SOMETHING SOMETHING GROUP MANAGEMENT

by

BENJAMIN HENRY PAYNE

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2013

Approved by

Dr. Bruce McMillin, Advisor
Dr. Alireza Hurson
Dr. Wei Jiang
Dr. Sriram Chellappan
Dr. Sahra Sedighsarvestani

ABSTRACT

SOMETHING SOMETHING GROUP MANAGEMENT

by

BENJAMIN HENRY PAYNE

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

2013

Approved by

Dr. Bruce McMillin, Advisor
Dr. Alireza Hurson
Dr. Wei Jiang

Abstract

Cyber-physical systems (CPS) can improve the reliability of our critical infrastructure systems. By augmenting physical distribution systems with digital control through computation and communication, one can improve the overall reliability of a network. However, a system designer must consider what effects system unreliability in the cyber-domain can have on the physical distribution system. In this work, we examine the consequences of network unreliability on a core part of a Distributed Grid Intelligence (DGI) for the FREEDM (Future Renewable Electric Energy Delivery and Management) Project. To do this, we apply different rates of packet loss in specific configurations to the communication stack of the software and observe the behavior of a critical component (Group Management) under those conditions. These components will allow us to identify the amount of time spent in a group, working, as a function of the network reliability.

ACKNOWLEDGMENTS

The authors acknowledge the support of the Future Renewable Electric Energy Delivery and Management Center, a National Science Foundation supported Engineering Research Center under grant NSF EEC-081212, and the United States Department of Education GAANN program.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

Table Page

1. INTRODUCTION

Historically, leader elections have had limited applications in critical systems. However, in the smart grid domain, there is a great opportunity to apply leader election algorithms in a directly beneficial way. [1] presented a simple scheme for performing power distribution and stabilization that relies on formed groups. Algorithms like Zhang, et. al's Incremental Consensus Algorithm [2], begin with the assumption that there is a group of nodes who coordinate to distribute power. In a system where 100% up time is not guaranteed, leader elections are a promising method of establishing these groups.

A strong cyber-physical system should be able to survive and adapt to network outages in both the physical and cyber domains. When one of these outages occurs, the physical or cyber components must take corrective action to allow the rest of the network to continue operating normally. Additionally, other nodes may need to react to the state change of the failed node. In the realm of computing, algorithms for managing and detecting when other nodes have failed is a common distributed systems problem known as leader election.

This work observes the effects of network unreliability on the the group management module of the Distributed Grid Intelligence (DGI) used by the FREEDM smart-grid project. This system uses a broker system architecture to coordinate several software modules that form a control system for a smart power grid. These modules include: group management, which handles coordinating nodes via leader election; state collection, a module which captures a global system state; and load balancing which uses the captured global state to bring the system to a stable state.

It is important for the designer of a cyber-physical system to consider what effects the cyber components will have on the overall system. Failures in the cyber

domain can lead to critical instabilities which bring down the entire system if not handled properly. In fact, there is a major shortage of work within the realm of the effects cyber outages have on CPSs [3] [4]. In this paper we present a slice of what sort of analysis can be performed on a distributed cyber control by subjecting the system to packet loss. The analysis focuses on quantifiable changes in the amount of time a node of the system could spend participating in energy management with other nodes.

2. RELATED WORK

2.1. VIRTUAL SYNCHRONY

(Do I need to describe the difference between Recieve and Deliver?)

Consider a distributed system where events can be causually related based on their execution on the local processors and communication between processes, as defined in CITE. Virtually synchronous systems use multicast to communicate reliabily between proceses.

1. If e and e' are events local to a process P_i and e occurs before e', then $e \rightarrow e'$
2. If $e = send(m) and e' = deliver(m)$ for the same message m, then $e \rightarrow e'$

This defines a dependence and casuality relation between events in the system. If two events cannot be causually related (that is $e \not\rightarrow e'$ and $e' \not\rightarrow e$ then the events are considered concurrent. From this one can define an execution history H.

Virtual synchrony is a concept employed by a great number of distributed frameworks to impose synchronization on a system. Virtual synchrony is a model of communication and execution that allows the system designer to enforce synchronization on the system. Although the processes do not execute tasks simultaneously, the execution history of each process cannot be differenitated from a trace where tasks are executed simultaneously. A pair of histories H and H' are equivalent if for each procress in the system (noted as p) $H|_p$ cannot be differenatiated $H'|_p$ based on the casual relationships between the events in the history.

Additionally, a history is considered complete if all sent messages are delivered and there are no casual holes. A casual hole is a circumstance where an event e is casually related to e' by $e' \rightarrow e$ and e appears in a history, but e' is not.

A virtually synchronous system is one where each history the system is indistinguishable from all histories produced by a synchronous system.

2.1.1. Process Groups. Virtual synchrony models also support process groups. Although each implementation of a virtually synchron/ous system applies a different structure to the way processes are grouped, there are common features between each implementation.

A process group in a virtual synchronous system is commonly refered to as a view. A view is a collection of processes that are virtually synchronous with eachother. Process groups in virtually synchronous systems place obligations on the delivery of messages to members of the view. A history H is legal if:

1. Given a function $time(e)$ that returns a global time of when the event occured e, then $e \rightarrow e' \Rightarrow time(e) < time(e')$.
2. $time(e) \neq time(e') \forall e, e' \in H|_p$ (where $e \neq e'$) for each process.
3. A change in view (group membership) occur at the same logical time for all processes in the view.
4. All multicast message deliveres occur in the view of a group. That is, if a message is sent in a view, it is delivered in that view, for every process in that view.
5. Atomic broadcast (*abcast*) messages are totally ordered.

A process group interacting creates a legal history for the virtually synchronous system. However, processes can fail. The crash-failure model is commonly used. To achieve this, the virtual synchrony model has the following properties:

- The system employs a membership service. This service monitors for failures and reports them to the other processes as part of the process group (or view) system.

- When a process is identified as failing it is removed from the groups that it belongs to and the remaining processes determine a new view.
- After a process has been identified as failing, no message will be received from it.

As part of the failure model, it is worth noting two commonly used multicast delivery garuntees: uniform and non-uniform. The uniform propery obligates that if one multicast is delivered to a node in the current view, it is delivered to all nodes in the current view.

2.2. EXTENDED VIRTUAL SYNCHRONY

One of the major shortcomings of the original virtual synchrony model lay in how it handled network partitions. In virtual synchrony, when the network partitioned, only processes in the primary partition were allowed to continue. Processes that were not in the primary partition could not rejoin the primary partition without being restarted, that is, the process joining the view must be a new process so that there are no message delivery obligations for that process in the view.

Mosler et. al's group at the University of California Santa Barbara designed an extended version of virtual synchrony, dubbed extended virtual synchrony. Extended virtual synchrony is compatible with the original virtual synchrony, and can implement all the functionality and limitations of the original design, as specified in the original paper. Because it supports virtual synchrony, extended virtual synchrony has become the basis for a number of related frameworks that have been designed since Isis including Horus, Totem, Transis, and Spread.

Note that in Mosler's work, the name view has been substituted for configuration. For consistency in this section, the word view will be used to describe a group or a configuration, and all three terms are equivalent.

Extended virtual synchrony places the following obligations on the message delivery service, described informally below:

1. As defined in Virtual Synchrony, events can be cansually related. Furthermore, if a message is delivered, the delivery is casually related to the send event for that message.

2. If a message m is sent in some view c by some process p then p cannot send m in some other view c'

3. If not all processes install a view or a process leaves a view, a new view is created. Furthermore, views are unique and events occur after a view's creation and before its destruction. Messages which are delivered before a view's destruction must be delivered by all processes in that view (unless a process fails). Similarly, a message delivery which occurs after the creation of a view must occur after the view is installed by every process in the view.

4. Every sent message is delivered by the process that sends it (unless it fails) even if the message is only delivered to that process.

5. If two processes are in sequential views, they deliver the same set of messages in the second configuration.

6. If the send events of two messages is casually related, then if the second of those messages is delivered, the first message is also delivered.

7. Messages delivered in total order must delivered at the same logical time. Additionally, this total order must be consistent with the partial, casual order. When a view changes, a process is not obligated to deliver messages for processes that are not in the same view.

8. If one process in a view delivers a message, all processes in that view deliver the message, unless that process fails. If an event which delivers a message in some view occurs, then the messages that installed that view was also delivered.

Extended virtual synchrony lifts the obligation that messages will no longer be received by processes that have been removed by a view. It also provides additional restrictions on the sending of messages between two different views (Item 2) and an obligation on the delivery of messages (Item 4).

The concept of a primary view or primary configuration still exists within the extended virtual synchrony model and is still described by the notion of being the largest view. Since views can now partition and rejoin the primary partition the rules presented above also allow the history of the primary partition to be totally ordered. Additionally, two consecutive primary views have at least process that was a member of each view.

To join views, processes are first informed of a failure (or joinable partition) by a membership service. Processes maintain an obligation set of messages they have acknowledged but not yet delivered. After being informed of the need to change views, the processes begin buffering received messages and transition into a transitional view. In the transitional view messages are sent and delivered by the processes to fulfill the casuality and ordering requirements listed above. Once all messages have be transmitted, received and delivered as needed and the processes obligation set is empty, the view transitions from a transitional view to a regular view and execution continues as normal.

