APPLICATIONS FOR GROUP MANAGEMENT TO MANAGE CPS STABILITY

by

STEPHEN JACKSON

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

2014

Approved by

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

ABSTRACT

APPLICATIONS FOR GROUP MANAGEMENT TO MANAGE CPS STABILITY

by

STEPHEN JACKSON

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

2014

Approved by

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

Abstract

Cyber-physical systems (CPS) are an attractive option for future development of critical infrastructure systems. By supplementing the traditional physical network with cyber control, the performance and reliability of the system can be increased. In some of these networks, distributing the cyber control offers increased redundancy and availability during fault conditions. However, there are very few works which study the effects of cyber faults on a distributed cyber-physical system. These are of a particular interest in the Smart Grid environment where outages and failures are very costly. By examining the behavior of a distributed system under fault scenarios, the overall robustness of the system can be improved by planning characteristics and responses to faults that allow the system to continue operating in difficult circumstances.

This work examines the consequences of network unreliability on a core part of the Distributed Grid Intelligence (DGI) for the FREEDM (Future Renewable Electric Energy Delivery and Management) Project. By applying 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 identify the amount of time spent in a group, working, as a function of the network reliability. Given this one can parameterize the amount of messages that are lost and thus the number of failed physical actuations. Knowing this and the physical characteristics of the system, one can tune the amount of time between reconfiguration in order to prevent the number of failed physical changes from causing the system to become unstable.

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

Figure Page

LIST OF TABLES

1. INTRODUCTION

In the smart grid domain, leader elections are an attractive option for autonomously configuring cyber components. Proposed algorithms such as [1] and [2] are distributed algorithms for managing power in a smart grid rely on an assumption that a group of peers will be able to coordinate together. 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.

FREEDM (Future Renewable Electric Energy Delivery and Management) System is a Smart Grid project focused on the future of the electrical grid. Major proposed features of the FREEDM network include the Solid State Transformer, distributed local energy storage, and distributed local energy generation[3]. This vein

of research emphasizes decentralizing the power grid: making it more reliable by distributing energy production resources. Part of this design requires the system to operate in islanded mode, where portions of the distribution network are segmented from each other. However, there is a major shortage of work within the realm of the effects cyber outages have on CPSs [4] [5]. Addtitionally, research that has been done such as [6] indicate that cyber faults can cause a physical system to apply unstable settings.

This work presents the initial steps to better understanding and planning for these faults. By taking a new approach to considering how a distributed system interacts during a fault condition, new techniques for managing a fault scenario in a cyber-physical systems will be created. To do this, we present an approach in modeling the grouping behavior of a system using Markov chains. These chains produce expectations of how long a system can be expected to stay in a particular state, or how much time it will be able to spend coordinating and doing useful work over a period of time. Using these measures, the behavior of the control system for the physical devices can be adjusted to prevent faults.

2. RELATED WORK

2.1. VIRTUAL SYNCHRONY

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

Consider a distributed system where events can be causally related based on their execution on the local processors and communication between processes, as defined in [7, p .101]. Virtually synchronous systems use multicast to communicate reliably between processes.

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 $e \rightarrow e'$

[7, p .101]

This defines a dependence and causality relation between events in the system. If two events cannot be causally 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 synchronous 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 differentiated from a trace where tasks are executed simultaneously. A pair of histories H and H' are equivalent if for each process in the system (noted as p) $H|_p$ cannot be differentiated $H'|_p$ based on the casual relationships between the events in the history. [7, p .103]

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. [7, p .104]

2.1.1. Process Groups. Virtual synchrony models also support process groups. Although each implementation of a virtually synchronous 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 referred to as a view. A view is a collection of processes that are virtually synchronous with each other. 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 occurred 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 delivers 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.

[7, p .103]

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.

[7, p .102]

As part of the failure model, it is worth noting two commonly used multicast delivery guarantees: uniform and non-uniform. The uniform property 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[8] 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 [8] 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 causally 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.

[8]

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 causality 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 [7] and Horus [9] 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 reliable 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 guaranteed to be received 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 limitation.

2.4. TRANSIS (1992)

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

Transis [10] 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 exchange 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 adherence 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[11] 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

Spread[12] is a modern distributed framework. It is available as a library for a variety of languages including Java, Python and C++. Spread is designed as a series of daemons which communicate over a wide area network. Processes on the same LAN as the daemons connect to the daemon directly. This is analogous the gateways in the other distributed toolkits, however the daemon is a special, dedicated message passing process, rather than a participating peer with a special role.

Using a dedicated daemon allows a spread configuration to reconfigure less frequently when a process stops responding (which normally correlates to a view

change) since the daemons can insert a message informing other processes that a process has left while still maintaining the message ordering without disruption. Major reconfigurations only need to occur when a daemon is suspected of failing.

Spread implements an extended virtual synchrony model. However, message ordering is performed at the daemon level, rather than at the group level. Total ordering is done using a token passing scheme among the daemons.

2.7. FAILURE DETECTORS

Failure detectors [13] (Sometimes referred to as unreliable failure detectors) are special class of processes in a distributed system that detect other failed processes. Distributed systems use failure detection to identify failed processes for group management routines. Because it isn't possible to directly detect a failed process in an asynchronous system, there has been a wide breath of work related to different classifications of failure detectors, with different properties. Some of the properties include:

- Strong Completeness - Every faulty process is eventually suspected by every other working process.
- Weak Completeness - Every faulty process is eventually suspected by some other working process.
- Strong Accuracy - No process is suspected before it actually fails.
- Weak Accuracy - There exists some process is never suspected of failure.
- Eventual Strong Accuracy - There is an initial period where strong accuracy is not kept. Eventually, working processes are identified as such, and are not suspected unless they actually fail.

- Eventual Weak Accuracy - There is an initial period where weak accuracy is not kept. Eventually, working processes are identified as such, and there is some process that is never suspected of failing again.

[13]

One class of failure detectors, Omega class Failure detectors, are particularly interesting because of [14]. An eventual weak failure (weak completeness and eventual weak accuracy) detector is the weakest detector which can still solve consensus. It is denoted several ways in various works including $\diamond\mathcal{W}$ [13], $\mathcal{W}$ [15] [16] and Ω (Omega) [14].

[14] studies an Omega class failure detector using OmNet++, a network simulation software. Instead of omission failures, however, it considers crash failures. Each configuration goes through a predefined sequence of crash failures. OmNet is used to count the number of messages sent by each of three different leader election algorithms. Additionally, [14] only considers the system to be in a complete and active state when all participants have consensus on a single leader.

3. BACKGROUND THEORY

3.1. FREEDM DGI

In this work we model the group management module of the FREEDM DGI (Distributed Grid Intelligence). The DGI (Distributed Grid Intelligence) 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. These algorithms work in concert to move power from areas of excess supply to excess demand.

The DGI uses several modules to manage a distributed smart-grid system. Group management, the focus of this work, implements a leader election algorithm to discover which nodes are reachable in the cyber domain.

Other modules provide additional functionality such as collecting global snapshots, and a module that negotiates the migrations and gives commands to physical components.

The DGI is a real-time system: certain actions (and reactions) involving power system components need to be completed with a pre-specified time-frame to keep the system stable. The DGI uses a round robin scheduler: each module is given a predetermined window of execution which it may use to perform its duties. When a module's time period expires, the next module in the line is allowed to execute.

3.2. BROKER ARCHITECTURE

The DGI software is designed around a broker architectural specification. Each core functionality of the system is implemented within a module that is provided access to core interfaces which deliver functionality such as scheduling requests, message passing, and a framework to manipulate physical devices.

The Broker provides a common message passing interface that all modules are allowed to access. Information is passed between modules using this message passing interface. For example, the list of peers in the group is made available to other modules with a message.

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. UDP also allows development of protocols with various properties to evaluate which properties are desirable. To accomplish this lightweight protocols which are best effort oriented were implemented to deliver messages as quickly as possible within the requirements.

The decision to go with a lighter weight protocol was also influenced by the FREEDM center targeting lower cost, less powerful implementation platforms, with less available computing resources than a traditional server or desktop. Furthermore,

the protocols listed here continue operating despite omission failures: they follow the assumption that not every message is critical to the operation of the DGI and that the channel does not need to halt entirely to deliver one of the messages.

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 delivered 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 acknowledgement for. There is a Resend() function is periodically called to attempt to redeliver lost messages to the receiver.

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 acknowledgement. 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 included in Appendix REF

3.2.3. 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.

