates to the total number of trigger messages that have yet to be received by the system and if this value is zero then the algorithm raises an alert to indicate that all triggers have been received. If there are remaining triggers yet to be received then the newly computed deficit value is used to compute each node's layer threshold value for the next round of the algorithm.

# Chapter 4: Simulating a Distributed System

The Java language is chosen for the implementation language as it offers the benefit of platform independence. The application code is developed using the Eclipse SDK version 3.5.2. The simulation is developed under a java project named DTC that contains application Classes in the default *src* directory with one *xmpt* source subdirectory containing Classes for the inter-process communication that make up the channels in the distributed system. The application Classes include a Driver Class, an ExternalTrigger Class, and algorithm specific CentralizedNode, TreeBasedNode, and LayeredRandNode Classes that are developed to perform the simulation experiments. The application is packaged as a Java executable jar file that can be run on any platform that supports the Java Runtime Environment. An overview of the application and communication Classes is given here and further details including source code is provided in the report appendix.

### DRIVER CLASS

A Driver Class is developed to be the main running program in our distributed application which accepts user command line parameter inputs and initiates the DTC algorithm exercisers. The Driver is responsible for spawning the various threads that make up the distributed system processor nodes and will also complete any necessary initialization before entering the loop that invokes ProcessorNode.start(). The node initialization loop is performed by the Driver Class based on the user specified algorithm type, the number of layer in the spanning tree, the total number of triggers, and the hostname with port number. Each processor node is then initialized having the input hostname and port number where the port number is incremented by one for each subsequent node that is started. When the initialization loop has completed the Driver

will sleep for a small amount of time relative to the total number of nodes in the distributed system to allow for stabilization.

```
java -jar dtc_sim.jar <algorithm type> <hostname:port> <tree layers> <total triggers>
```

Figure 7: Command line specification for Distributed Ticket Counting simulation

The Driver then proceeds to begin the algorithm exerciser by spawning an external trigger thread that will function as the source of the external trigger messages that are sent to the distributed system. The Driver also functions as the source of application messages that are sent to the distributed system. The application messages can serve as a verification tool for ensuring that an application computation is not starved out by messages generated from the snapshot algorithm. For example, the Driver can send a "Report" message to the distributed system and each processor node will process this request and print out node and channel state statistics for the concurrently running global snapshot algorithm.

The snapshot algorithm that is being exercised will raise an alert once it has detected that all trigger messages have been received by the system. This initiates a post phase computation where the root node requests the final statistics of all other nodes in the distributed system. The root node then processes the responses and generates statistics related to the exercised snapshot algorithm. The root node will display the algorithms total number of send and receive messages that correspond to the total number of control message events that were required for the algorithm to determine that all tickets have been received by the system. Additionally, the root node will display statistics related to the number of MaxLoad send and receive messages for the algorithm after examining the statistics reported by each processor node.

```
Node# 0 DTCRcvd: 397 DTCSent: 397 MaxLoadRcvd: 397 MaxLoadSent: 397
```

Figure 8: Output of post phase computation after the snapshot algorithm exerciser has successfully detected that all triggers have been received.

```
public class Driver {
   BufferedReader din; // command line argument reader
   PrintStream pout;  // print to standard out
   Integer L;  // number of layers in tree
   Integer N;  // number of processor nodes
   Integer W;  // number of triggers
   RT routeTable = new RT(); // processor node hostname and port number

   public static void main(String args[])
   {     // algorithm type to exercise
         int alg_type = Integer.parseInt(args[0]);
         Driver D = new Driver();
         // Create input message stream from second entry in args[]
         StringTokenizer st = new StringTokenizer(args[1], ":");
         // error if st.coutTokens() != 2
         D.routeTable.hostname = st.nextToken();
         D.routeTable.port = st.nextToken();
         D.L = Integer.parseInt(args[2]); // # of layers or rows/columns
         D.W = Integer.parseInt(args[3]); // # of triggers or tokens

         if (alg_type == 1) // #r of nodes for LayeredRand
                  D.N  = (int)Math.pow(2, D.L) - 1;
         else if (alg_type == 2) // # of nodes for GridBased
                  D.N = D.L * D.L;
         else if (alg_type == 3) // # of nodes for TreeBased
                  D.N  = (int)Math.pow(2, D.L) - 1;
         else if (alg_type == 4) // # of  nodes for Centralized
                  D.N  = (int)Math.pow(2, D.L) - 1;
         else
                  System.out.println("Error: Invalid Parameter.");
   }
}
```

Figure 9: Main Driver class for the Distributed Ticket Counting simulator.