2.2.1. Comparison To DGI. The DGI places similar but distinct requirements on execution of processes in its system. The DGI enforces synchronization between processes that obligates each active process to enter each module's phase simultaneously. This is fulfilled by using a clock synchronization algorithm. The

virtual synchrony model, at its most basic, does not require a clock to enforce its ordering.

However, this simplifies the DGI fulfilling it's real time requirements. processes must be able to react to changes in the power system within a maximum amount of time. These interactions do not require interaction between all processes within the group. Additionally, this allows for private transactions to occur between DGI processes. Furthermore, hardware performance is limited in the current specification of the DGI platform, since the project is currently targeting a low power ARM board for deployment.

The DGI has also been implemented using message delivery schemes that are unicast instead of multicast, since this is the easiest to achieve in practice. A majority of systems implementing virtual synchrony use a model where local area communication is emphasized, with additional structures in place to transfer information across a WAN to other process groups.

Groups in DGI are not obligated to deliver any subset of the messages to any peer in the system. Some of the employed algorithms in DGI need all of the messages to be delivered in a timely manner, but they do not require the same message to be delivered to every peer in the current design of the system.

2.3. ISIS (1989) AND HORUS (1996)

(Note: I want it to be clear that Isis is Virtual Synchrony, Horus supports extended through the correct application of layers, but doesn't have to)

A product of Dr. Kenneth Birman and his group at Cornell, Isis and Horus are two distributed frameworks which are comparable to the DGI. Although these projects are no longer actively developed, Isis and Horus are the foundation which all other virtual synchrony frameworks are based.

Isis was originally developed to create a reliable distributed framework for creating other applications. As Isis was one of the first of its kind, the burden of maintaining a robust framework that met the development needs of its users eventually made Birman's group create a newer, updated framework called Horus, which implemented the improved extended virtual synchrony model. However, Isis and Horus largely implement the same concepts.

Both Horus and Isis are described as a group communications system. They provide a messaging architecture which clients use to deliver messages between processes. The frameworks provide a reliabile distributed multicast, and a failure detection scheme. Horus offered a modular design with a variety of extensions which could change message characteristics and performance as needed by the project.

2.3.1. Group Model. In Horus and Isis, a group is a collection of processes which can communicate with one another to do work. Multicasts are directed to the group, and are gaurnteed to be recieved by all members or no members. The collection of processes which make up a group is called a view.

Over time, due to failure, the view will change. Since views are distributed concurrently and asynchronously, each process can have a different view. Horus is designed to have a modular communication structure composed of layers, which allows the communication channel to have different properties which will affect which messages will be delivered in the event of a view change. Horus' layers allow developers to go as far to not use the complete virtual synchrony model, if the programmer desires. For example, Horus can implement total order using a token passing layer, or casual ordering using vector clocks.

Isis has limited support for partitioning. In the event that a partition forms, dividing the large group into subgroups, only the primary group is allowed to continue operating. The primary group is selected by choosing the largest partition. Horus,

on the other hand implements the extended virtual synchrony model and does not have that limtitation.

2.4. TRANSIS (1992)

(Transis is similar to virtual synchrony but supports partitions, but is not explicitly extended virtual synchrony)

Transis was developed as a more pragmatic approach to the creation of a distributed framework. Transis is developed with the key consideration that, while multicast is the most efficient for distributing information in a view or group of processors (as point to point quickly gives rise to N^2 complexity) it can be impractical, especially over a wide area network to rely on broadcast to deliver messages. Transis, then considers two components for the network: a local area component and a wide area component.

The local area component, Lansis, is responsible for the excange of messages across a LAN. Transis uses a combination of acknowledgements, which are piggy-backed on regular messages to identify lost messages. If a process observes a message being acknowledged that it does not receive then it sends a negative acknowledgement broadcast informing the other members of the view it did not receive that message. In Lansis, the acknowledgement signals that a process is ready to deliver a message. Lansis assumes that all messages can be casually related in a global, directed, acyclic graph and there are a number of schemes that deliver messages based on adherance to that graph.

Like other frameworks, Transis uses gateways, which that call Xporters to deliver messages between the local views.

2.5. TOTEM (1996)

(Totem should clearly be Extended Virtual Synchrony)

Totem is largely in the same vein as Isis. Developed at the University of California, Santa Barbara by Moser, Melliar-Smith, Agarwal et. al. Totem is designed to use local area networks, connected by gateways. The local area networks use a token passing ring and multicast the messages, much like Isis and Horus. The gateways join these rings into a multi-ring topology. Messages are first delivered on the ring which they are sent, then forwarded by the gateway which connects the local ring to the wide area ring for delivery to the other rings. The message protocol gives the message total ordering using a token passing protocol. Topology changes are handled in the local ring, then forwarded through the gateway where the system determines if the local change necessitates a change in the wide area ring. Failure detection is also implemented to detect failed gateways.

2.6. SPREAD

3. BACKGROUND THEORY

3.1. FREEDM DGI

The FREEDM DGI is a smart grid operating system that organizes and coordinates power electronics and negotiates contracts to deliver power to devices and regions that cannot effectively facilitate their own need.

To accomplish this, the DGI software consists of a central component, the broker, which is responsible for presenting a communication interface and furnishing any common functionality needed by any algorithms used by the system. These algorithms are grouped into modules.

The intial work this document uses a version of the FREEDM DGI software with only one module: group management. Group management implements a leader election algorithm to discover which nodes are reachable in the cyber domain.

3.2. BROKER ARCHITECTURE

The DGI software is designed around the broker architecture specification. Each core functionality of the system is implemented within a module which is provided access to core interfaces which deliver functionality such as scheduling requests, message passing, and a framework to manipulate physical devices, including those which exist only in simulation environments such as PSCAD[5] and RSCAD[6].

The Broker provides a common message passing interface which all modules are allowed access to. This interface also provides the inter-module communication which delivers messages between software modules, effectively decoupling them outside of the requirement for them to be able to recognize messages addressed to them from other modules.

Several of the distributed algorithms used in the software require the use of ordered communication channels. To achieve this, FREEDM provides a reliable ordered communication protocol (The sequenced reliable connection or SRC) to the modules, as well as a "best effort" protocol (The sequenced unreliable connection or SUC) which is also FIFO (first in, first out), but provides limited delivery guarantees.

We elected to design and implement our own simple message delivery schemes in order to avoid complexities introduced by using TCP in our system. During development, it was observed that constructing a TCP connection to a node that had failed or was unreachable took a considerable amount of time. We elected to use UDP packets which do not have those issues, since the protocol is connectionless. To accomplish this lightweight protocols which are best effort oriented were implemented to deliver messages as quickly as possible within the following requirements.

3.2.1. Sequenced Reliable Connection.. The sequenced reliable connection is a modified send and wait protocol with the ability to stop resending messages and move on to the next one in the queue if the message delivery time exceeds some timeout. When designing this scheme we wanted to achieve several criteria:

- Messages must be accepted in order - Some distributed algorithms rely on the assumption that the underlying message channel is FIFO.
- Messages can become irrelevant - Some messages may only have a short period in which they are worth sending. Outside of that time period, they should be considered inconsequential and should be skipped. To achieve this, we have added message expiration times. After a certain amount of time has passed, the sender will no longer attempt to write that message to the channel. Instead, he will proceed to the next unexpired message and attach a "kill" value to the message being sent, with the number of the last message the sender knows the receiver accepted.

- As much effort as possible should be applied to deliver a message while it is still relevant.

There one adjustable parameter, the resend time, which controls how often the system would attempt to deliver a message it hadn't yet received an acknowledgment for.

To further explain the characteristics of the protocol, the psuedocode is included below:

$inseqno \leftarrow 0$

$outseqno \leftarrow 1$

$outqueue \leftarrow []$

$kill \leftarrow null$

$lastack \leftarrow 0$

function RECEIVE(msg)

if $msg.type = MSG$ **then**

if $msg.seqno = inseqno + 1$ **then**

$SendAck(msg.seqno)$

$inseqno \leftarrow inseqno + 1$

else if $msg.seqno > inseqno$ and $msg.kill \neq null$ and $msg.kill \leq inseqno$ **then**

$SendAck(msg.seqno)$

$inseqno \leftarrow msg.seqno + 1$

else

$SendAck(inseqno)$

end if

else if $msg.type = ACK$ **then**

if $msg.seqno = outqueue.front.seqno$ **then**

$outqueue.pop()$

```
            kill ← null
            lastack ← msg.seqno
            Write(outqueue.front, kill)
        else
            Write(outqueue.front, kill)
        end if
    end if
end function
function SEND(msg)
    msg.seqno ← outseqno
    outseqno ← outseqno + 1
    outqueue.push(msg)
    if outqueue.size = 0 then
        Write(outqueue.front, kill)
    end if
end function
function RESEND
    while outqueue.size ≥ 0 and outqueue.front.expired do
        outqueue.pop()
        kill ← lastack
    end while
    if outqueue.size ≥ 0 then
        Write(outqueue.front, kill)
    end if
end function
```

Note that the *Resend*() function is periodically called to attempt to redeliver lost messages to the reciever. This is, of course, a version with unbounded sequence

numbers. The implementation available with the FREEDM source code is modified to allow for bounded sequence numbers.

3.2.2. Sequenced Unreliable Connection.. The SUC protocol is simply a best effort protocol: it employs a sliding window to try to deliver messages as quickly as possible. A window size is decided, and then at any given time, the sender can have up to that many messages in the channel, awaiting acknowledgment. The receiver will look for increasing sequence numbers, and disregard any message that is of a lower sequence number than is expected. The purpose of this protocol is to implement a bare minimum: messages are accepted in the order they are sent.