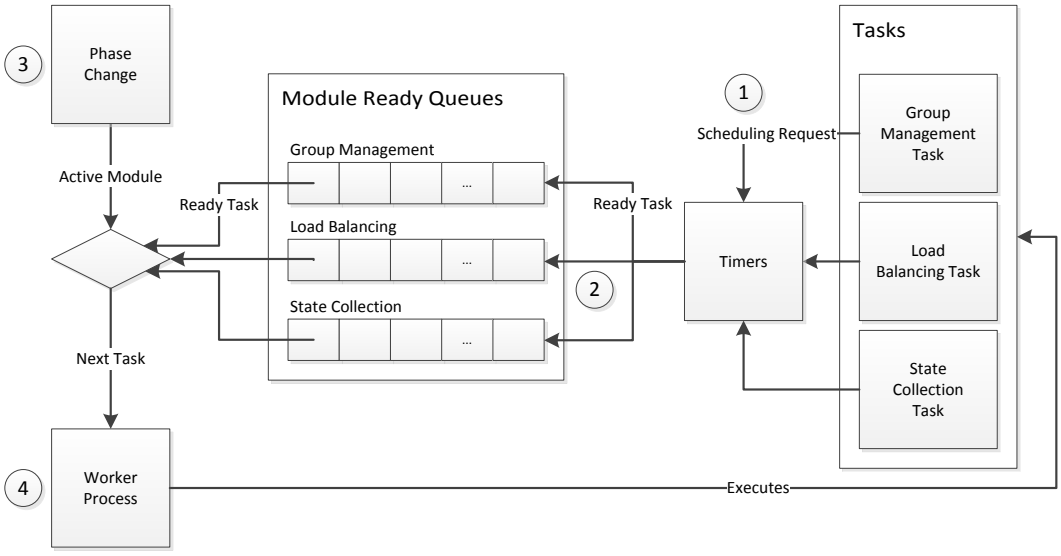

Figure 3.1: The real time scheduler uses a round robin approach to allot execution time to modules.

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.

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 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.

3.3. GROUP MANAGEMENT ALGORITHM

The DGI uses the leader election algorithm, "Invitation Election Algorithm" written by Garcia-Molina in [17]. Originally published in 1982, 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.

Since Garcia-Molina's original publication, there has been a large body of work creating various election algorithms. Each algorithm is designed to be well suited to the circumstances it will deployed in: there are specialized algorithms for wireless sensor networks[18][19], detecting failures in certain circumstances[20][21], and of course, transient partitions. Work on leader elections has been incorporated into a variety of distributed frameworks: Isis[7], Horus[9], Totem[11], Transis[10], and Spread[12] all have methods for creating groups. Despite this wide body of work, the fundamentals of leader election are consistent across all work: nodes arrive at a consensus of a single peer who coordinates the group, and nodes the fail are detected and removed from the group.

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 failure detector [14]. 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 separate roles to play in the failure detection process. Leaders use a periodic search to locate other leaders in order to merge groups. 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 leader's group. The member 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, but it is not implemented in DGI code.

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

In this work it is assumed that a leader does not span two partitioned 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. 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 transitions between these states. Each transition has some probability attached to them.

Since the system, taken as a whole, can be reasonably modeled as a collection of states each describing the state or configuration of the system, and that the transitions between those state (failure events or election events) are probabilistic, rather than deterministic, it is a natural extension to model the distributed system as a Markov Chain.

3.5.1. Continuous Time Markov Chains. 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.

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

Additionally, the models we present are time homogeneous, meaning that the time that transition probabilities 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 continuous 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 continuous 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. Continuous 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 distribution 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 is:

$$\sum \lambda(x, y) = \sum \lambda p_{x,y} = \lambda(x) \tag{3.5}$$

Where lambda(x,y) is the expected amount of time before state x transitions to state y. Interestingly, the expected time in a state ($\lambda(x)$) is related to a the expected time for an individual transition ($\lambda(x, y)$) by a probability $p_{x,y}$.

Each transition lends to an expected amount of time expected in the state. Then, to do a random walk of a continuous 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.

3.5.2. Assumptions. In order to model the system, we assumed that the time between events was exponentially distributed. Furthermore, we assumed that the system would be fairly well synchronized, with most elections occurring at the same time. This assumption was valid for 2-Node cases of our non-real-time code, but was a major issue as the number of nodes began to increase. However, with of the use of the round-robin scheduler with synchronization to enforce our real-time requirements, assuming the synchronization of processes is not a major leap.

All participating peers are assumed to be on the same schedule: all peers begin execution of a model simultaneously. This is accomplished using [22]. This work assumes that the clocks are synchronized: even if the network has faulted, process

clocks have not drifted noticeably from their last synchronization. Additionally, a production system would likely use GPS time synchronization in order to take certain power system readings [23].

3.5.3. Constructing The Markov Chain. 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. Processes in the system occasionally exchange messages to determine if the other processes have crashed, and to discover new leaders.

The DGI can perform work assuming that it is in a group, and not in an election state (since the group management module instructs other modules to stop during an election). The collected data in the previous sections is based on that assumption, and the Markov chains that models those scenarios needed to as well.

Processes in the DGI are either members or leaders. Leaders are processes which have won elections among its members.

As stated previously, it was assumed that the events in the distributed system were distributed exponentially. Events are modeled in the chain using $\lambda(x)$ which is the parameter of the exponential distribution. It is important to note that:

$$\mathrm{E}[X] = \frac{1}{\lambda}. \tag{3.6}$$

and

$$\lambda(x) = \sum \lambda(x, y) = \sum \lambda(x) p(x, y) \tag{3.7}$$

Where $\lambda(x)$ is the exponential parameter for the total time spend in a state x, $\lambda(x, y)$ is the exponential parameter for a transition from state x to state y, and $p(x, y)$ is the normally distributed probability that a state transitions from state x to state y.

Failure Detection. 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.

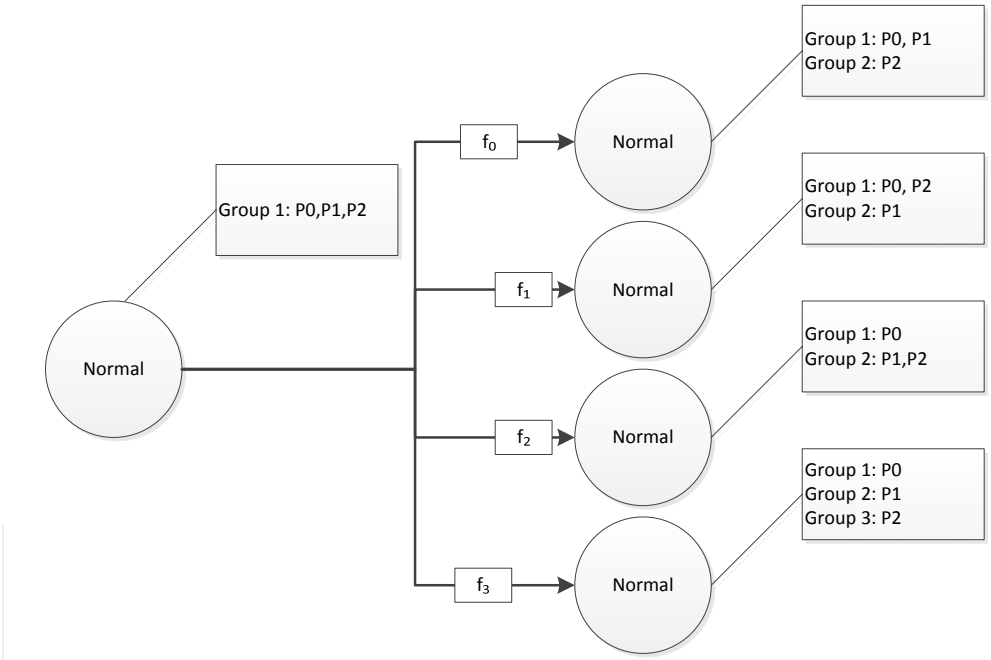

Figure 3.2: A diagram showing a partial Markov chain for failure detection

Members cannot suspect other processes of being crashed, only the leader can identify failed group members.

A model of a failure detection stage of the leader election algorithm is presented in Figure 3.2. A set of nodes begin in a normal state as part of a group. The leader sends a query to every member, and every member sends a query to the leader. If a response is not received in either direction, the process is considered to be unreachable and is either ejected from the group by the leader (if the query originated from the leader) or the member leaves the group and becomes a coordinator themselves.

The system will stay in the original state as long as all nodes complete their queries and responses. Let T_R be the amount of time allowed for a response, T_C be the time between discovery attempts, and p_F is the probability that at least one peer

fails to complete the exchange. Based on this, the expected amount of time in the grouped state (T_G) is:

$$\begin{cases} T_G = (T_R + T_C)/p_F & p_F > 0 \\ \infty & p_F = 0 \end{cases} \tag{3.8}$$

Let δ equal exponential parameter of the exponential distribution for the base state. Then we can relate the probabilities of each possible transition to the parameter for the base state. Let p_i be the probability of transitioning to configuration i after leaving the base state and let f_i be the exponential parameter for the transition to an individual configuration:

$$\delta = \sum f_i = \sum \delta p_i = \frac{1}{T_G} \tag{3.9}$$

Leader Election. 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.

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 flow network and the cyber communication network.