### EXTERNAL TRIGGER CLASS

The External Trigger Class extends the Java Thread Class and overwrites the *run*() method with its own where it performs the various actions associated with sending a trigger to the distributed system. An object created from the *ExternalTrigger* Class is invoked with a set of parameters that specify the total number of triggers, the total number of nodes in the system, and the root node's port number. The java.util.Random Class is used to provide an easy way to generate uniformly distributed random numbers of integer type. The main loop iterates from zero to W - 1 and each time a random target is selected by adding a random generated number to the root node's base port number which forms the target port number. A trigger message is constructed to contain the trigger ID thereby making each trigger message unique. The external trigger thread sets the appropriate socket connection using the hostname and the randomly generated target port then sends the trigger message to the target processor node. This procedure repeats until all W triggers have been sent by the external trigger thread.

The External Trigger Class is modified to include logic for sending a random number of messages (within a given range) every second. This is an alternative than allowing the trigger thread to send triggers as quickly as possible and offers some additional opportunities for observations about the algorithms being evaluated. These observations are later explored when the experimental results are discussed.

```
public void run()
{
   try
   {
         for (int cnt = 0; cnt < this.W; cnt++)
         {
            int randomTarget = generator.nextInt(N) + rootPort;
            outS.setConnect(hostname, Integer.toString(randomTarget));
            TriggerMessage.setMsg("<XMsg color='" + "white" +
            "' type='" + "trigger" + "'>" + "<triggerID>" + cnt +
            "</triggerID>" + "</XMsg>");

            outS.sendXMsg(TriggerMessage);
            outS.closeSocket();
         }
   } catch (Exception e) {
         e.printStackTrace();
}
```

Figure 10: The External Trigger main loop is responsible for sending trigger messages to the distributed system.

### PROCESSOR NODE CLASS

The Processor Node Class extends the Thread Class and is the main class used for the various processor nodes in the distributed system. The Driver Class will create objects of Process Node type depending on the algorithm that is being exercised. When the Centralized algorithm is exercised CentralizedProcessor node objects are created from the base class CentralizedNode. Similarly, TreeBasedProcessor node objects are created from the TreeBasedNode base class for the Tree-Based algorithm and LayeredRandProcessor node objects are created from the LayeredRandNode base class for the LayeredRand algorithm. To communicate with other nodes in the distributed system, a processor node creates an XMsgInput stream and one or more XMsgOutput streams to send and receive messages of XMsg object type. Each processor node creates XMsg queues where it stores the messages that have been received, but not yet processed.