Like the SRC protocol, the SUC protocol's resend time can be adjusted. Additionally, the window size is also configurable, but was left unchanged for the tests presented in this work.

The psuedocode is included for clarity below:

$inseqno \leftarrow 0$

$outseqno \leftarrow 0$

$outqueue \leftarrow []$

function RECEIVE(msg)

  if $msg.type = MSG$ **then**

    if $msg.seqno > inseqno$ **then**

      $SendAck(msg.seqno)$

      $inseqno \leftarrow msg.seqno$

    else

      $SendAck(inseqno)$

    end if

  else if $msg.type = ACK$ **then**

    $popped \leftarrow 0$

    if $msg.seqno \leq outqueue.front.seqno$ **then**

```
            outqueue.pop()
            popped ← popped + 1
        end if
        for i = WindowSize − popped → min(WindowSize − 1, outqueue.size) do
            Write(outqueue[i])
        end for
    end if
end function
function SEND(msg)
    msg.seqno ← outseqno
    outseqno ← outseqno + 1
    outqueue.push(msg)
    if outqueue.size ≤ WindowSize then
        Write(msg)
    end if
end function
function RESEND
    for i = 0 → min(WindowSize − 1, outqueue.size) do
        Write(outqueue[i])
    end for
end function
```

3.3. GROUP MANAGEMENT ALGORITHM

Our software uses a leader election algorithm, "Invitation Election Algorithm" written by Garcia-Molina and listed in [7]. His algorithm provides a robust election procedure which allows for transient partitions. Transient partitions are formed when

a faulty link between two or more clusters of DGIs causes the groups to temporarily divide. These transient partitions merge when the link is more reliable. The election algorithm allows for failures that disconnect two distinct sub-networks. These sub networks are fully connected, but connectivity between the two sub-networks is limited by an unreliable link.

$AllNodes \leftarrow \{1, 2, ..., N\}$

$Coordinators \leftarrow \emptyset$

$UpNodes \leftarrow Me$

$State \leftarrow Normal$

$Coordinator \leftarrow Me$

$Responses \leftarrow \emptyset$

$Counter \leftarrow$ A random initial identifier

$GroupID \leftarrow (Me, Counter)$

function CHECK

This function is called periodically by the leader

if $State = Normal$ and $Coordinator \leftarrow Me$ **then**

$Responses \leftarrow \emptyset$

$TempSet \leftarrow \emptyset$

for $j = (AllNodes - \{Me\})$ **do**

$AreYouCoordinator(j)$

$TempSet \leftarrow TempSet \cup j$

end for

Nodes which respond "Yes" to $AreYouCoordinator$ are put into the $Responses$ set. When all nodes have responded or after $Timeout(CheckTimeout)$, Nodes that do not respond are removed from UpNodes and execution continues

$UpNodes \leftarrow (TempSet - Responses) \cup Me$

if $Responses = \emptyset$ **then return**

end if

$p \leftarrow \max(Responses)$

if **then**$Me < P$

Wait time proportional to p-i

end ifMERGE(Responses)

end if

The next call to this is after Timeout(CheckTimeout)

end function

function TIMEOUT

This function is called periodically by the group members

if $Coordinator = Me$ **then return**

elseARE YOU THERE(Coordinator,GroupID,Me)

if Response is No or after $Timeout(TimeoutTimeout)$ **then** RECOVERY

end if

end if

The next call to this is after Timeout(TimeoutTimeout)

end function

function MERGE(Coordinators)

This function invites all coordinators in Coordinators to join a group led by Me

$State \leftarrow Election$

Stop work

$Counter \leftarrow Counter + 1$

$GroupID \leftarrow (Me, Counter)$

$Coordinator \leftarrow Me$

$TempSet \leftarrow UpNodes - Me$

$UpNodes \leftarrow \emptyset$

for $j \in Coordinators$ **do** INVITE(j,Coordinator,GroupID)

end for

for $j \in TempSet$ **do** INVITE(j,Coordinator,GroupID)

end for

Wait for $Timeout(InviteTimeout)$, Nodes that accept the invite are added to UpNodes

$State \leftarrow Reorganization$

for $j \in UpNodes$ **do** READY(j,Coordinator,GroupID,UpNodes)

end for

$State \leftarrow Normal$

end function

function RECEIVEREADY(Sender,Leader, Identifier, Peers)

if $State = Reorganization$ and $GroupID = Identifier$ **then**

$UpNodes \leftarrow Peers$

$State \leftarrow Normal$

end if

end function

function RECEIVEAREYOUCOORDINATOR(Sender)

if $State = Normal$ and $Coordinator = Me$ **then**

Respond Yes

else

Respond No

end if

end function

function ReceiveAreYouThere(Sender, Identifier)

if $GroupID = Identifier$ and $Coordinator = Me$ and $Sender \in UpNodes$ **then**

Respond Yes

else

Respond No

end if

end function

function ReceiveInvitation(Sender,Leader,Identifier)

if $State \neq Normal$ **then return**

end if

Stop Work

$Temp \leftarrow Coordinator$

$TempSet \leftarrow UpNodes$

$State \leftarrow Election$

$Coordinator \leftarrow Leader$

$GroupID \leftarrow Identifier$

if $Temp = Me$ **then**

Forward invite to old group members

for do$j \in TempSet$

$Invite(j, Coordinator, GroupID)$

end for

end if

$Accept(Coordinator, GroupID)$

$State \leftarrow Reorganization$

if $Timeout(ReadyTimeout)$ expires before *Ready* is recieved **then**

$Recovery()$

end if

end function

function RECEIVEACCEPT(Sender,Leader,Identifier)

if $State \leftarrow Election$ and $GroupID = Identifier$ and $Coordinator = Leader$ **then**

$UpNodes \leftarrow UpNodes \cup Sender$

end if

end function

function RECOVERY

$State \leftarrow Election$

Stop Work

$Counter \leftarrow Counter + 1$

$GroupID \leftarrow (Me, Counter)$

$Coordinator \leftarrow Me$

$UpNodes \leftarrow Me$

$State \leftarrow Reorganization$

$State \leftarrow Normal$

end function

The elected leader is responsible for making work assignments and identifying and merging with other coordinators when they are found, as well as maintaining a up-to-date list of peers for the members of his group. Likewise, members of the group can detect the failure of the group leader by periodically checking if the group

leader is still alive by sending a message. If the leader fails to respond, the querying node will enter a recovery state and operate alone until they can identify another coordinator to join with. Therefore, a leader and each of the members maintains a set of processes which are currently reachable, which is a subset of all known processes in the system.

This Leader election can also be classified as a sort of failure detector (CITE). Failure detectors are algorithms which detect the failure of processes in a system. A failure detector algorithm maintains a list of processes that it suspects have crashed. This informal description gives the failure detector strong ties to the Leader Election process. The Group Management module maintains a list of suspected processes which can be determined from the set of all processes and the current membership.

The leader and members have seperate roles to play in the failure detection process. The leader, using the $Check()$ function will constantly search for other leaders to join groups with. This serves as a ping / response query for detecting failures in the system. It is also capable of detecting a change in state either by network issue or crash failure that causes the process being queried to no longer consider itself part of the leaders group. The member on the other hand, as the algorithm is written will only suspect the leader, and not the other processes. Of course, simple modifications could allow the member to suspect other members by use of a heart beat or query-reply system, it is not implemented in DGI code.

In this work it is assumed that a leader does not span two partitoned networks: if a group is able to form all members have some chance of communicating with each other.

3.4. NETWORK SIMULATION

Network unreliability is simulated by dropping datagrams from specific sources on the receiver side. Each receiver was given an XML file describing the prescribed reliability of messages arriving from a specific source. The network settings were loaded at run time and could be polled if necessary for changes in the link reliability.

On receipt of a message, the broker's communication layer examine the source and select randomly based on the reliability prescribed in the XML file whether or not to drop a message. A dropped message was not delivered to any of the sub-modules and was not acknowledged by the receiver. Using this method we were able to emulate a lossy network link but not one with message delays.

3.5. SYSTEM IMPLEMENTATION

The FREEDM DGI software uses a Broker Architectural pattern. This design is realized in C++ using the Boost Library[8]. We have also make use of other languages such as Python to provide bootstrapping and start-up routines for the software.

3.6. HOW THE NETWORK RELIABILITY SIMULATOR FITS INTO THE COMMUNICATION STACK

Because the DGI's network communication is implemented using UDP, there is a listener class which is responsible for accepting all incoming messages on the socket the system is listening on. This component is responsible for querying the appropriate protocol's class to determine if a message should be accepted. To do this, when a message is received, the message is parsed by the listener. At this point the network simulation will halt processing the message if it should be discarded based

on the defined random chance in the configuration file. Otherwise, it is delivered to the addressed module.

3.7. REAL TIME

The DGI's specifications also call for real time reaction to events in the system. The DGI's real-time requirements are designed to enforce a tight upper bound on the amount of time used creating groups, discovering peers, collecting the global state, and performing migrations.

To enforce these bounds, The real-time DGI has distinct phases which modules are allowed to use for all processing. Each module is given a phase which grants it a specific amount of processor time to complete any tasks it has prepared. When the allotted time is up the scheduler changes context to the next module. This interaction is shown in Figure 3.1

Modules inform the scheduler of tasks it wishes to perform by either submitting them to be performed at some point in the future, or informing the scheduler of a tasks that is ready to be executed immediately.