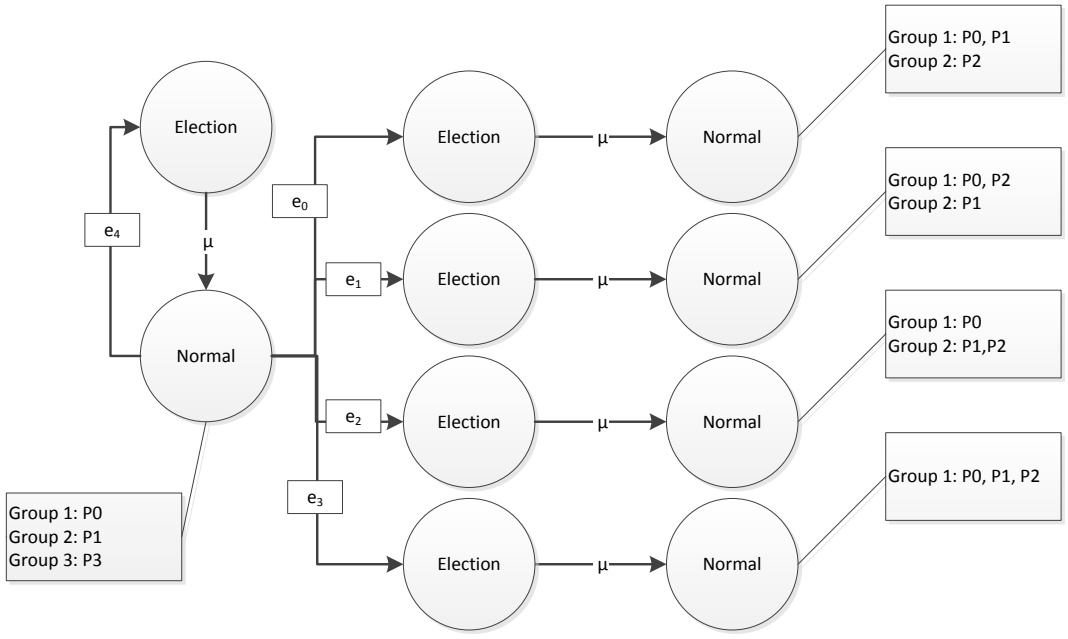

Figure 3.3: A diagram showing a partial Markov chain for an election

A continuous time Markov model of a single election is presented in Figure 3.3. A set of leaders begin in a normal state. After some time T_D an "are you coordinator" message discovers some other peer. T_D is a function of the number of discovery checks which discover no leaders (which in turn is a function of the link reliability). Let T_R be the amount of time allowed for a response, T_C be the time between discovery attempts, and p_D is the probability that the exchange discovers a leader.

$$\begin{cases} T_D = (T_R + T_C)/p_D & p_D > 0 \\ \infty & p_D = 0 \end{cases} \tag{3.10}$$

Then, the parameters e_x in Figure 3.3 are a function of T_D and p_x, the probability an election results in configuration x.

$$e_x = \frac{p_x}{T_D} \tag{3.11}$$

Once a leader has been discovered, the system transitions into an election state, based on the potential outcome, where the peers hold an election to determine a new configuration. As shown in Figure 3.3, an election can either succeed or fail, resulting in a new system configuration, or each involved process to return to the single member group state.

The amount of time that an election takes is fixed before the algorithm is executed. Let T_E be the mean time it takes to complete any election. Therefore:

$$\mu = \frac{1}{T_E} \tag{3.12}$$

Combined Model. A combined model combines election and failure detection Markov chain components. Except for the states where all reachable nodes are in the same group and the states where there are no reachable leaders each state has a combination of election transitions and failure transitions. The combined model is predictive of the overall characteristics of the system. The time spent in a particular configuration is a function of the λ's of all the events that can cause the system to transition away from a configuration.

To construct the Markov chain, simulations of individual events are performed. The circumstances for the events are assumed to be homogeneous: processes only differ by their process id. Using this assumption, the simulation of events can be broken

down into a series of scenarios that are representative of the events in the system. Since each scenario is independent of other scenarios, each scenario can be run independently. Additionally, since the circumstances are assumed to be homogeneous, scenarios that are similar, such as ones where two processes swap roles can be simulated only once, and the results can be transformed from one scenario to another with a simple mapping. This mapping scheme and parallizability helps keep the state space explosion of the potential states under control.

4. RESULTS

4.1. INITIAL RESULTS

Initial data was collected from a non-real time version of the DGI code. For each selected message arrival chance, as many as forty tests were run. The collected results from the tests are divided into several target scenarios as well as the protocol used.

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. These graphs first appeared in [24].

4.1.1. Sequenced Reliable Connection.

Two Node Case. The 100ms resend SRC test with two nodes can be considered a sort of a control. These tests, pictured in Figure 4.1, highlights the performance of the SRC protocol, achieving the maximum in-group time of 9 minutes with only 15% of datagrams arriving at the receiver.

Figures 4.2 demonstrates that as the rate at which lost datagrams are resent is decreased to 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.

Transient Partition Case. The transient partition case shows a simple example where a network partition separates two groups of DGI processes. In the simplest case where the opposite side of the partition is unreachable, 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

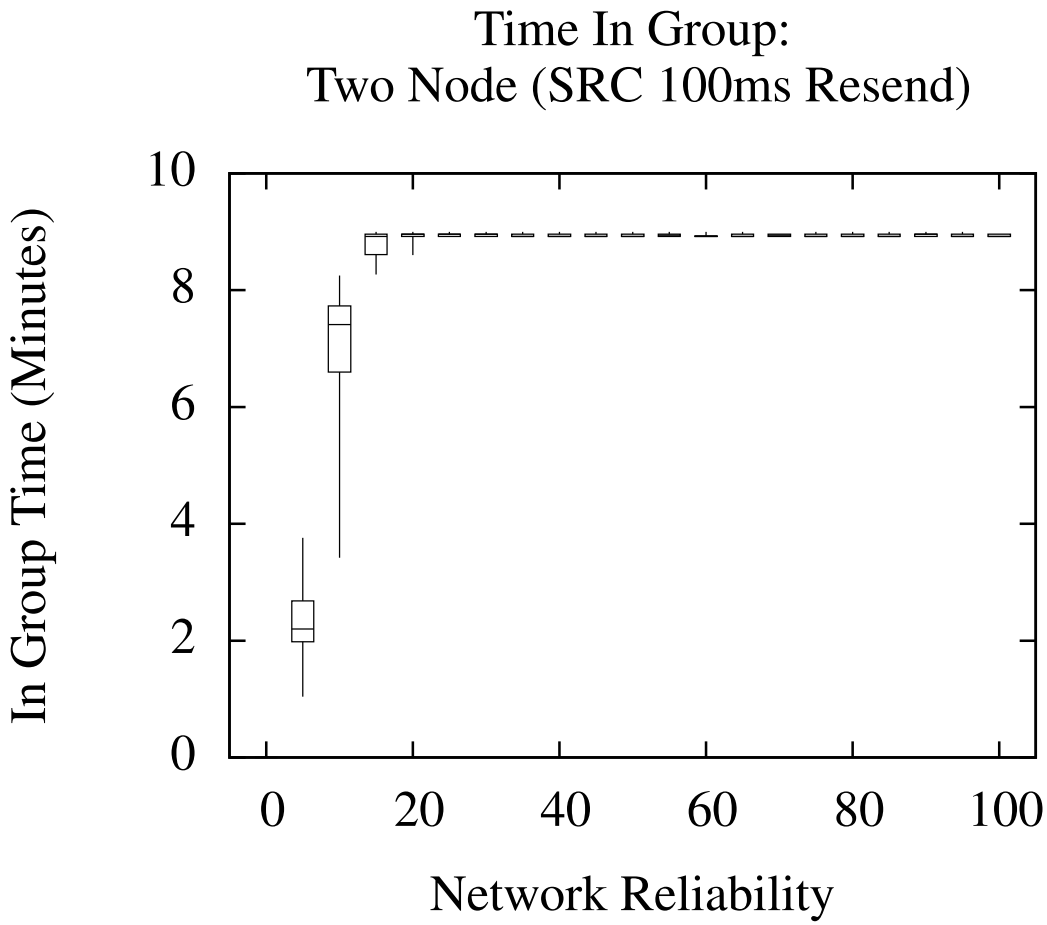

Figure 4.1: Time in group over a 10 minute run for two node system with 100ms resend time

datagram crossing the partition is increased as the experiment continues. The 100ms case is shown in Figures 4.3 and 4.4.