These message queues are important as they allow the processor node to service snapshot control messages while not blocking the channel from receiving and queuing non-control messages that are later handled when all control messages have been processed.

```
public void run() {
        while (true) {
                try {
                        Inmessage = inS.receiveXMsg();
                        if (canHandle)
                        {       handleMsg(Inmessage);}
                        else {  msgQueue.add();}
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Figure 11: Processor Node Class main loop for receiving, handling, and queuing messages.

The main loop in the *run*() method for a processor node is similar for each algorithm where the node will receive the message and determines if it can be handled right away or if it needs to be queued. The snapshot algorithm logic is implemented in the processor node methods and while some methods are common to all processor node types each Node Class will have some algorithm specific methods. For example, a processor node for any given algorithm will use a spanning tree where two common methods exist that are associated with the converge cast operation. The common methods are the *sendtoChildren*() and *sendtoParent*() methods. Additional details about processor node classes and methods including source code are provided in the appendix.

## INTER-PROCESS COMMUNICATION

The XML Markup Language is used to make up the main data structure that forms a message in our distributed system simulation. We call this message an XMsg and it contains content and context as an indication of the role the message plays in the simulation. The java.xml package is imported for the standardized XML markup language and XML parsers to easily identify structure in a message. The XMsg Class implements the *Serializable* interface and provides a *setMsg*() method which accepts a String or DOM Document as the input parameter. The XMsg class also provides a *serialize*() method which transforms the XMsg or DOM Document into a String and allows for easy visibility of a message when printing to console.

A processor node creates an InputXMsgStream to receive XMsgs and one or more OuputXMsgStreams to send XMsgs. The InputXMsgSteam() method takes as input a port number where the processor node will listen for incoming messages whereas the OutputXMsgStream() method takes as input the hostname and port number of the destination processor node. To maximize the use of multi-core CPU designs the InputXMsgStream and OutputXMsgStream extend the Thread Class so that each processor node can have an independent thread of execution for each communication channel in the distributed system. When we consider a set of communication threads that are competing for run time on a multi-core CPU then the time between each dispatch can be viewed as the latency experienced in a physical network of nodes.

As mentioned previously, the Processor Node Class contains logic for handling and queuing messages as they relate to the global snapshot algorithm or application computation. In contrast, the InputXMsgStream Class and OutputXMsgStream Class manage the low level TCP logic that is needed for the communication channels used by the processor nodes. Each XMsgInputStream channel object contains a synchronous

queue that holds the received messages whenever the node is busy processing other requests. The queued messages are then delivered when the processor node is ready to request the next message from the stream. This model improves performance of the processor node by reducing path length execution for the processor node thread because the communication logic and buffering is occurring on a separate thread of execution. Hence, the processor node need only concern itself with handling the messages that are delivered by the input stream thread.

This multi-threaded model is further leveraged to provide a processor node with a means to listen on multiple ports or to have multiple output streams and each with its own dedicated thread of execution. This can be viewed as a comparable design to that of a physical node having multiple network interfaces where the network adapter performs the various networking logic involved in accepting and delivering messages across the physical network.

```
inStream = new InputXMsgStream(port);
outStream = new OutputXMsgStream("localhost", rootNodePort);