Tasks that have become ready, either by being inserted as ready, or the time period that specified when it should be executed after has passed. The prepared task is inserted into a ready queue for the module that the task has been scheduled for.

When that modules phase is active, the task is pulled from the ready queue and executed. When the phase is complete, the scheduler will stop pulling tasks from the previous modules queue and begin pulling from the next modules queue.

This allows enforcement upper bound message delay. The modules have a specific amount of processing time allotted. Modules with messages that invoke responses (or a series of queries and responses) typically are required to be received

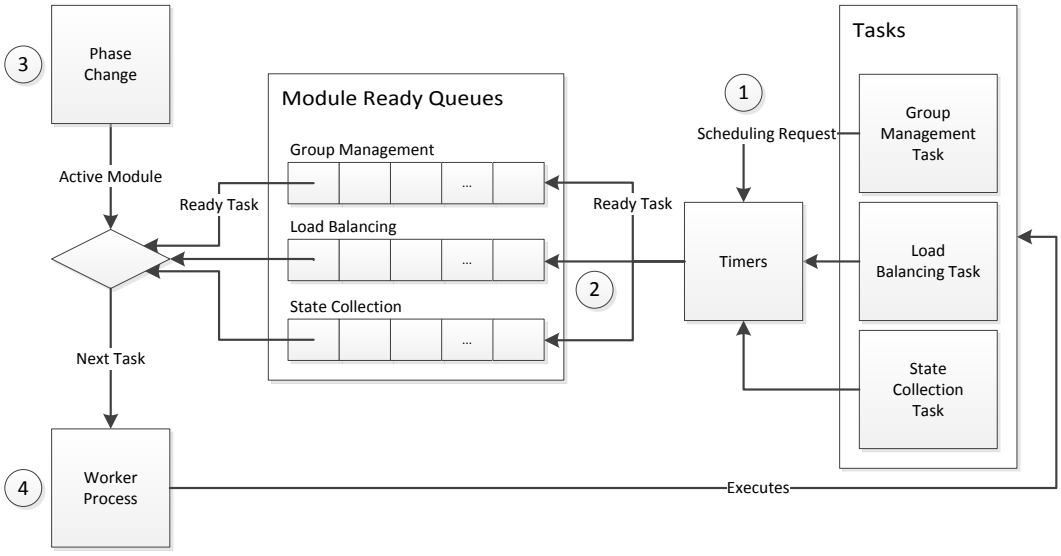

Figure 3.1: The realtime scheduler uses a round robin approach to allot execution time to modules.

1. Modules request that a task be executed by specifying a time in the future to execute a task. A timer is set to count down to the specified moment. Modules may also place tasks immediately into the ready queue if the task may be executed immediately.

2. When the timer expires the task is placed into the ready queue for the module that requested the task be executed.

3. Modules are assigned periods of execution (called phases) which are a predetermined length. After the specificed amount of time has passed, the module's phase ends and the next module in the schedule's tasks begin to execute.

4. The worker selects the next ready task for the active module from the ready queue and executes it. These tasks may also schedule other tasks to be run in the future.

within the same phase, using round numbers which enforce that the message was sent within the same phase.

Modules are designed and allotted time to allow for parameters such as maximum query-response time (based on the latency between communicating processes).

This implies that a module which engages in these activities has an upper-bound in latency before messages are considered lost.

3.8. MARKOV MODELS

Markov models are a common way of recording probabilistic processes that can be in various states which change over time. A Markov model is a directed graph composed of states, and transititons between these states. Each transition has some probability attached to them.

The models begins in some initial state, and then transitions into other states based on the probabilities assigned on each edge of the graph. Each state is memoryless, meaning that the history of the system, or the previous states have no effect on the next transition that occurs. Letting $Fx(s)$ equal the complete history of a Markov chain X up to time s, and letting $j \in S$, where S is the complete set of states in the model. CITE STOICH BIO

$$P\{X(t) = j|F_X(s)\} = P\{X(t) = j|X(s)\} \tag{3.1}$$

Additionally, the models we present are time homogenous, meaning that the time that transition probabilties are not affected by the amount of time that has passed in the simulation. CITE STOICH BIO

$$P\{X(t) = j|X(s)\} = P\{X(t) = j|X(0)\} \tag{3.2}$$

Models can be either discrete time or continous time. In a discrete time model time is divided into distinct slices. After each "slice" the system transitions based on the random chance from each of possible transitions. The discrete model also allows for a transition which returns to the same state.

To contrast, a continous time model assumes that the time between transitions are exponentially distributed. Each transition has some expected value or mean value which describes the amount of time before a transition occurs. Continous time models do not have transitions which return to the same state since the expected value of the transition time describes when how long the system remains in the same state The probability density function (PDF) of the exponential distrobution can be written as: CITE

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases} \tag{3.3}$$

As a result, the expected or mean value of an exponential distribution, is a function of the parameter λ: CITE

$$\mathrm{E}[X] = \frac{1}{\lambda}. \tag{3.4}$$

When there are multiple possible transitions from a state, each with their own expected transition time, the expected amount of time in the state cate can be written as:

Each transition lends to an expected amount of time expected in the state. Then, to do a random walk of a continous time Markov chain, an intensity matrix must also be generated in order to describe which transition is taken after the exponentially distributed amount of time in the state has passed. Consider then two streams of random variables, one of which is exponentially distributed and used to determine the amount of time in a state. The second stream is normally distributed and used to determine which state to transition to through the intensity matrix.

To collect the in group time from a Markov model the SharpE tool was used. Developed by Trivedi's team at Duke University, SharpE is a tool commonly used for

reliability and availibilty testing, and features a Markov chain tool. Data collected from the simulator was used with SharpE to collect a steady state probability (the probability that the system will be in a given state at any given instant) from the collected simulation data.

Understanding how the dynamics of group formation is captured in a Markov Model is critical for both assessing its applicability and accuracy in this application. First, however, the dynamics of group membership must be understood as part of the distributed system.

Consider a set of processes, which are linked by some packet based network protocol. In our experiments we provide two protocols, each with different delivery characteristics. Under ideal conditions a packet sent by one process will always be delivered to its destination. Without a delivery protocol, as soon as packets are lost by the communication network, the message that it contained is lost forever. Therefore to compensate for the network losing packets, a large variety of delivery protocols have been adapted. Each protocol has a different set of goals and objectives, depending on the application.

Keeping in mind that a single lost packet does not necessitate the message it contained is forever lost, different protocols allow for different levels of reliability despite packet loss.

The leader election algorithm is centered around two critical events: checking, and elections. The check system is used to detect both failures and the availability of nodes for election.

Consider a set of processes which have already formed a group. These processes occasionally exchange messages to determine if the other processes have crashed. These processes can be classified into two sets: Leaders and Members.

When a leader sends its check messages, the nodes that receive it either respond in the positive, indicating that they are also leaders, or in the negative indicating that they have already joined a group. This message is sent to all known nodes in the system. If a process replies that it is also a leader, the original sender will enter and election mode and attempt to combine groups with the first process. Nodes that fail to respond are removed from the leaders group, if they were members.

The member on the other hand will only direct its check message to the leader of its current group. As with the leader's check message, the response can either be positive or negative. A yes response indicates that the leader is still available and considers the member a part of its group. A no response indicates that either the leader has failed and recovered, or it has suspected the member process of being unreachable (either due to crash or network issue) and has removed them from the group. In this event the member will enter a recovery state and reset itself to an initial configuration where it is in a group by itself.

On any membership change, either due to recovery, or a suspected failure, the list of members for a group is pushed to every member of that group by the leader. Members cannot suspect other processes of being crashed, only the leader can identify failed group members.

During elections, a highest priority leader (identified by its process id) will send invites to the other leaders it has identified. If those leaders accept the highest priority leader's invites, they will reply with an accept message and forward the invite to their members, if their are any. If the highest priority process fails to become the leader the next highest will send invites after a specified interval has passed.

Therefore, the membership of the system can be affected in two ways: election events which change the size of groups and failure suspicion (via checks) which decreases the size of groups. Note that elections can decrease the size of groups as

well as increase them: If a round of forwarding invites fails by the new leader to his original group, the group size could decrease.

When a process is initialized it begins in the "solo" state: it is in a group with itself as the only member. As nodes are discovered by checks, the processes combine into groups. Groups are not limited by increasing one a time; they can increase by combined size of the groups of the leader processes.

The result is similar to a life death process with more complex skips forward or backward across the population.

We define a metric to assess the performance of the system under duress, we first consider that the distributed can only perform meaningful work when the processes can work together to perform physical migration. This means that there are two networks that affect the system's ability to do work: the physical and the cyber.

4. EXPERIMENTAL DESIGN

Tests were the system were completed by applying network settings and then running the nodes in the prescribed configuration for ten minutes (using the UNIX timeout command). At this point the test was terminated and the group management system appends statistics to an output file. New settings were applied and the next test was begun.

4.1. TOOLS USED, SYSTEMS USED

The application of settings and the initiation of tests was completed using a custom script written in Python. This script used a library, Fabric [9], to start runs of the system by the secure shell (SSH). This was run on one of the machine and monitored the I/O of all nodes to ensure everything was behaving correctly.