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 (as the reliability increases), 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

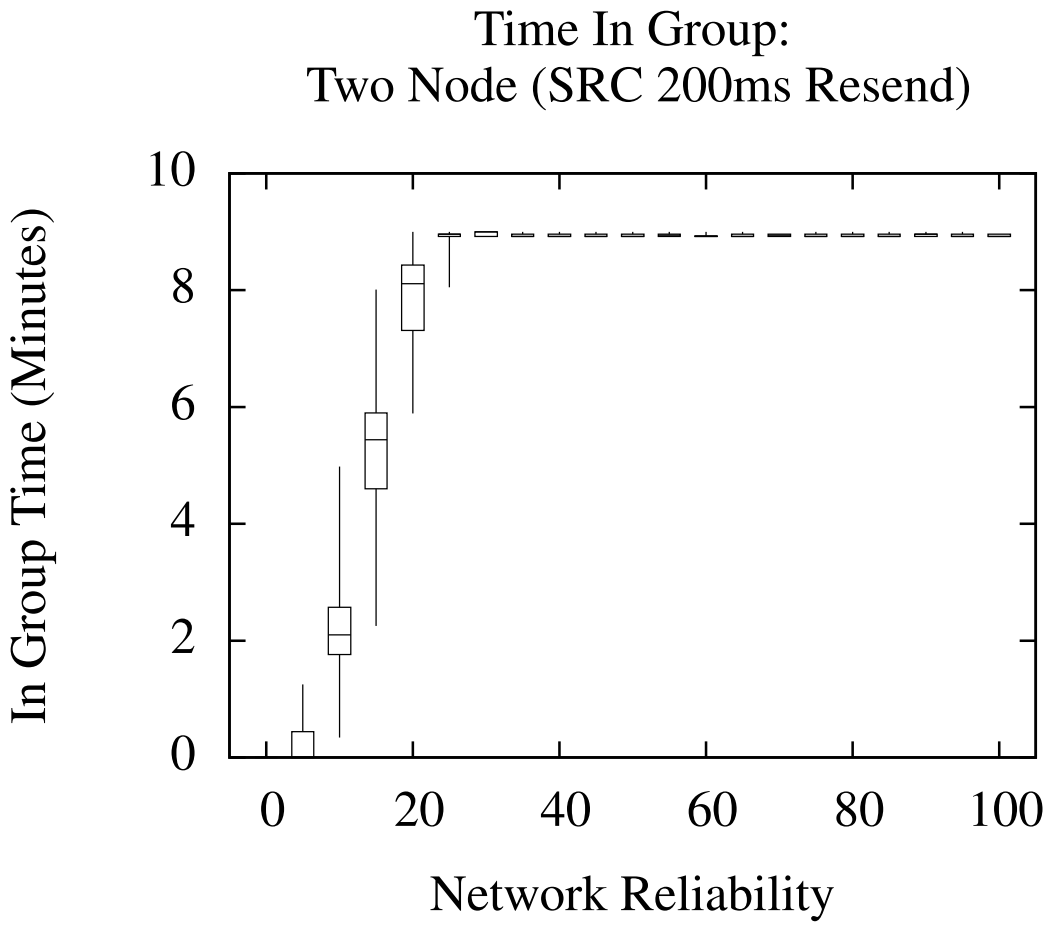

Figure 4.2: Time in group over a 10 minute run for two node system with 200ms resend time

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 4.5 and 4.6 displays similar 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

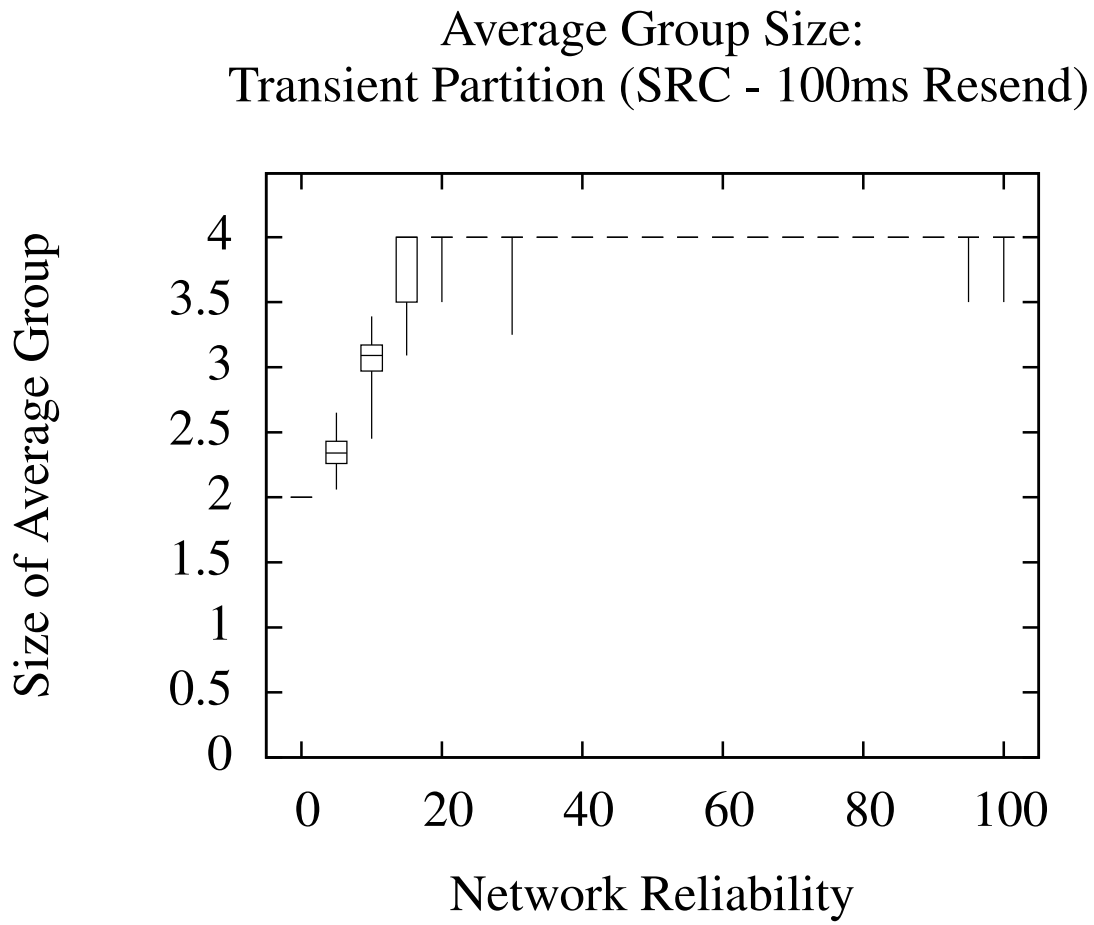

Figure 4.3: Average size of formed groups for the transient partition case with 100ms resend time

race conditions is shown in discussed during the SUC charts since it is more prevalent in those experiments.

4.1.2. Sequenced Unreliable Connection.

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 time in group and group size charts, shown in Figure 4.7 there is a high amount of variance for any particular trial.

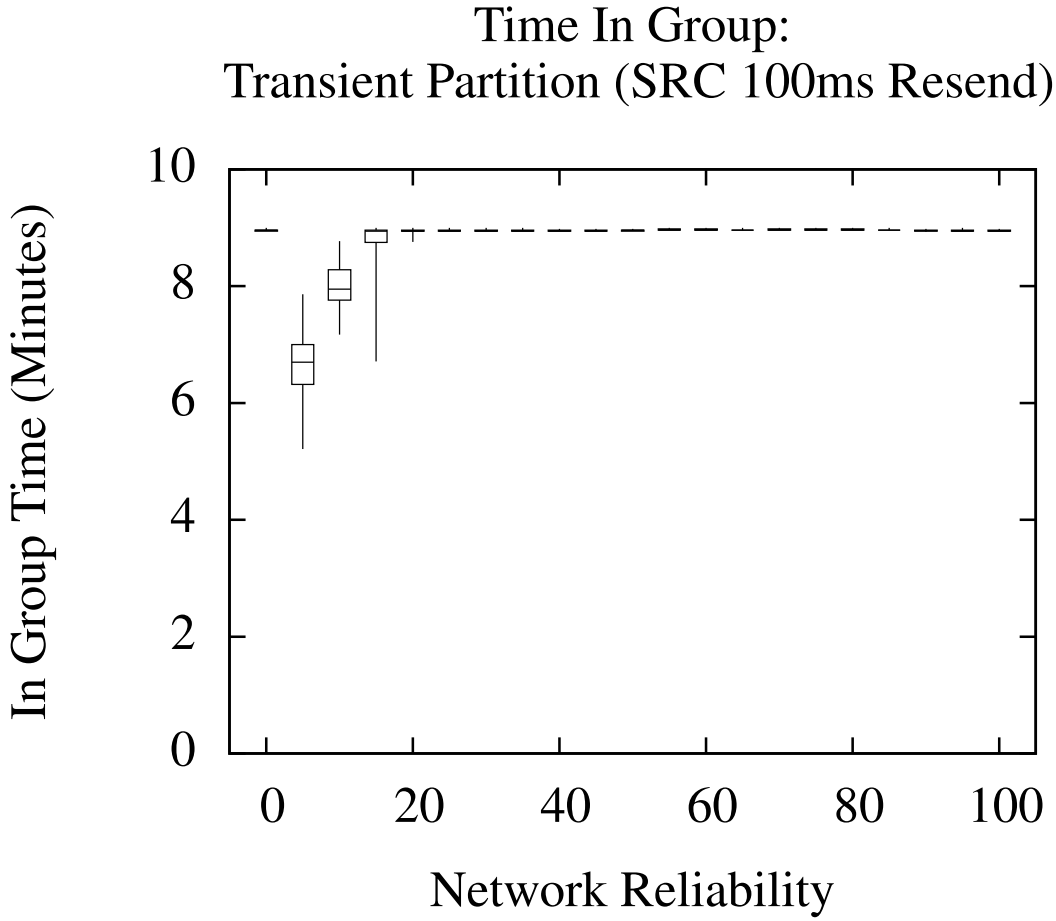

Figure 4.4: Time in group over a 10 minute run for the transient partition case with 100ms resend time

In the 200ms resend case, show in Figure 4.8, it can be observed that there is a more 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). However, due to the large amount of variance in the collected in-group time, it is not possible to state with confidence that the there is a significant difference between the two cases.