XMsg msg = new XMsg("<XMsg> </ XMsg >");
Inmessage = inStream.receiveXMsg();
outputStream.sendXMsg(msg);
```

Figure12: Initialization of input stream, output stream, and XMsg.

# Chapter 5: Experimental Results

The experiment consists of several runs of the simulation to exercise each global snapshot algorithm ten times for each set of input data. The simulation is performed on an IBM P7 Power PC Server with a total of twenty 64-bit CPUs each having a processor clock speed of 3.30 GHz and a system total of 16 Gigabytes of physical memory. The results of each simulation run are recorded and then computed to derive the minimum, maximum, and average mean for the total number of sent and received messages as well as the MaxLoad for sent and received messages. The numbers of nodes that are used in the simulation are: 127, 255, 511, and 1023. Each simulation run is set to have an expected trigger count of 40,000 triggers that will be sent by the external trigger thread. In the following set of experiments the trigger thread does not experience any delay in sending the total number of triggers to random nodes in the system. A table containing simulation results for each metric is provided with a brief summary of the performance and scalability comparisons for the algorithms exercised in the simulation.

| N | Centralized | | | Tree Based | | | LayeredRand | | |
|---|---|---|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg | Min | Max | Avg |
| 127 | 39669 | 39723 | 39690 | 140 | 318 | 197 | 110 | 143 | 124 |
| 255 | 39813 | 39859 | 39834 | 233 | 529 | 354 | 120 | 153 | 141 |
| 511 | 39916 | 39925 | 39919 | 438 | 811 | 612 | 131 | 141 | 136 |
| 1023 | 39946 | 39967 | 39959 | 1294 | 1976 | 1633 | 142 | 162 | 151 |

Table 1: MaxLoad Received messages for the DTC algorithms.

The results in Table 1 report the minimum, maximum, and average number of MaxLoad received messages for each of the algorithms being evaluated in the simulation. The Centralized algorithm is the least favored algorithm for the MaxLoad received messages because the average number of messages increases linearly with the number of processor nodes. In contrast, the Tree Based algorithm reduces the MaxLoad received messages considerably for all values of N processor nodes in our simulation. The Tree Based algorithm exhibits good scaling up to 511 nodes with the average number of received messages showing a steady increase relative to the number of processor nodes. Additionally, as the number of nodes increase beyond 1000, we begin to observe that the average number of MaxLoad received messages increases at a faster rate. This can be attributed to the increase in message complexity when the number of processor nodes grows to produce many layers in the tree. The LayeredRand algorithm shows the most promising results when scaling beyond 1000 nodes as the average MaxLoad received messages are relatively consistent for all values of N processor nodes that were tested in the simulation. We also observe that there is a wide distribution gap between the minimum number of messages and maximum number of messages in the Tree Based algorithm whereas the LayeredRand algorithm maintains a relatively small difference between the minimum and maximum MaxLoad receive messages.

| N | Centralized | | | Tree Based | | | LayeredRand | | |
|---|---|---|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg | Min | Max | Avg |
| 127 | 350 | 365 | 358 | 80 | 196 | 120 | 66 | 85 | 75 |
| 255 | 185 | 200 | 191 | 82 | 200 | 123 | 70 | 95 | 85 |
| 511 | 102 | 119 | 109 | 72 | 101 | 87 | 76 | 89 | 81 |
| 1023 | 56 | 65 | 60 | 59 | 81 | 66 | 84 | 92 | 89 |

Table 2: The MaxLoad sent messages for each DTC algorithm.

The results in Table 2 report the minimum, maximum, and average number of MaxLoad sent messages for each of the algorithms being evaluated in the simulation. The Centralized algorithm shows an improvement in MaxLoad sent messages as the number of processor nodes increases. This is justified as the number of trigger messages are randomly sent to the distributed system and the processor nodes need only report to the root node that the trigger message has been received.

The Tree Based algorithm offers improved performance over the Centralized algorithm for distributed systems where N is greater than 255 processor nodes. Additionally, the Tree Based algorithm maintains relatively consistent minimum MaxLoad sent messages for all values of N processor nodes that are tested in the simulation. The LayeredRand algorithm offers the most consistent average MaxLoad sent messages for all values of N. The results show that the LayeredRand algorithm scales very well for MaxLoad sent messages as the value of N increases beyond 1000 processor nodes. Furthermore, the LayeredRand algorithm exhibits excellent minimum and maximum MaxLoad sent messages for all values of N processor nodes due to the fact that the processor node will send control messages only after a threshold of received triggers is exceeded.

| N | Centralized | | | Tree Based | | | LayeredRand | | |
|---|---|---|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg | Min | Max | Avg |
| 127 | 39669 | 39723 | 39690 | 4382 | 5415 | 4842 | 5081 | 6363 | 5780 |
| 255 | 39813 | 39859 | 39834 | 6652 | 10514 | 8816 | 10589 | 13289 | 12427 |
| 511 | 39916 | 39925 | 39919 | 12587 | 14250 | 13424 | 21519 | 15150 | 22917 |
| 1023 | 39946 | 39967 | 39959 | 18788 | 19359 | 19078 | 46189 | 47081 | 46592 |

Table 3: The total number of messages received for each DTC algorithm.

The results in Table 3 report the total number of received messages for each of the algorithms being evaluated in the simulation. The Centralized algorithm has the largest number of total messages received because the root node is responsible for processing a message for every trigger that is received by a processor node in the distributed system. The LayeredRand algorithm can more effectively determine when all trigger messages have been delivered to the system than the Centralized algorithm because there are fewer messages involved in communicating the fact that a trigger message has been received.

One characteristic of the LayeredRand algorithm is that it maintains the number of trigger messages received at each node and the sum is computed using the converge cast operation. This allows the algorithm to derive a new threshold for the next round of the algorithm and in doing so it causes an increase to the total number of received messages. Contrary to this approach, the Tree Based algorithm maintains the total number deficit tokens distributed evenly across all processor nodes. This offers an improvement over the LayeredRand approach since fewer rounds are required for the algorithm to determine when all trigger messages have been received by the system. Therefore, the Tree Based algorithm exhibits improved scaling over the LayeredRand

algorithm for total number of messages received in distributed systems consisting of greater than 511 processor nodes.

| N | Centralized | | | Tree Based | | | LayeredRand | | |
|---|---|---|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg | Min | Max | Avg |
| 127 | 39669 | 39723 | 39690 | 4382 | 5415 | 4842 | 3404 | 4384 | 3980 |
| 255 | 39813 | 39859 | 39834 | 39834 | 6652 | 10514 | 7271 | 9300 | 8692 |
| 511 | 39916 | 39925 | 39919 | 12587 | 14250 | 13424 | 14846 | 17388 | 15887 |
| 1023 | 39946 | 39967 | 39959 | 18788 | 19359 | 19078 | 31297 | 31938 | 31678 |

Table 4: The total number of messages sent for each DTC algorithm.

The results in Table 4 report the total number of received messages for each of the algorithms being evaluated in the simulation. The Centralized algorithm has the largest number of total messages sent because a processor node must send a message to inform the root node that a trigger message has been received. The total number of messages sent for the Tree Based algorithm and LayeredRand algorithm show similar performance comparison results as in Table 3. This reinforces our findings that the LayeredRand algorithm will require more rounds than the Tree Based algorithm in distributed systems consisting of larger than 511 processor nodes. As a result, the Tree Based algorithm exhibits the fewest number of total messages sent for all values of N processor nodes in our distributed system simulation.

# Chapter 6: Conclusion and Future Work

We conclude the distributed system simulation with our findings showing the LayeredRand algorithm providing the best performance for MaxLoad sent and received messages for the three DTC algorithms exercised in the experiments. We also conclude that the Tree Based algorithm offers the best results for fewest total number of sent and received messages for the three DTC algorithms exercised in the experiments. We find that the LayeredRand algorithm has the potential to outperform the Tree Based algorithm if the total number of rounds for the algorithm can be minimized to a deterministic lower bound.

The external trigger in our experiments was set to send trigger messages without any delay which often causes a processor node to queue a portion of trigger messages in favor of control messages generated by the DTC algorithm. For future work, the external trigger thread can be set to send a variable number of triggers each second and this will likely result in slightly different results due to the fact that the DTC algorithm may complete more or less computational rounds before all triggers are delivered to the system. Additionally, future work can include scaling the distributed system beyond 1023 processor nodes to observe the performance of the DTC algorithm when trigger messages are sent without delay compared to sending the trigger messages at a variable rate.

# Appendix (or Appendices)

What follows is the implementation source code for the java based distributed-application system simulator.