Our experimental software also provided for "bussing," where a group of edges would have the same reliability and were iterated together, and "fixing," which allowed for edges that would not change reliability across any of the runs.

All tests were run on four Pentium 4 3GHz machines with 1GB of RAM and Hyper-threading. Tests were run on an ArchLinux install using a real-time kernel, however, the snapshot of the FREEDM software used to run the tests does not feature a real-time scheduler.

The testing software was responsible for initializing instances, allowing them to run and then terminate after a fixed time limit. Additionally it provided an iterative object which generated network settings which were copied to the target machines before each test began.

Each node recorded its own state information, which was appended to a log file at termination of the run. This data was then coupled with the experimental procedure data to create the tables and charts in the results.

For each run of the system, the first 60 seconds of the system were not logged to filter out transients. This leads to a maximum recordable in-group time of nine minutes.

4.2. TESTS PERFORMED

Our experiments considered two configurations of the system which can be considered highly characteristic of most other scenarios. The first, a two node configuration was intended to observe a slice of the behavior of the system when two nodes (a leader and a group member) struggle to communicate with one another.

The second configuration was a four node configuration with a transient partition, where the nodes were divided into pairs. Each pair of nodes could reliably communicate with each other, but reliable communication across pairs was not guaranteed. We would vary the reliability of the connection between the pairs and observe the effects on the system.

For both tests, we ran the system using both our sequenced reliable protocol as well as our sequenced unreliable protocol. Additionally, we varied the amount of time between resends for both protocols. A full list of the tests we performed are listed in Table 4.1.

In each test, we recorded the number of elections which began, the number that completed successfully, the amount of time spent working on elections, the amount of time spent in a group, and the mean group size. Using these metrics, we hoped to capture a good representation of what kind effects network problems could have on the stability of the groups formed.

Table 4.1: Tests Performed

Test No.	Test Type	Protocol	Resend Time	Window Size
1	2 Node	SRC	200ms	N/A
2	2 Node	SUC	200ms	8
3	2 Node	SRC	100ms	N/A
4	2 Node	SUC	100ms	8
5	Transient	SRC	200ms	N/A
6	Transient	SUC	200ms	8
7	Transient	SRC	100ms	N/A
8	Transient	SUC	100ms	8

5. RESULTS

Initial results were collected using the experimental platform. As described in the experimental procedure chapter, various system topologies were tested with the described packet loss rates. Tests using the experimental platform were run as many as 40 times. The collected results have been divided into two sections: SRC and SUC, the two delivery protocols used during testing.

The first minute of each test in the experimental test is discarded to remove any transients in the test. The result is that while the tests were run for ten minutes, the maximum result is 9 minutes of in group time.

5.1. SEQUENCED RELIABLE CONNECTION

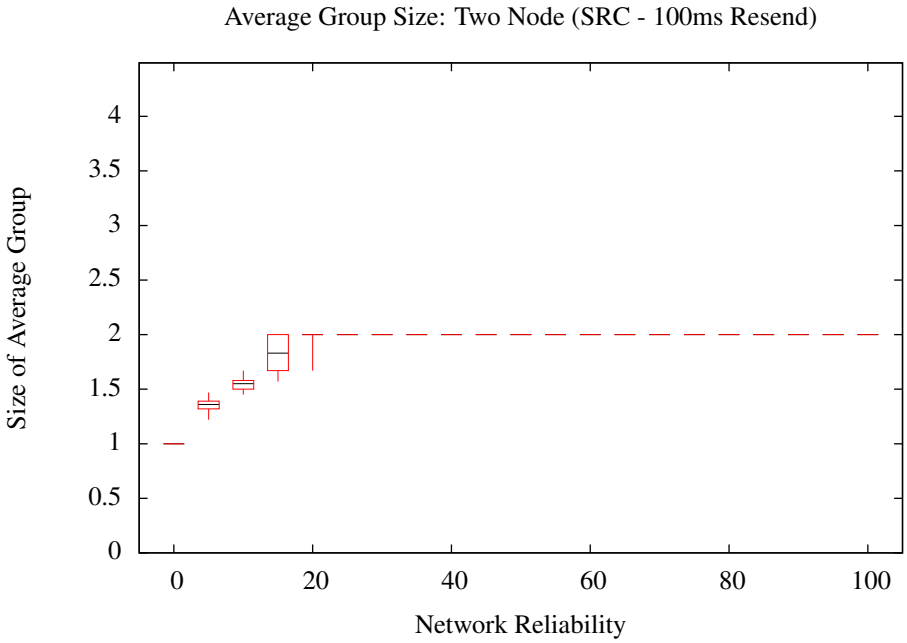

Figure 5.1: Average size of formed groups for two node system with 100ms resend time

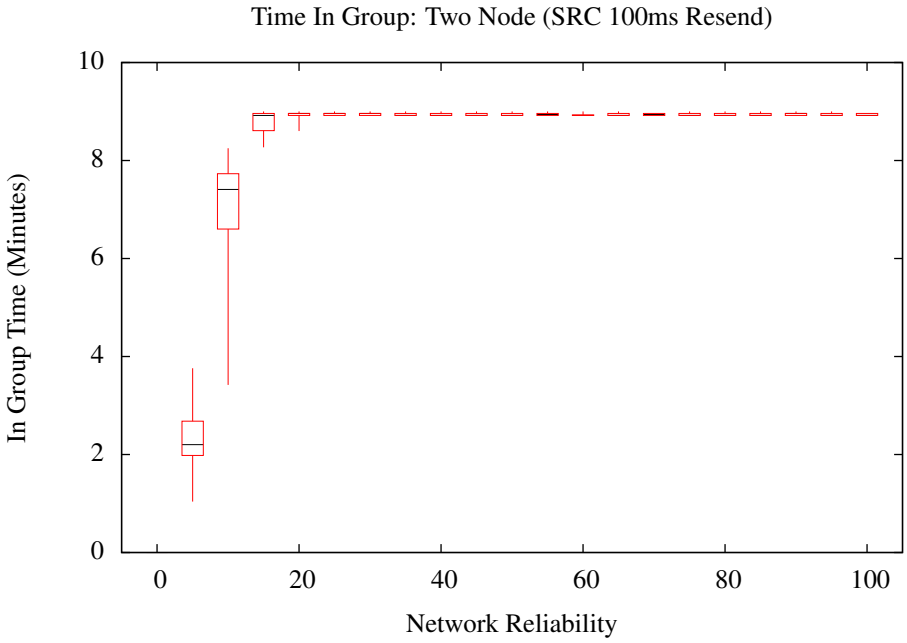

Figure 5.2: Average size of formed groups for two node system with 100ms resend time

5.1.1. Two Node Case. The 100ms resend SRC test with two nodes can be considered a sort of a control. These tests, pictured in Figures 5.1 and 5.2. This test highlights the excellent performance of the SRC protocol, achieving the maximum in group time of 9 minutes with only 15% of datagrams arriving at the reciever. Figure 5.1 shows that when there is no chance of datagrams arriving, the maximum group size is one since no elections can occur. As the reliability increases, more time is spent in a group. Since the maximum group size is 2, it is directly related to the in group time.

Figures 5.3 and 5.4 demonstrates that as the rate at which lost datagrams are resent is decreased to resend every 200ms the time in group falls off. This behavior is expected, since each exchange has a time limit for each message to arrive and the number of attempts is reduced by increasing the resend time.

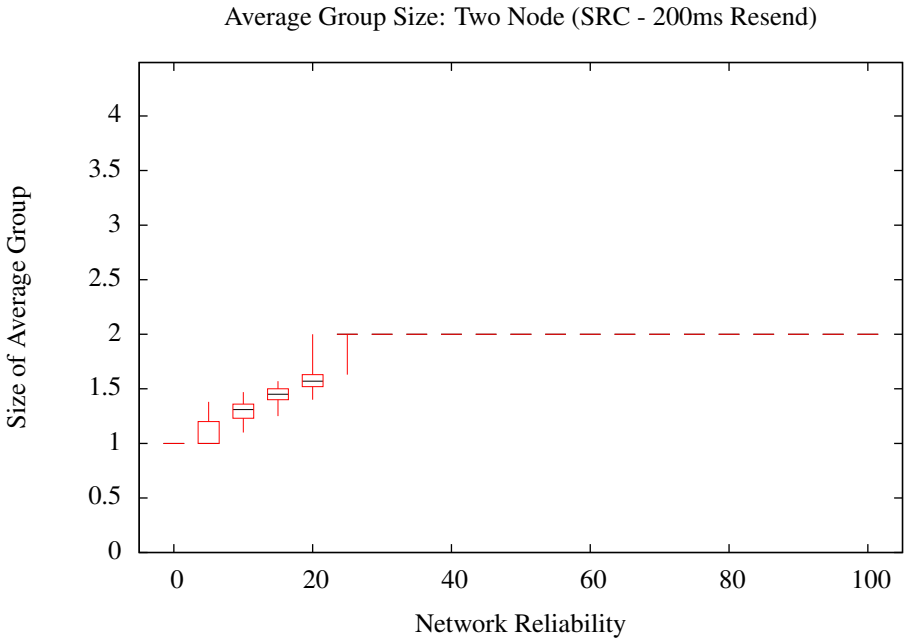

Figure 5.3: Average size of formed groups for two node system with 200ms resend time