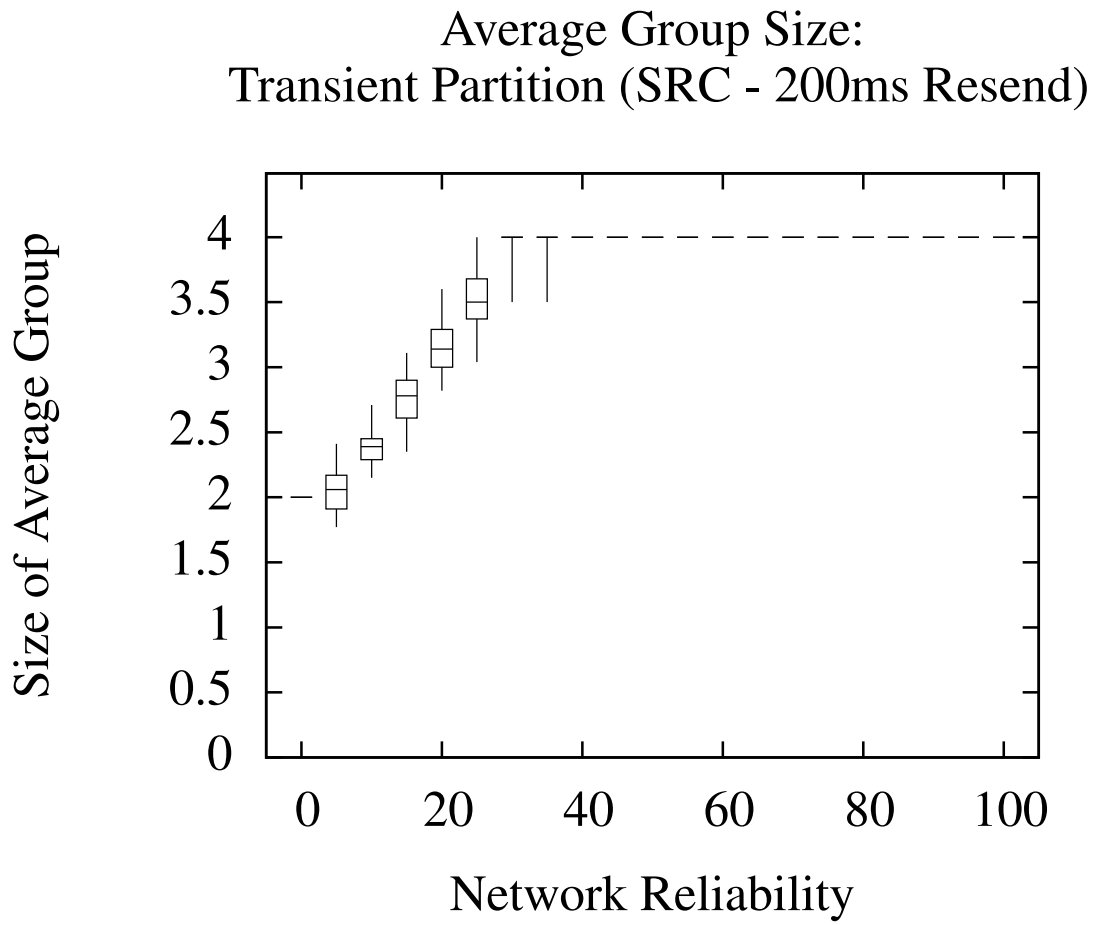

Figure 4.5: Average size of formed groups for the transient partition case with 200ms resend time

4.2. MARKOV MODELS

Due to the high amount of variance in the collected data, and the resulting difficulty making any sort of prediction about other systems from the data, a more formal approach was tried.

4.2.1. Initial Model Calibration. The presented methodology of constructing the model was initially calibrated against the original two-node case, using a non-real-time version of the DGI codebase. The resulting Markov chain was processed using SharpE [25][26] made by Dr. Kishor Trivedi's group at Duke University, a

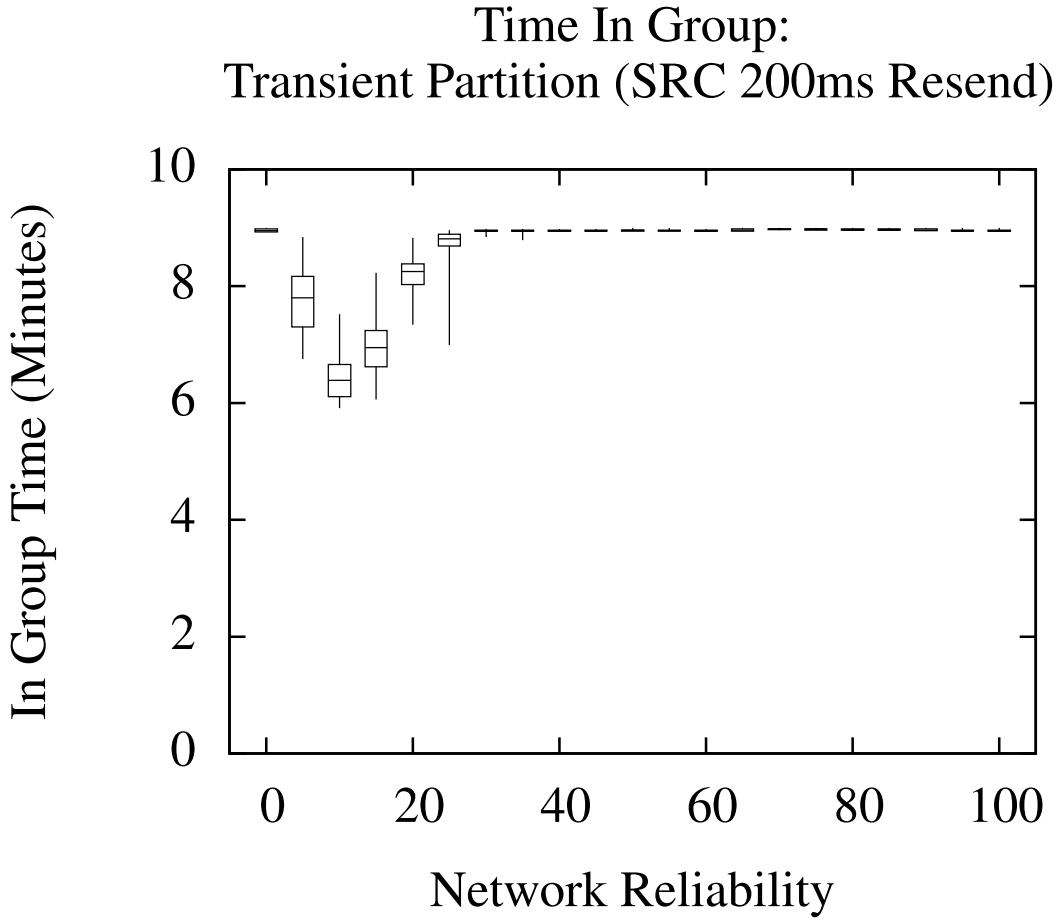

Figure 4.6: Time in group over a 10 minute run for the transient partition case with 200ms resend time

popular tool for reliability analysis. SharpE measured the reward collected in 600 seconds, minus the reward that was collected in the first 60 seconds (to emulate that the first 60 seconds were discarded in the experimental runs.) The SharpE results are plotted along with the experimental results in Figures 4.9 and 4.10.

The race condition between processes during an election is a consideration in the original leader election algorithm, and is an additional factor here. The simulator provided a parameter to allow the operator to select how closely synchronized the peers were (the time difference between when each of them would search for leaders.)