```
import java.util.*;
import xmpt.*;

public class LayeredRandNode extends Thread {
   Integer Counter = 0;
   Integer Dtriggers = 0;
   int N;
   OutputXMsgStream oParent;
   OutputXMsgStream oLeftChild;
   OutputXMsgStream oRightChild;
   private Queue<XMsg> default_Q = new LinkedList<XMsg>();
   private Queue<XMsg> priority_Q = new LinkedList<XMsg>();

   public LayeredRandNode(int rNodePT, int rNodeID, int me, int numserv,
                                                    RT rt, int W, int L, int lml) {
         super();
         this.N = numserv;
         this.myprocId = me;
         this.W = W;
         this.Layer = L;
         this.generator = new Random();
         this.inS = new InputXMsgStream(rt.port);
   }
}
```

Figure 13: LayeredRandNode Constructor.

```
 import java.util.*;
import xmpt.*;

public class TreeBasedNode extends Thread {
   int N;
   int L;
   int myW;
   int maxW;
   int swapPending;
   int splitPending;

   XMsg Inmessage;
   XMsg Outmessage;
   OutputXMsgStream oParent;
   OutputXMsgStream oLeftChild;
   OutputXMsgStream oRightChild;
   private Queue<XMsg> WhiteMsg_Q = new LinkedList<XMsg>();
   private Queue<XMsg> RedMsg_Q = new LinkedList<XMsg>();

   public TreeBasedNode(int rNodePT, int rNodeID, int me, int l, int
                                          numnodes, int port, int w, int myw) {
                this.myprocId = me;
                this.N = numnodes;
                this.L = l;
                this.W_total = w;
                this.myW = this.maxW = myw;
                this.inS = new InputXMsgStream(Integer.toString(port));
                this.recordedState = false;
                this.rootNodePT = rNodePT;
                this.rootNodeID = rNodeID;
          this.inS = new InputXMsgStream(rt.port);
   }
}
```

Figure 14: TreeBasedNode Constructor.