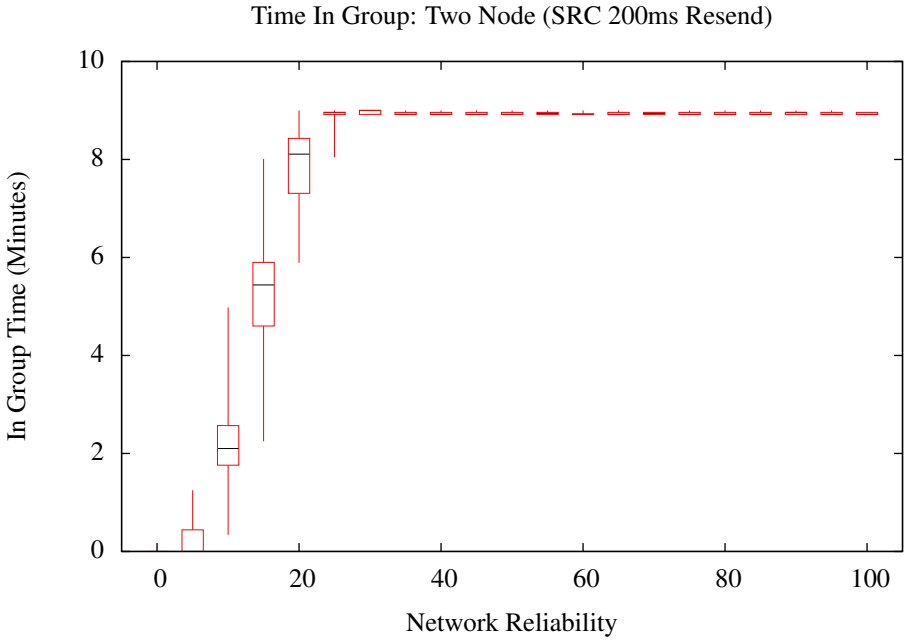

Figure 5.4: Average size of formed groups for two node system with 200ms resend time

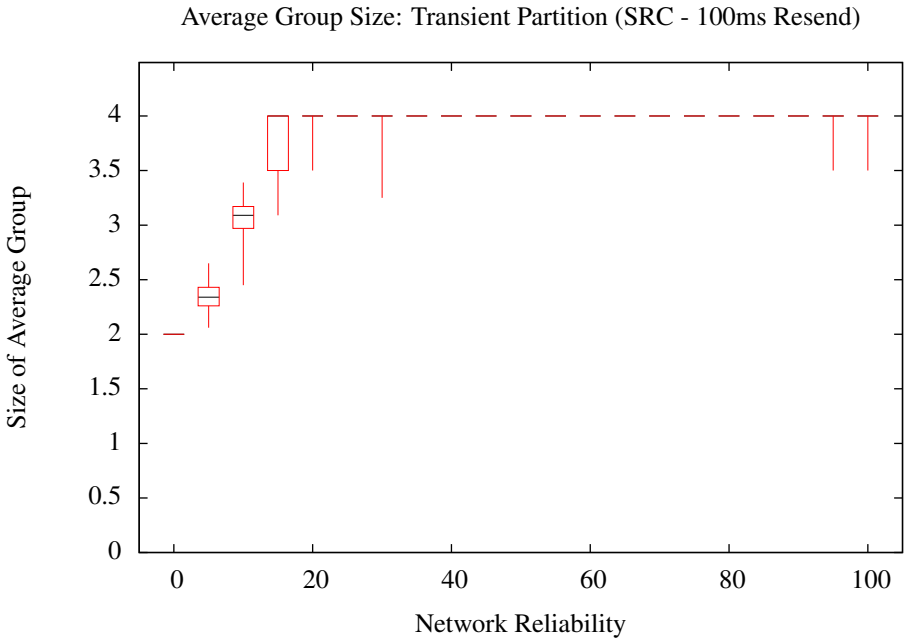

Figure 5.5: Average size of formed groups for the transient partition case with 100ms resend time

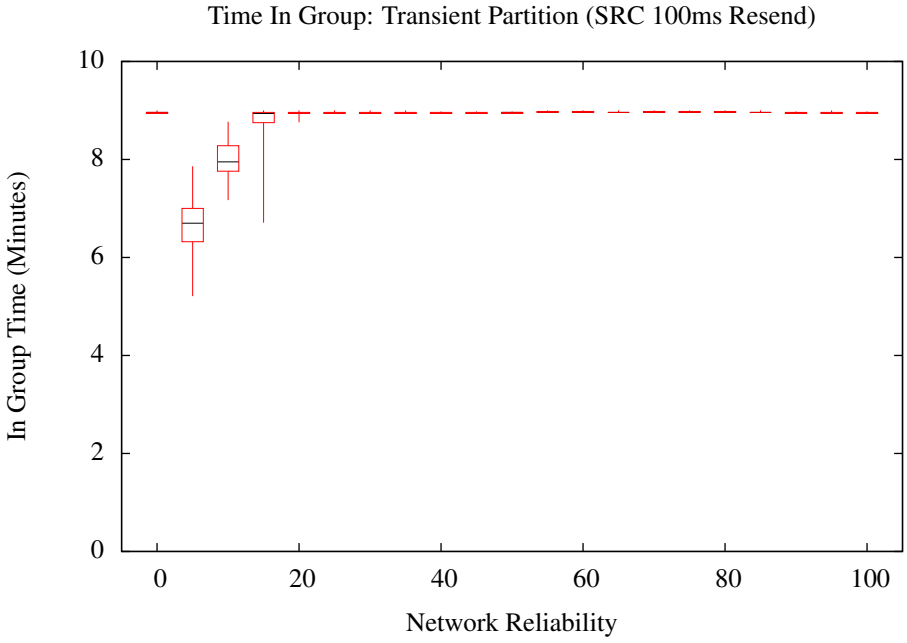

Figure 5.6: Average size of formed groups for the transient partition case with 100ms resend time

5.1.2. Transient Partition Case. The transient partition case, shows a simple example where a network partition seperates two groups of DGI. In the simplest case where the opposite side of the partition is unreachabe, nodes will form a group with the other nodes on the same side of the partition. In our tests, there are two nodes on each side of the partition. In the experiment, the probability of a datagram crossing the partition is increased as the experiment continues. The 100ms case is shown in Figures **??** and **??**.

While messages cannot cross the partition, the DGIs stay in a group with the nodes on the same side of the partition leading to an in group time of 9 minutes, the maximum value. As packets begin to cross the partition (with the reliability increasing), DGI instances on either side begin to attempt to complete elections with the nodes on the opposite partition and the time in group begins to fall. However during this time, the mean group size continues to increase, meaning while the elections are decreasing the amount of time that the module spends in state where it can actively do work, it typically does not fall into a state where it is in a group by itself, which means that most of the lost in group time comes from elections.

The 200ms case, shown in Figures 5.7 and 5.8 displays similiar behavior, with a wider valley due to the limited number of datagrams. It is also worth noting the that the mean group size dips below 2 in the figure, possibly because the longer resend times allow for more race conditions between potential leaders. Discussion of these race conditions is shown in discussed during the SUC charts since it is more prevalant in those experiments.

5.2. SEQUENCED UNRELIABLE CONNECTION

5.2.1. Two Node Case. The SUC protocol's experimental tests show an immediate problem: although there is a general trend of growth in the amount of

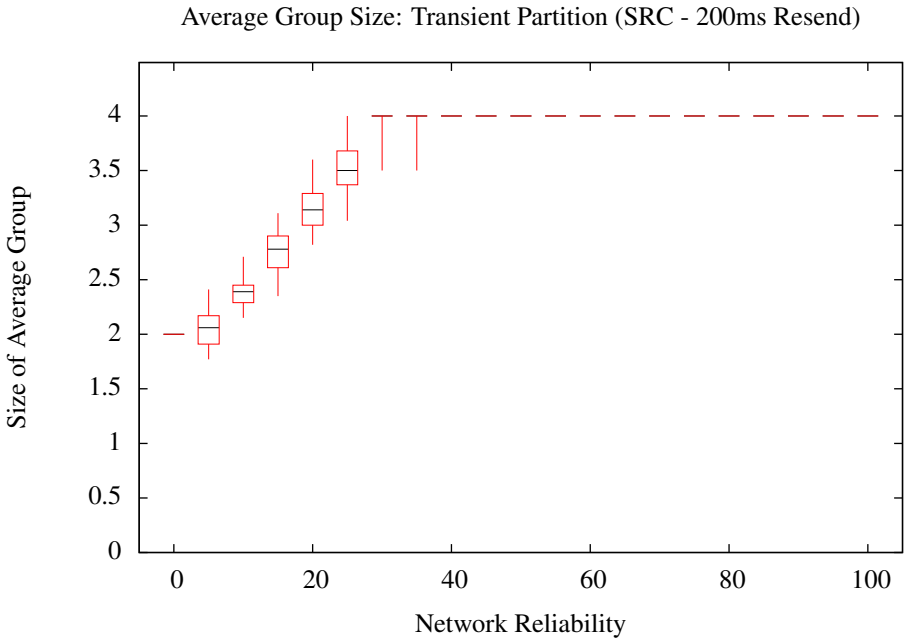

Figure 5.7: Average size of formed groups for the transient partition case with 200ms resend time