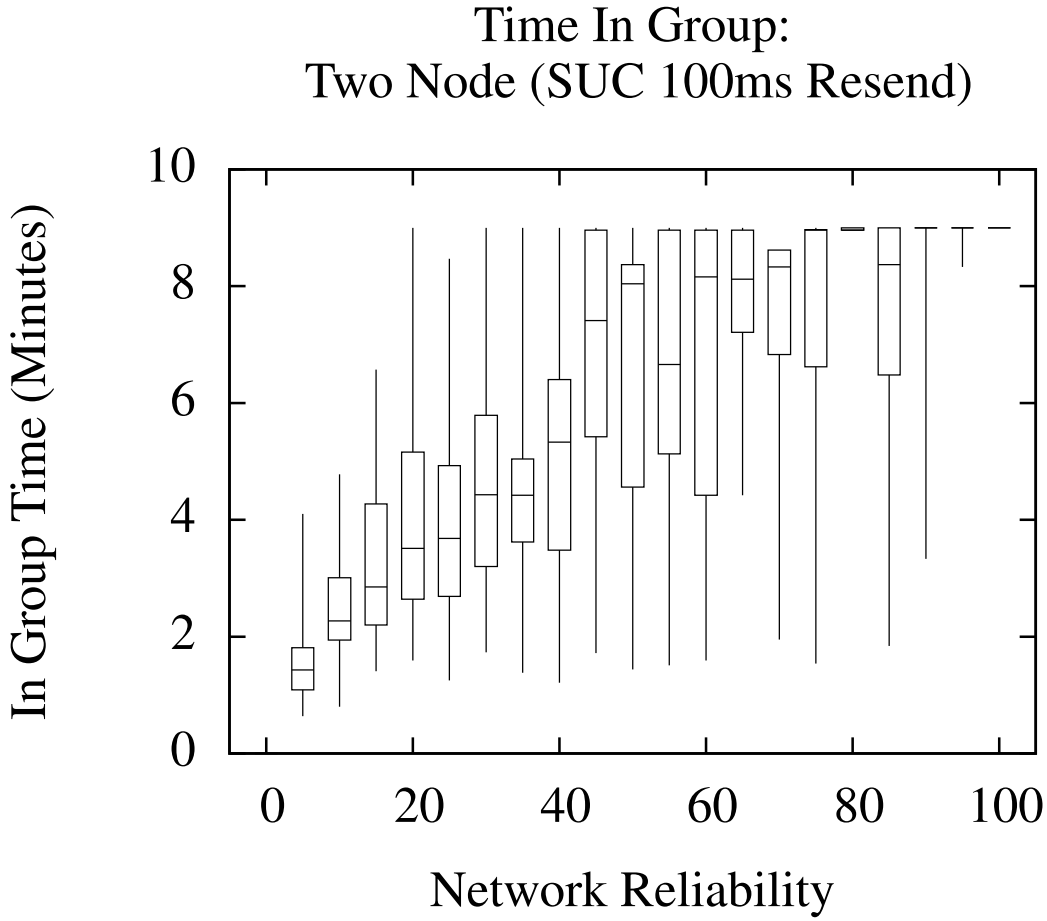

Figure 4.7: Time in group over a 10 minute run for two node system with 100ms resend time

The exchange of messages, particularly during an election had a tendency to synchronize nodes during elections, and so the nodes could synchronize even if they did not initially begin in a synchronized state. As a result, the simulation results aligned best for the 100ms resend case with 1 ticks (Approximately 100ms difference in synchronization between processes) and 2 ticks (Approximately 400ms) in the 200ms resend case.

Models fit to the non-real-time code in groups larger than 2 processes did not fit well. This is presumed to be a combination of several factors. The suspected major

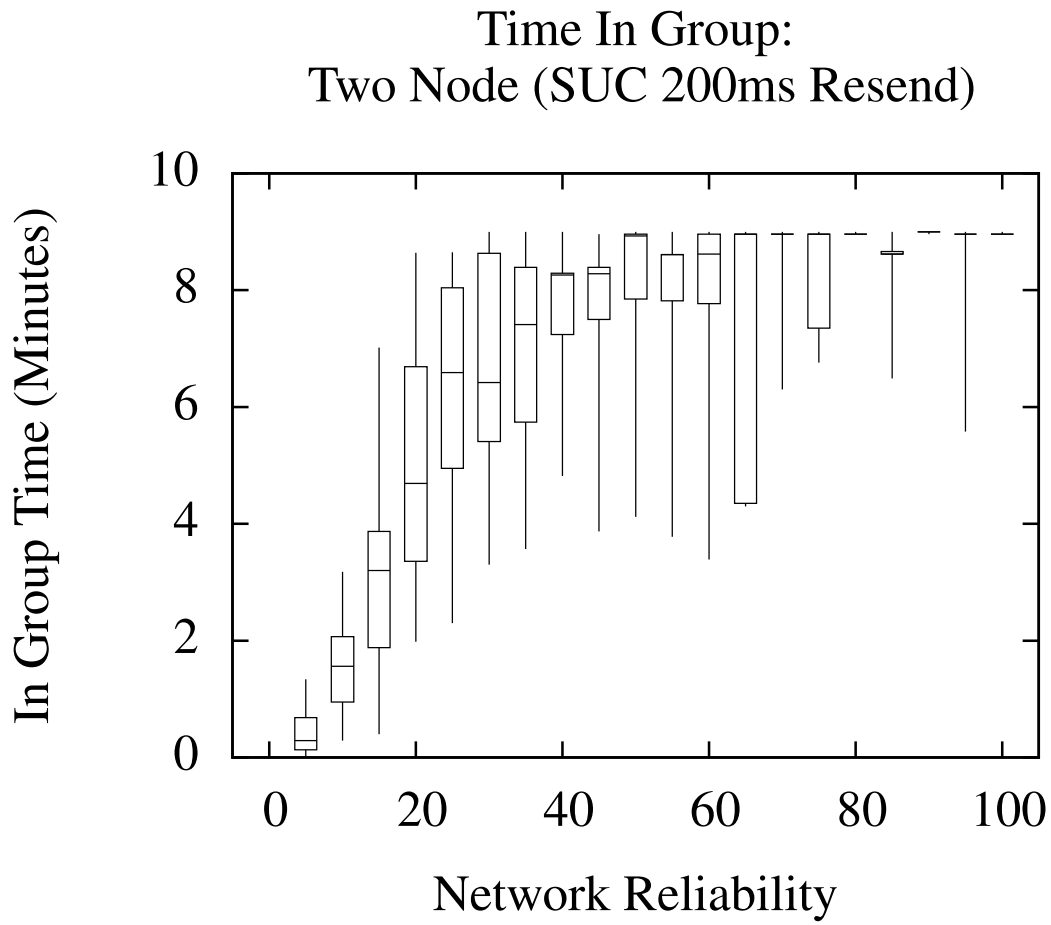

Figure 4.8: Time in group over a 10 minute run for two node system with 200ms resend time

source of fault included the structure of the chain, which naturally assumes that all processes enter the election state a roughly the same time, which is not typically true for any number of processes greater than 2. Additionally, the simulator could only assume that the synchronization between processes was mostly fixed, which was not the case in the larger configurations, since the coincidental synchronization that occurred in the two node case was suppressed by the increased number of peers. Furthermore, an issue with SharpE was discovered that prevented the particular structure of the chains produced from being handled correctly. To circumvent that, issue, SharpE

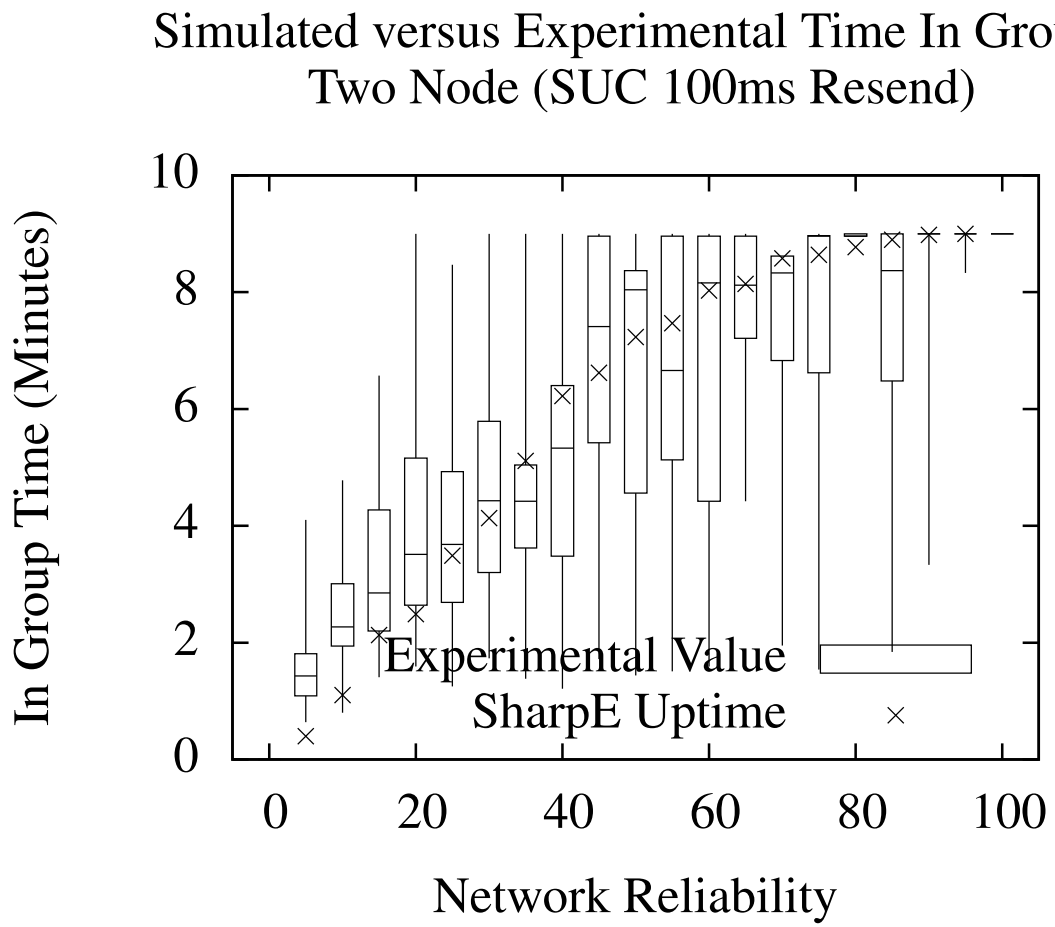

Figure 4.9: Comparison of in-group time as collected from the experimental platform and the simulator (1 tick offset between processes).

was replaced by a random-walker which generates exponentially distributed numbers and follows the paths of the chain, across several hundred trials, in order to collect time in group data for models which SharpE cannot process.