```
import xmpt.*;
import java.util.*;

public class CentralizedNode extends Thread {

   int myprocId;
   int N;
   int TotalDefecitTokens = 0;
   XMsg Inmessage;
   XMsg toRootMsg;
   InputXMsgStream inS;
   OutputXMsgStream oRootNode;
   int rootNodePort = 0;
   int rootNodeID = 0;

   public CentralizedNode(int rNode, int rNodeID, int me, int n, int port,
                                                                  int w) {
         this.myprocId = me;
         this.N = n;
         this.rootNodeID = rNodeID;
         this.rootNodePort = rNode;
         this.TotalDefecitTokens = w;
         this.inS = new InputXMsgStream(Integer.toString(port));
         if(me != rNodeID) {
                  // establish this node's output stream to RootNode
                  if (oRootNode == null)
                           oRootNode = new OutputXMsgStream("localhost",
                                    Integer.toString(rootNodePort));
         }
   }
}
```

Figure 15: CentralizedNode Constructor.

```
public class ExternalTrigger extends Thread
{
        Integer W;
        Random generator;
        Integer N;
        Integer rootPort;
        InputXMsgStream inS;  // input XMsgStream for trigger thread
        OutputXMsgStream outS;  // output XMsgStream for trigger thread
        XMsg TriggerMessage;

        public ExternalTrigger(int numTriggers, int numNodes, int rootPT)
        {
                this.N = numNodes;
                this.W = numTriggers;
                this.rootPort = rootPT;
                this.generator = new Random();
                this.TriggerMessage = new XMsg("<XMsg> </XMsg>");
                // initialize output stream for trigger thread
                this.outS = new OutputXMsgStream(hostname, rootPort);
        }

        public void run()
        {
           try {
                for (int cnt = 0; cnt < this.W; cnt++)
                {
                   int randomTarget = generator.nextInt(N) + rootPort;
                   outS.setConnect(hostname, Integer.toString(randomTarget));
                   TriggerMessage.setMsg("<XMsg color='" + "white" +
                   "' type='" + "trigger" + "'>" + "<triggerID>" + cnt +
                   "</triggerID>" + "</XMsg>");

                   outS.sendXMsg(TriggerMessage);
                   outS.closeSocket();
                }
           } catch (Exception e) {
                e.printStackTrace();
        }
}
```

Figure 16: The ExternalTrigger Class extends Thread and is responsible for delivering the trigger messages to the distributed system.

```
package xmpt;
import java.io.*
import java.net.Socket;

public class OutputXMsgStream {
   ObjectOutputStream oStream;
   Socket server;
   String destIP;
   String destPort;

   public OutputXMsgStream(String destIP, String destPort) {
         super();
         this.destIP = destIP;
         this.destPort = destPort;
         // set up session with destIP:destPort
         try {
            server = new Socket(destIP, Integer.parseInt(destPort));
            oStream = new ObjectOutputStream(server.getOutputStream());

            } catch (IOException e) {
                  e.printStackTrace();
            }
   }


   public void sendXMsg(XMsg m) throws Exception {
         try {
                  oStream.writeObject(m);
                  oStream.flush();
         } catch (Exception e) {
                  e.printStackTrace();
                  oStream.close();
         }
   }
}
```

Figure 17: The OutputXMsgStream Class is responsible for establishing the outgoing channel TCP connection.

```
package xmpt;
import java.io.*;
import java.net.*;
import java.util.*;

public class InputXMsgStream {
   ObjectInputStream iStream;
   ServerSocket listener;
   private SynchronousQueue<XMsg> sync;

   public InputXMsgStream(String listenerPort) {
         super();
         this.sync = new SynchronousQueue<XMsg>();
         InputXMsgStream.Listener listener = null;
         try {listener = new Listener(listenerPort);
         } catch (NumberFormatException e) {e.printStackTrace();
         } catch (IOException e) {e.printStackTrace();
         }
         listener.start();
   }
   private class Listener extends Thread {
         ServerSocket listenerSocket;
         public Listener(String port) throws NumberFormatException,
                                                          IOException {
               this.listenerSocket = new ServerSocket(Integer.parseInt(port));
         }
         public void run() {
               for (;;) {
                     try {
                           // block until a connection is made
                           Socket clientSocket = listenerSocket.accept();
                           Handler handler = new
                           InputXMsgStream.Handler(clientSocket);
                           handler.start();
                     } catch (IOException e) {e.printStackTrace();
                     }
               }
         }
   }
```

Figure 18: The InputXMsgStream Class is responsible for accepting TCP connections and maintaining a message queue for the processor node.