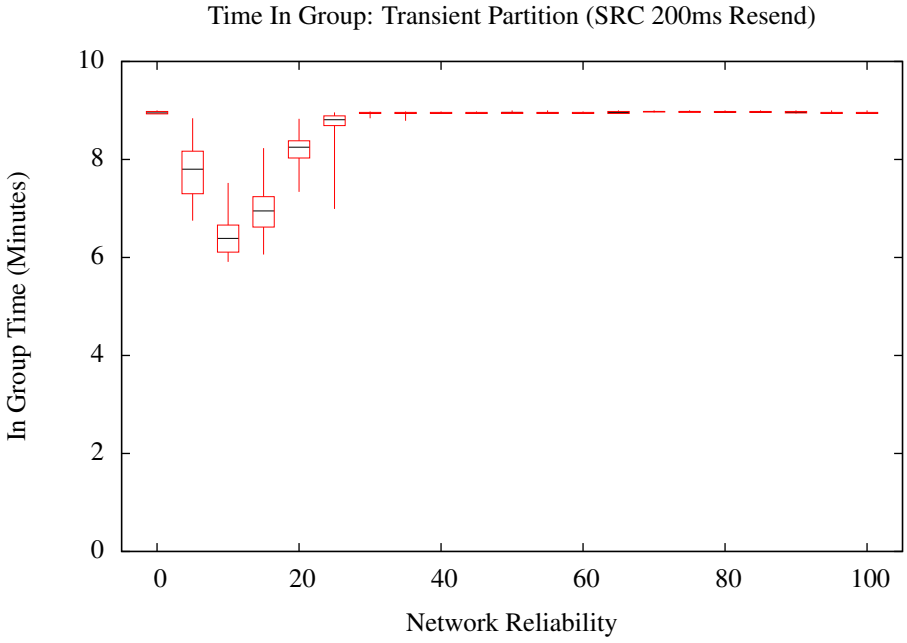

Figure 5.8: Average size of formed groups for the transient partition case with 200ms resend time

time in group and group size charts, shown in Figures 5.9 and 5.10 there is a high degree in variablity for any point collected in the experiments.

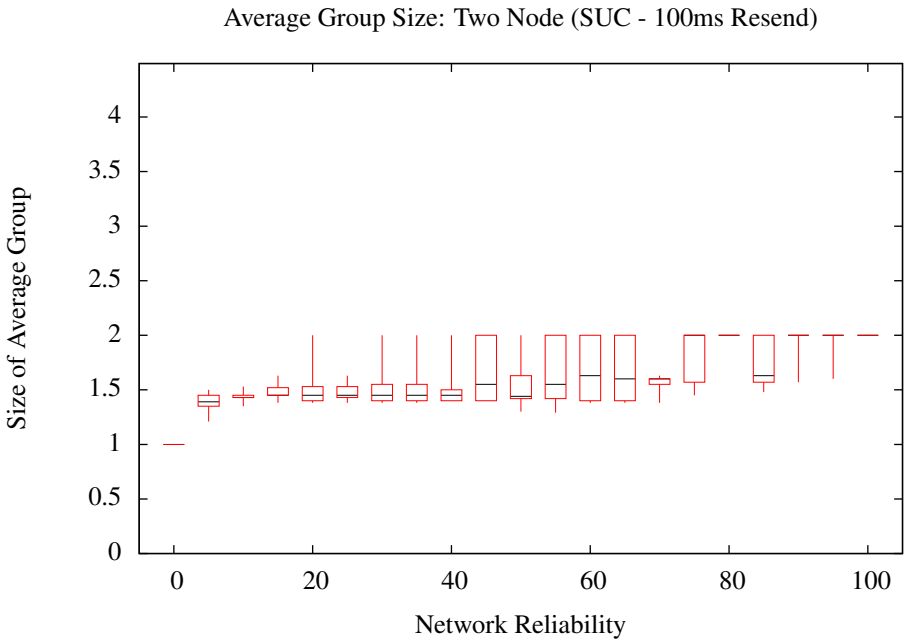

Figure 5.9: Average size of formed groups for two node system with 100ms resend time

In order to identify why these issues occured, the experimental simulator was developed. As stated previously, the simulator emulated the packet loss and behavior of the protocol. The simulator determined the amount of time necessary to complete and election, and the amount of time that group would last after being formed. The simulator allowed more trials of each prescribed reliability and was able to emulate each event at each probability a fixed number of times, unlike the in the experimental system, where rare events (such as elections or group failures) were only collected a few times. The collected data was fed into SharpE.

Using the simulation identified a race condition which is a major factor in the source of the variability in the collected data. As show in Figure MARK the behavior of the algorithm causes the peers to synchronize when there is little to no latency, as there is in the local area network used to perform the test. The result is

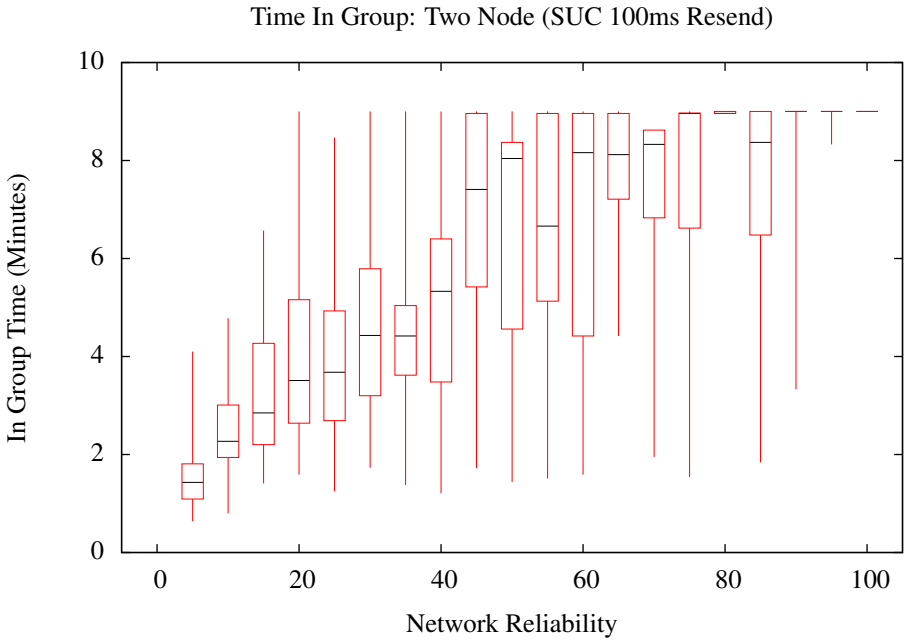

Figure 5.10: Average size of formed groups for two node system with 100ms resend time

that the coordinators and members attempt to exchange their keep-alive messages simultaneously. Because the SUC protocol will stop accepting older sequence numbers when a message is received, race conditions result in a greater number of lost messages. Since both parties are attempting to deliver queries and responses, the queries can be destroyed by the responses. This is a particularly noticeable problem at higher reliability figures. The probability of a message being lost because a newer message supersceedes it is shown in Figure MARK. (Need to make that one)

The simulator is capable of simulations assume the processes have exact synchronization, as well has a user defined offset, represented as the number of ticks for the simulation. A tick of the simulation is the amount of time that will pass in an experimental run between the time a packet is sent and the first time that packet is resent (the resend time). In order to accurately capture the behavoir of a specific

experimental run the offset must be estimated since a changing offset can have a major effect on the characteristics of the system (PROBABLY NEED TO PRODUCE ANOTHER CHART HERE SHOWING HOW OFFSET EFFECTS THE SHAPE OF THE GRAPH).

For the 100ms example, it was estimated that the processes synchronize fairly closely with eachother, and so the simulator was run with a very small offset (approximately 100ms, the smallest the simulator could emulate with one tick) between the two simulated processes. The result, as compared with the experimental value is shown in Figure 5.11.

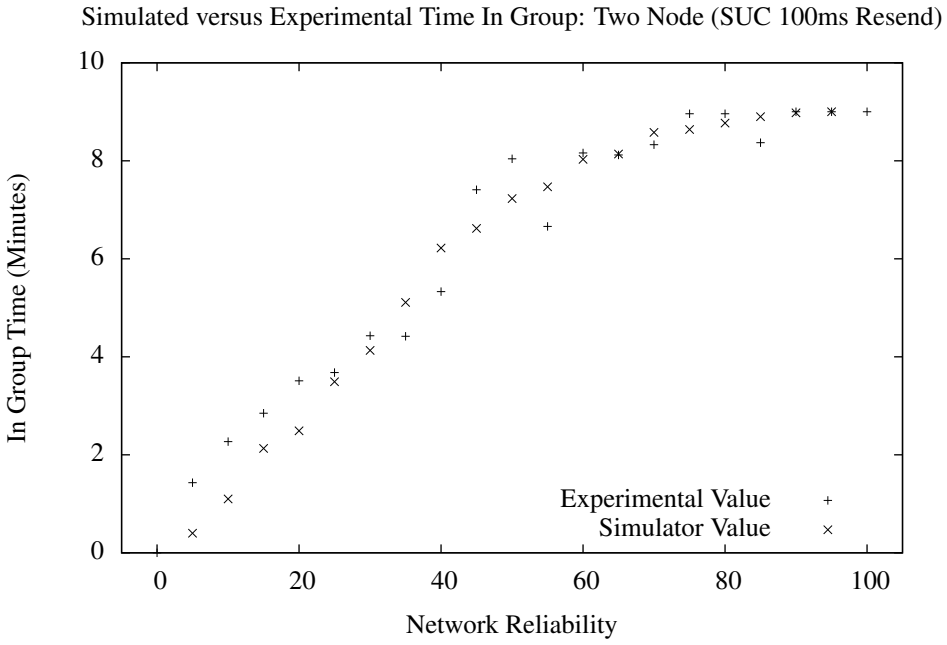

Figure 5.11: Comparison of in group time as collected from the experimental platform and the simulator (1 tick offset between processes).