The structure of the Markov Chain, which assumed that processes enter the election state mostly simultaneously was an appropriate assumption for the real-time system, since the round-robin scheduler synchronizes when processes run their group management modules. The simulator was set to assume that the synchronization between processes was very tight, and new experimental data was collected for the

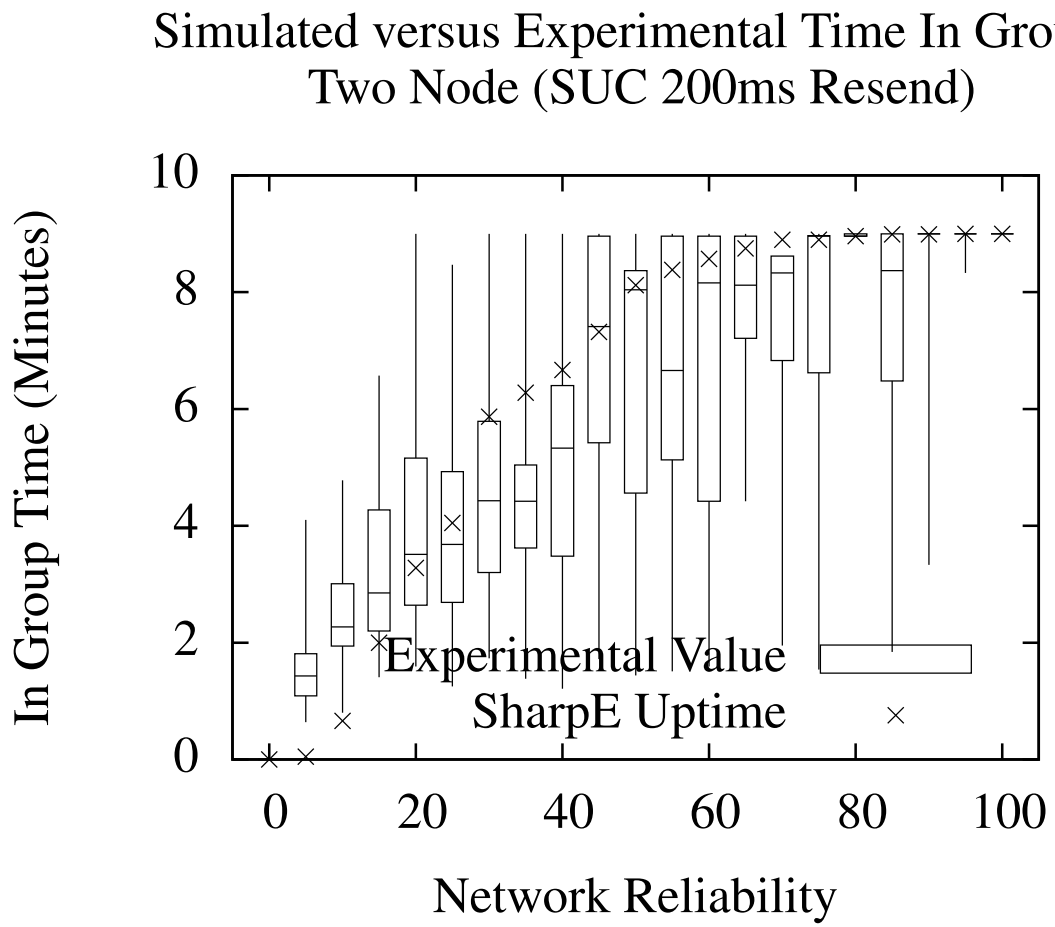

Figure 4.10: Comparison of in-group time as collected from the experimental platform and the simulator (2 tick offset between processes).

4 node, transient partition case. The collected data is overlaid with the results from the random walker in Figures 4.11 and 4.12.

As a measure of the strength of the model, the correlation between the predicted value was compared. The average error was also computed for each of the samples taken. This information is presented in Table 4.1.

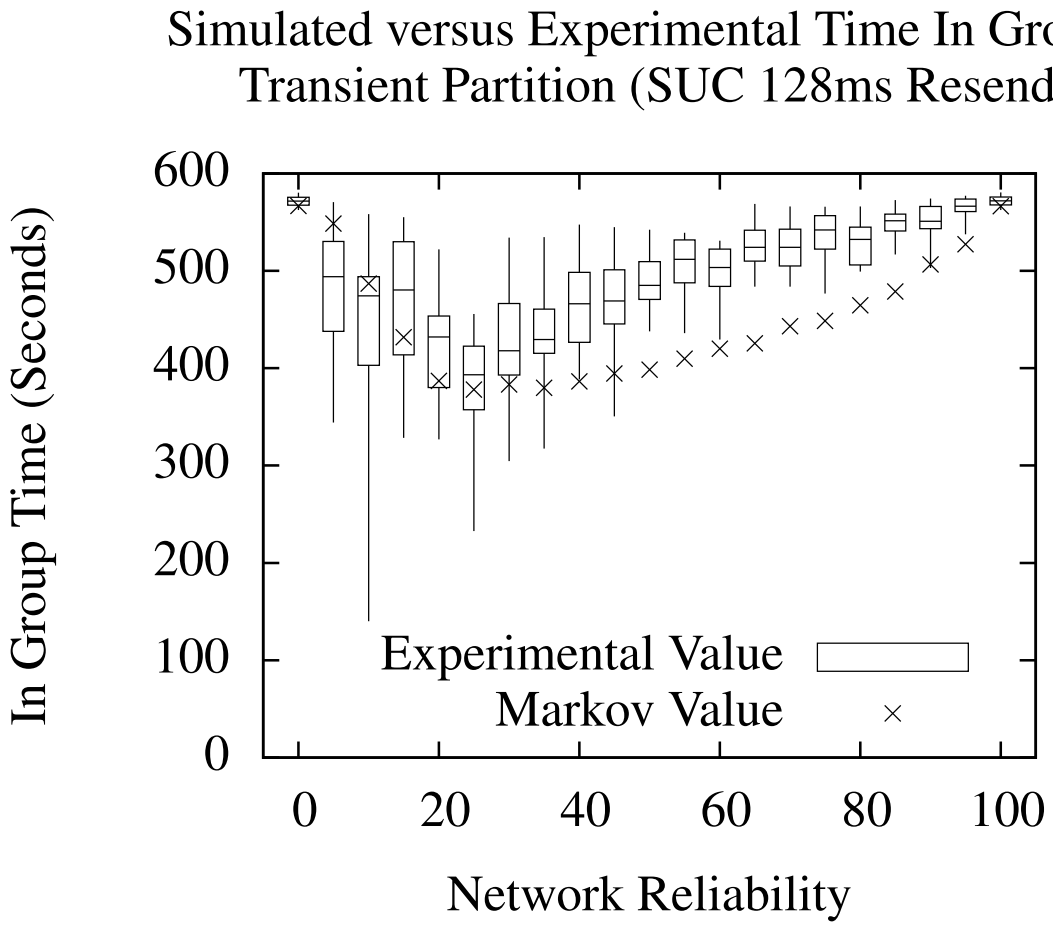

Figure 4.11: Comparison of in-group time as collected from the experimental platform and the time in group from the equivalent Markov chain (128ms between resends).

Table 4.1: Error and correlation of experimental data and Markov chain predictions