```
        private class Handler extends Thread {
                Socket client;
                ObjectInputStream iStream;

                public Handler(Socket client) throws IOException {
                        this.client = client;
                        this.iStream = new
                        ObjectInputStream(this.client.getInputStream());
                }

                public void run() {
                        for (;;) {
                                try {
                                        Object o = this.iStream.readObject();
                                        XMsg m = (XMsg) o;
                                        sync.put(m);
                                } catch (EOFException e) {
                                        try {
                                                this.iStream.close();
                                        } catch (IOException e1) {
                                                e1.printStackTrace();
                                        }
                                        break;
                                } catch (Exception e) {e.printStackTrace();
                                        continue;
                                }
                        }
        }

        public XMsg receiveXMsg() throws Exception {
                XMsg m = sync.take();
                return m;
        }

        public XMsg receiveXMsg(long timeout, TimeUnit unit) throws
                                                        InterruptedException {
                XMsg m = sync.poll(timeout, unit);
                return m;
        }
}
```

Figure 19: InputXMsgStream Class continued.

```
package xmpt;
import java.io.*;
import javax.xml.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class XMsg implements Serializable {
        private static final long serialVersionUID = 1L;
        private Document domDoc;

        public XMsg(String xmlStr) {
                super();
                this.setMsg(xmlStr);
        }
        public void setMsg(String xmlStr) {
                try {
                        InputSource inSrc = new InputSource(new
                        StringReader(xmlStr));
                        DocumentBuilderFactory dbf =
                        DocumentBuilderFactory.newInstance();
                        DocumentBuilder db = dbf.newDocumentBuilder();
                        this.domDoc = db.parse(inSrc);
                } catch (Exception e) {e.printStackTrace(System.err);}
        }
        public String serialize() {
                DOMSource dom = new DOMSource(this.domDoc);
                StringWriter writer = new StringWriter();
                StreamResult stream = new StreamResult(writer);
                try {
                        TransformerFactory tf;
                        Transformer t;
                        tf = TransformerFactory.newInstance();
                        t = tf.newTransformer();
                        t.transform(dom, stream);
                } catch (Exception e) {e.printStackTrace(System.err);
                }
                String serialization = writer.toString();
                return serialization;
        }
}
```

Figure 20: The XMsg Class is used for all messages in the distributed system simulation.

```
public void sendtoChildren (String msgtype, String msgpri, String msgtag, int
msgval) {
        XMsg BcastMsg = new XMsg("<?xml version=\"1.0\"
encoding=\"UTF-8\"?><XMsg> </XMsg>");

        // establish this node's output streams
        // establish this node's output streams
        if (oLeftChild == null)
                oLeftChild = new OutputXMsgStream("localhost",
Integer.toString(rootNodePort + ((2 * (myprocId + 1)) - 1)));
        if (oRightChild == null)
                oRightChild = new OutputXMsgStream("localhost",
Integer.toString(rootNodePort  + (2 * (myprocId + 1))));

        BcastMsg.setMsg("<XMsg qpri='" + msgpri + "' type='" + msgtype +
                        "'>"+ "<rnd>" + RoundNum  +  "</rnd>" +
                        "<"+msgtag+">" + msgval +  "</"+msgtag+">" +
                        "</XMsg>");

          try {
                oLeftChild.sendXMsg(BcastMsg);
          } catch (Exception e) {
                e.printStackTrace();
          }

          try {
                oRightChild.sendXMsg(BcastMsg);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
```

Figure 21: The sendtoChildren method is commonly used among all processor nodes when performing a converge cast computation.