From figures 5.11 and MARK one can conclude that for the SUC protocol the race condition between processes is a major factor in the amount of time processes can spend in groups. While it is obvious that competition for access to the channel is a detriment to the system, the replication of this competition in software is also a major factor in the stability of groups formed. Furthermore, based on the real-time

design of the DGI, synchronization is a major concern because the real-time execution is designed to cause the processes to synchronize as closely as possible.

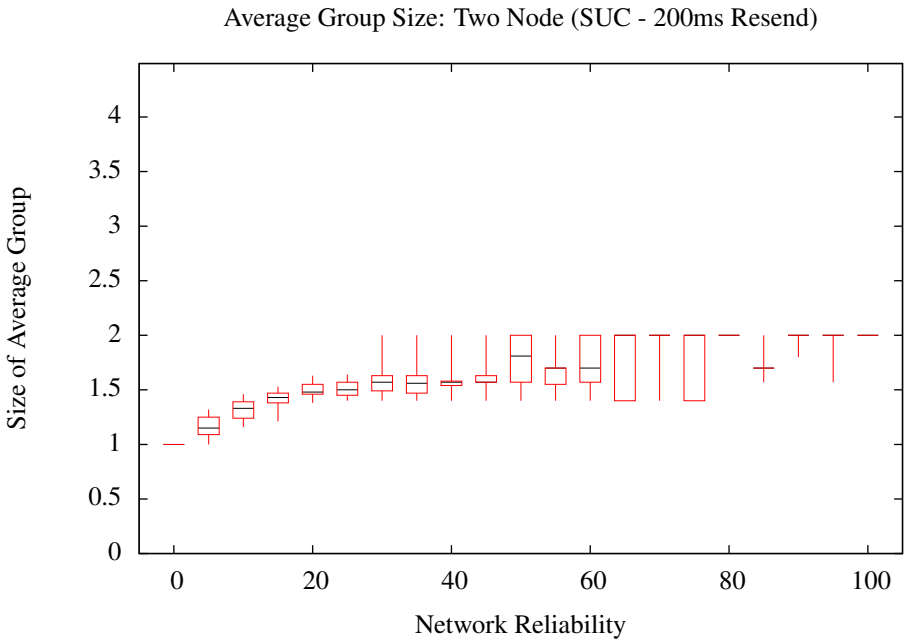

Figure 5.12: Average size of formed groups for two node system with 200ms resend time

In the 200ms resend case, show in Figures 5.12 and 5.13, we can observe a more consistent and growth rate in the in group time as a the reliability increases. In fact, averaging across all the collected data points from the experiment, the average in group time is higher for the 200ms case than it is for the 100ms case (6.86 vs 6.09). Once again, race conditions are the culprit: the collected data when fit to a run of the simulation corresponds to a greater offset in the simulator: approximately 400ms (2 ticks at 200ms), which is two ticks of the simulaton. The comparison between the experimental and simulator values is shown in Figure 5.14.

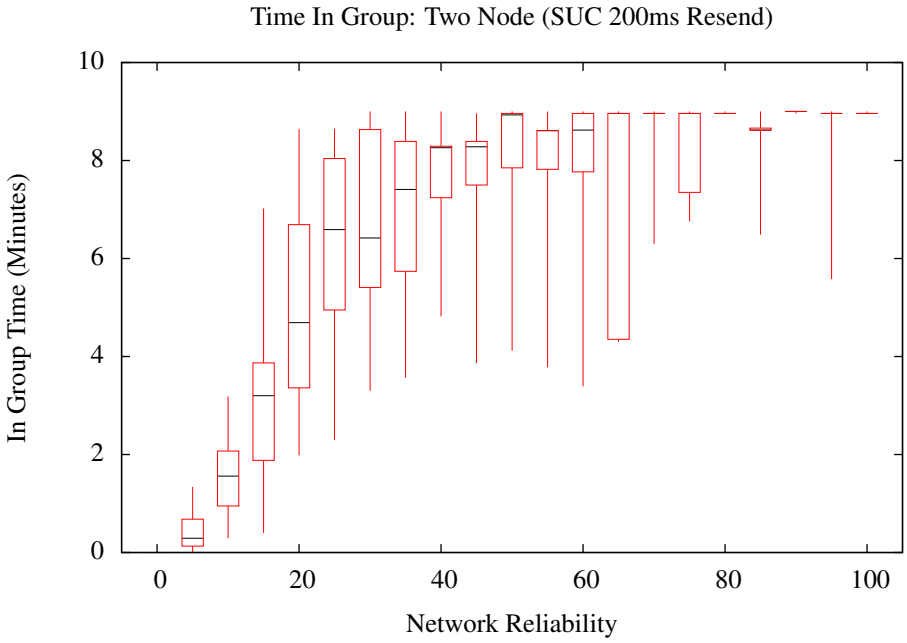

Figure 5.13: Average size of formed groups for two node system with 200ms resend time

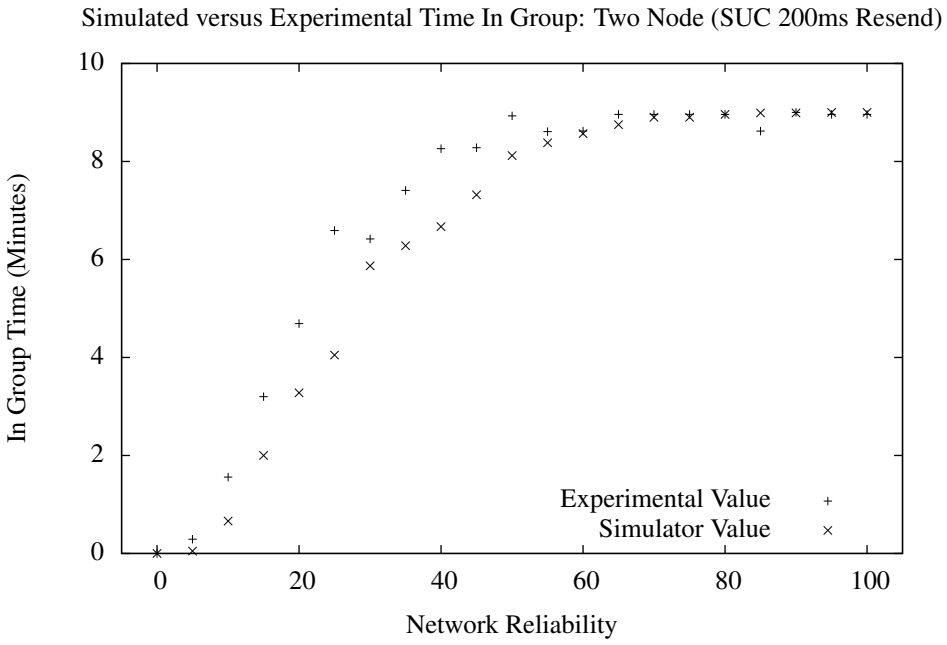

Figure 5.14: Comparison of in group time as collected from the experimental platform and the simulator (2 tick offset between processes).

6. CONCLUSION

In this work we have examined an application of leader election algorithms in a cyber-physical system. We showed that while there are definite benefits and uses for leader election algorithms in cyber-physical systems, they generate a flurry of new problems for system designers to deal with. We have shown that network instability can cause disruptions to the amount of service the cyber system can provide. In our analysis we questioned what the effect of transient partitions and link failures would be on the physical system, especially when the two networks are isomorphic. Our work shows that with the selection of an appropriate protocol under certain failure models, a good quality of service can be achieved in general. The transient partition case, by contrast, creates problems with group stability and is an issue to be investigated further with respect to its effect on CPSs.

BIBLIOGRAPHY

[1] R. Akella, Fanjun Meng, D. Ditch, B. McMillin, and M. Crow. Distributed power balancing for the FREEDM system. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 7 –12, October 2010.

[2] Ziang Zhang and Mo-Yuen Chow. Incremental cost consensus algorithm in a smart grid environment. In *Power and Energy Society General Meeting, 2011 IEEE*, pages 1 –6, July 2011.

[3] C. Singh and A. Sprintson. Reliability assurance of cyber-physical power systems. In *Power and Energy Society General Meeting, 2010 IEEE*, pages 1 –6, July 2010.

[4] Y. Yan, Y. Qian, H. Sharif, and D. Tipper. A survey on smart grid communication infrastructures: Motivations, requirements and challenges. *Communications Surveys Tutorials, IEEE*, PP(99):1 –16, 2012.

[5] Manitoba HVDC Research Centre. Pscad.com, May 2012. `http://pscad.com`.

[6] RTDS Technologies. Power systems simulator software: Rscad software suite, May 2012. `http://www.rtds.com/software/rscad/rscad.html`.

[7] H. Garcia-Molina. Elections in a distributed computing system. *Computers, IEEE Transactions on*, C-31(1):48 –59, January 1982.

[8] B. Dawes, D. Abrahams, and R. Rivera. Boost c++ libraries, May 2012. `http://www.boost.org`.

[9] C. V. Hansen and J. E. Forcier. Fabric, May 2012. `http://fabfile.org`.

VITA