Resend	Correlation	Error
128	0.7656	11.61%
64	0.8604	11.70%

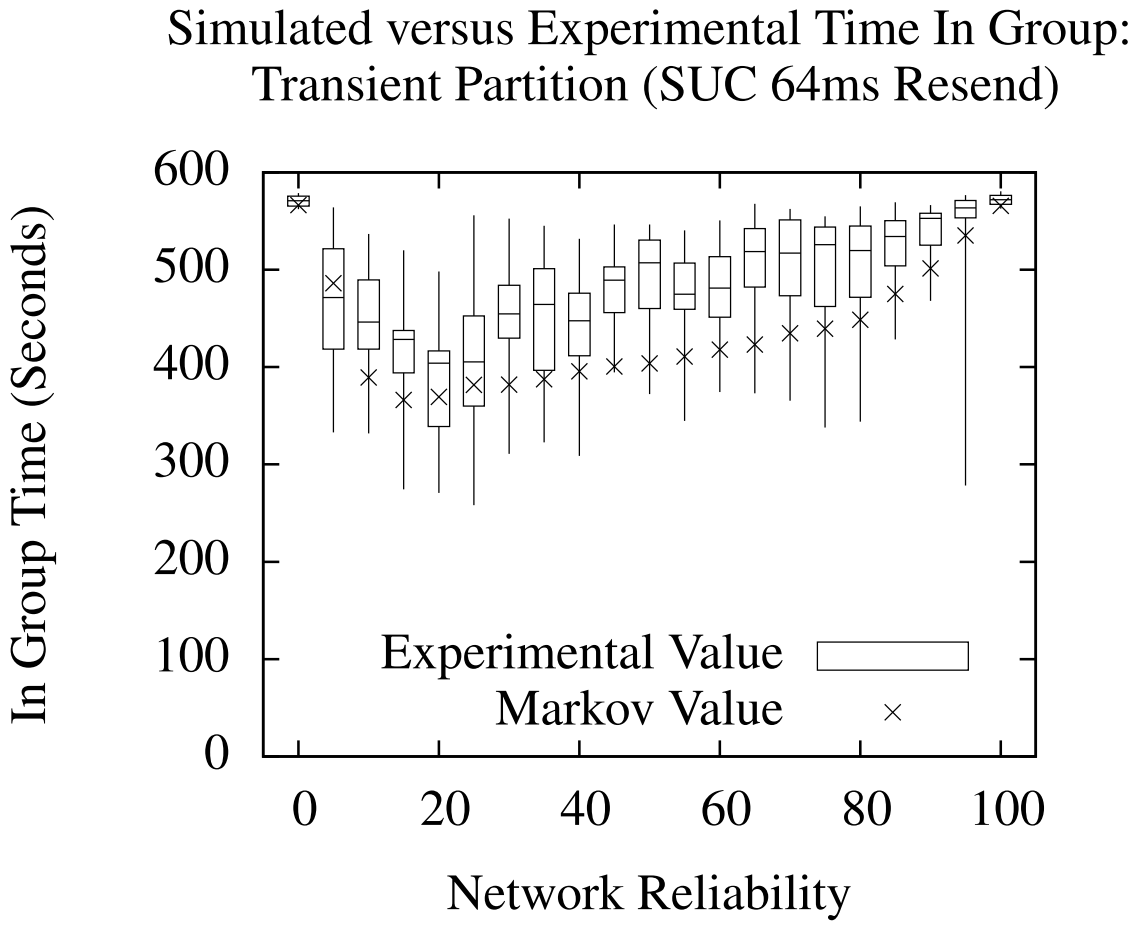

Figure 4.12: Comparison of in-group time as collected from the experimental platform and the time in group from the equivalent Markov chain (64ms between resends).

5. CONCLUSION

This work presented a new approach for predicting the behavior of a real-time distributed system under omission failure conditions. By using a continuous time Markov chain, a variety of insights can be gathered about the system, including observations such as how long a particular configuration will be stable for, and the behavior of the system in the long run. The Markov results will be used to make better real time schedules to better react to the network faults we plan on introducing to our testbeds. For example, if migrations are failing and a sufficient number of migrations can cause the physical system to fail, the scheduler may need to behave in a manner that limits the number of failed migrations that can occur before group reconfigures. This work is a stepping stone towards designing a real-time schedule that manages the system correctly when there are cyber faults. Schedules and behavior can be designed around how the system behaves on its worst days. This work also allows these schedule designs and evaluations to be completed much more quickly than they could be by running the system for long periods of time. Therefore results from the test bed, combined with results the models yield schedules that improve the stability of the system during cyber faults.

The DGI group management system was first run with a testing framework that organized various configurations as prescribed by a series of input files and commands. The collected results provided insight into the interactions of the system as the omission failures increased.

However in environments where there are fewer omission failures, there were not a large enough number of events collected to make a satisfactory conclusion about the inter-arrival time about events at those levels of omission failures. In correct this we developed a simulation tool that recreates the events of the full DGI system in

controlled circumstances. These collected events were used to construct a Markov chain that relates to the collected experimental results. These chains were fed into SharpE to produce an in group time metric equivalent to those collected using the experimental platform.

The simulator will allow the construction of larger models more quickly and will lead to increasingly refined methods of gauging the amount of time in group. This information can then been combined with the instability metrics collected from a physical system to rigorously determine the relationship between omission failures and the interaction between processes in controlling the physical system in order to prevent the number of omission failures from causing physical instability in the system.

It is important that the behavior of the cyber controller only supplements its physical component. If the behavior of the cyber component can potentially cause instability by making changes that do not benefit the physical system, particularly in cases where a system without a cyber component would have remained stable without interaction.

The primary concern are scenarios in which the cyber controller attempts to make physical components which are not connected in the physical network interact, and scenarios where a fault in the cyber network causes the paired events (where two physical controllers change to accomplish some transaction or exchange) to only be partially executed. For example, in the DGI load balancing scheme, a node in a supply state injects a quantum of power into the physical network, but the node in the demand state does not change to accept it. These errors, which is the primary focus of this work could cause instability if a sufficient number of these failed exchanges occur.

In [6], Cloudhari et. al. show that failed transactions can create a scenario where the frequency of a power system could become unstable. They classified the

stability of the system using a k-value which is a measure of the number of migrations which can fail before the system becomes unstable. As part of this the have created a series of invariants which a system must meet to be considered stable and remain stable. Given this, a metric of what the physical system can bear before becoming unstable, it is possible to tune the cyber control to limit the number of failed migrations by managing the membership of groups.

In a round-robin real-time schedule, each module gets a fixed amount of time to execute. As a consequence there is a fixed number of migrations a load balancing module can execute before the system re-evaluates the group's stability. Thus, a node which is failing to migrate due to lost messages can be handled correctly. It may be that the correct method is to remove from the group, change algorithms or adjust parameters. The correct action to take in the event of cyber failure in order to prevent physical failure is a topic of future research.

The rate that the system should reconfigure then is a function of the maximum number of failed migrations that the system can take, the time it takes to write to the channel and the time it takes process messages. The amount of time in group can also be consideration for which algorithm to select based on the needed amount of time to perform its work.

Group Management can be used as a critical component in a real-time distributed system to manage the number of lost messages, and as a consequence the number of failed migrations in a CPS. It is critical to understand how frequently nodes enter and exit the group based on lost messages and how many migrations fail as a consequence of those messages.

As a component of this work, the physical network will be examined. Using power simulations using PSCAD and the RTDS located at Florida State University, various models will be tested for physical stability while being subjected to packet loss.

The work presented in this document extends the work presented in [24] and will be presented as journal paper. Adding analysis by causing physical instability will be a novel analysis for smart grid projects and CPSes in general. Future work should yield at least an additional journal paper and conference paper.

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] 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, Oct 2010.

[4] 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.

[5] 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.

[6] A. Choudhari, H. Ramaprasad, T. Paul, J. W. Kimball, M. Zawodniok, B. McMillin, and S. Chellappan. Stability of a cyber-physical smart grid system using cooperating invariants. In *International Computer Software and Applications Conference, 37th Annual*, 2013.

[7] K. Birman and Renesse R. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA 90720-1264, 1994.

[8] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pages 56–65, 1994.

[9] Robbert Van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of horus: A lightweight group communications system. Technical report, Cornell, 1994.

[10] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication subsystem for high availability. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 76–84, 1992.

[11] L.E. Moser, P.M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39:54–63, 1996.

[12] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical report, Johns Hopkins University, 2004.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[14] C. Gomez-Calzado, M. Larrea, I. Soraluze, A. Lafuente, and R. Cortinas. An evaluation of efficient leader election algorithms for crash-recovery systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 180–188, 2013.

[15] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, PODC '92, pages 147–158, New York, NY, USA, 1992. ACM.

[16] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

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

[18] M.M. Shirmohammadi, K. Faez, and M. Chhardoli. Lele: Leader election with load balancing energy in wireless sensor network. In *Communications and Mobile Computing, 2009. CMC '09. WRI International Conference on*, volume 2, pages 106–110, Jan 2009.

[19] Qi Dong and Donggang Liu. Resilient cluster leader election for wireless sensor networks. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2009. SECON '09. 6th Annual IEEE Communications Society Conference on*, pages 1–9, June 2009.

[20] S. Vasudevan, B. DeCleene, N. Immerman, J. Kurose, and D. Towsley. Leader election algorithms for wireless ad hoc networks. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 1, pages 261–272 vol.1, April 2003.

[21] N. Mohammed, H. Otrok, Lingyu Wang, M. Debbabi, and P. Bhattacharya. Mechanism design-based secure leader election model for intrusion detection in manet. *Dependable and Secure Computing, IEEE Transactions on*, 8(1):89–103, Jan 2011.

[22] Bong Jun Choi, Hao Liang, Xuemin Shen, and Weihua Zhuang. Dcs: Distributed asynchronous clock synchronization in delay tolerant networks. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):491–504, March 2012.

[23] A. P S Meliopoulos, G.J. Cokkinides, O. Wasynczuk, E. Coyle, M. Bell, C. Hoffmann, C. Nita-Rotaru, T. Downar, L. Tsoukalas, and R. Gao. Pmu data characterization and application to stability monitoring. In *Power Engineering Society General Meeting, 2006. IEEE*, pages 8 pp.–, 2006.

[24] Stephen Jackson and Bruce M McMillin. The effects of network link unreliability for leader election algorithm in a smart grid system. In *Critical Information Infrastructures Security*, pages 59–70. Springer Berlin Heidelberg, 2013.

[25] K. S. Kishor. Sharpe, March 2014. `http://sharpe.pratt.duke.edu/`.

[26] R.A. Sahner and K.S. Trivedi. Sharpe: a modeler's toolkit. In *Computer Performance and Dependability Symposium, 1996., Proceedings of IEEE International*, pages 58–, Sep 1996.