```
// send a message to this node's parent
public void sendtoParent (String msgtype, String msgpri, String msgtag,
                                                        int msgval) {
        XMsg msg = new XMsg("<?xml version=\"1.0\" encoding=\"UTF-
8\"?><XMsg> </XMsg>");

        msg.setMsg("<XMsg qpri='" + msgpri + "' type='" + msgtype + "'>"
                +       "<rnd>" + RoundNum
                +       "</rnd>"
                +       "<pid>" + myprocId
                +       "</pid>"
                +       "<"+msgtag+">" + msgval
                +       "</"+msgtag+">"
                +       "</XMsg>");

        try {
                oParent.sendXMsg(msg);
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

Figure 22: The sendtoParent method is commonly used among all processor nodes when performing a converge cast computation.

# Glossary

MaxLoad - A MaxLoad for a global snapshot algorithm refers to the maximum number of messages sent and received by any processor node in the system.

Message Complexity - The message complexity for a global snapshot algorithm refers to the number of messages exchanged between the processors.

Shared Clock - This refers to a relation based on the physical time model. In distributed systems the processor nodes do not have a shared clock and instead uses a happened-before model or logical clock for a distributed computation.

Converge cast – This refers to the operation where a node in a distributed system uses a spanning tree that connects all nodes in the system to broadcast a message to all other processor nodes in the system. Those processor nodes then respond appropriately to the message and the responses converge back the tree to the node that initiated the broadcast.

# References


[CL85] K.M. Chandy and L. Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. Computer Systems, vol. 3, no. 1, pp. 63-75, 1985.

[CSG10] Venkatesan T. Chakaravarthy, Anamitra R. Choudhury, Vijay K. Garg, Yogish Sabharwal, *A Decentralized Algorithm for Distributed Trigger Counting*. Proc. International Conference on Distributed Computing and Networking (ICDCN), Jan 2011.

[Gar04] Garg, V. K., *Concurrent and Distributed Computing in Java*, John Wiley & Sons, 2004.

[GGS06] R. Garg, V. Garg, and Y. Sabharwal, *Scalable Algorithms for Global Snapshots in Distributed Systems*, Proc. 20th Ann. Conf. Supercomputing, pp. 269-277, Nov. 2006.

[GGS10] R. Garg, V. Garg, and Y. Sabharwal, *Efficient Algorithms for Global Snapshots in Large Distributed Systems*, IEEE Trans. Parallel and Distributed Systems, vol. 21, no. 5, pp. 620-630, May 2010.

[JMH89] J.M. Helary, *Observing Global States of Asynchronous Dsitributed Algorithms*, LNCS 392 (Belink: Springer) pp.124-134, 1989.

[KRS05] A. Kshemkalyani, M. Raynal, and M. Singhal, *An Introduction to Snapshot Algorithms in Distributed Computing, Distributed Systems Eng*., vol. 2, no. 4, pp. 224-233, 1995.

[Mat93] F. Mattern, *Efficient algorithms for distributed snapshots and global vrtual time approximation,* Journal of Parallel and Distrubted Computing, pages 423-434, August 1993.

[SK86] M. Spezialetti and P. Kearns, *Efficient distributed snapshots*, 6th Int. Conf. on Distributed Computing Systems pp. 382-388, 1986.

[Ven93] S. Venkatesan, *Message-optimal Incremental Snapshots*, Journal of Computer Software Engineering 1(3) pp. 211-231, 1993.

# Vita

Juan Manuel Casas Jr. was born in McAllen, Texas. After completing his work at PSJA High School, San Juan, Texas, in 1997, he entered The University of Texas-Pan American in Edinburg, Texas. During the summer of 2005 he attended Texas A&M University as undergraduate research assistance where he co-published an article appearing in the Transaction on Networking Journal Volume 13 Issue 5. He received the degree of Bachelor of Science from The University of Texas-Pan American in 2006. During the following years, he has been employed as a Software Engineer with IBM Systems and Technology Group in Austin, Texas. In January, 2009, he entered the Graduate School at the University of Texas at Austin.

Permanent address: 1806 Wallin Loop
Round Rock, Texas 78664

This report was typed by the author.