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
Dr. Sriram Chellappan
Dr. Sahra Sedighsarvestani

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 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 presents a techinque for modelling the availability of a cyberphysical system. This is of high value because there is a limited amount of research that describes that models the availibility of distributed systems, and CPSs in general. This approach encodes the behavior of a dynamic reconfiguration system of a CPS as a continous-time Markov chain. These models provide an estimate of when a cyberphysical system will be able to do work and when it is reconfiguring. Generated models are presented and evaluted against data collected from actual runs of the system. Using these models, we will be able to develop methods and gaurantees to quantify and protect operation of a CPS.

ACKNOWLEDGMENTS

The author acknowledges 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

1. INTRODUCTION

A robust cyber-physical system should be able to survive and adapt to communication 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 system to continue operating normally. Additionally, process may need to react to the state change of some other process. Managing and detecting when other processes have failed is commonly handled by a leader election algorithm and failure detector.

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

FREEDM (Future Renewable Electric Energy Delivery and Management) 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 (SST), local energy storage, and 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. There is a major shortage of work on the effects cyber outages have on CPSs [4] [5]. Research that has been done, such as [6] indicate that cyber faults can cause a physical system to apply unstable settings.

This work begins with observations on the effects of network unreliability on 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 grid system. These modules include: group management, which handles coordinating processes 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.

This work presents the initial steps to better understanding and planning for these faults. As part of an analysis of omission failures, 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 cyber system can be specified to prevent faults.

In Chapter 2, we outline the state of the art: other distributed frameworks and works similar to our own. Chapter 3 provides a background of the architecture of DGI and the properties of the created Markov models. Chapter 4 shows the results that have been collected so far. Chapter 5 summarizes the work and research goals.

2. RELATED WORK

2.1. FAILURE CLASSIFICATION

Processes can fail in a variety of ways. In this work we focus on how cyber processes can fail. There are a variety of faults that have been classified. In this work, we focus on two different modes of failure:

2.1.1. Crash or Fail-Stop Failure. A crash failure or its more generalized form, a fail-stop failure, describes a failure in which a process stops executing. In general, this is considered to be an irreversible failure, since a process typically does not resume from a crashed state. There are categorizations of crash failures, called napping failure where a process will appear to have crashed for a finite amount of time before resuming normal operation. Crash failures are impossible to detect with absolute certainty in an asynchronus system. Processes can be suspected by other processes through the use of challenge/response messages or heartbeat messages that allow a process to prove that it has not crashed yet. A system that can handle a crash failure is implied to be able to handle a fail-stop failure.[7]

The fail-stop failure has three properties [7]:

1. When a failure occurs the program execution is stopped.

2. A process can detect when another fail-stop process has failed.

3. Volatile storage is lost when the fail-stop process is stopped.

2.1.2. Omission Failure. An omission failure occurs when a message is never recieved by the reciever. Omission failures can occur when the communication medium is unavalable, or when the latency of message exceeds a timeout for it's expected delivery. Protocols like TCP do not tolerate omission failures: a packet is

resent until it is acknowledged by the reciever. If the acknowledgement never comes, the connection is closed. As a contrast, UDP assumes that any datagram could be lost and it is the responsibility of the programmer to handle missing datagrams appropriately. An omission failure can have the same observable affects as a napping failure in some situations. [7]

2.2. VIRTUAL SYNCHRONY

Virtual synchrony (and its original implementation, Isis[8]) is an execution model for distributed systems. Virtual synchrony is based on the idea of synchronous execution where each process acts based on a globally shared clock. Synchronous systems are the easiest to program for, but true synchrony is not feasible in most circumstances. Instead, the virtual synchrony execution model was developed.

Virtual synchrony is a model of communication and execution that allows the system designer to emulate synchronous execution. 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.

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 [8, p .101].

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

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. A pair

of histories H and H' are equivalent if, the history of a process p, $H|_p$, cannot be differentiated from another history, $H'|_p$, based on the casual relationships between the events in the history. [8, 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' does not. A virtually synchronous system is one where each history in the system is indistinguishable from all histories produced by a synchronous system. [8, p .104]

2.2.1. 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, implementations share common features. 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. A view refers to a specific group: the set of processes in that view, and a leader if there is one. A group is a general descriptor for a collection of processes. Process groups in virtually synchronous systems place obligations on the delivery of messages to members of a view. A history H of a view is legal if [8, p .103]:

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) occurs at the same logical time for all processes in the view.

4. All multicast message deliveries occur in the current 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.

A process group interacting via message passing creates a legal history for the virtually synchronous system. However, processes can fail. Virtual synchrony systems have several properties to handle fail-stop failures[8, p .102]:

- The system employs a membership service. This service monitors for failures and reports them to the other processes in the view.
- 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.

Virtual synchrony's failure model uses the fail-stop failure model. Virtual synchrony does not guarantee how messages will be delivered while transitioning between views. Virtual synchrony only guarantees that view changes are totally ordered with respect to the regular messages sent in the view. A message first sent in an old view may be delivered in a new view. If partitions were allowed to join, the ordering of messages still outstanding from the original views is ambiguous.

Regular messages in virtual synchrony, those which do not announce view changes, can be casually ordered or totally ordered. Additionally, virtual synchrony supports two classes of multicast delivery guarantees: uniform and non-uniform. The uniform property obligates that if a multicast message is delivered to a process in a view, it is delivered to all processes that view. Non-uniform multicast is a multicast that does not guarantee the uniform delivery property.

2.3. EXTENDED VIRTUAL SYNCHRONY

One of the major shortcomings of the original virtual synchrony model lays in how it handles network partitions. In virtual synchrony, when the network is 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: the process joining the view must be a new process so that there are no message delivery obligations for that process in the view. [9] presented an improved 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. 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.

Extended virtual synchrony places obligations on the message delivery service, described informally below [9]:

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. A view is "installed" when process recognizes a change in group membership.

4. 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.

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

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

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

8. 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.

9. 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 were also delivered.

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 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 views with respect to the regular messages in the primary partition to be totally ordered. Additionally, two consecutive primary views have at least one 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 process' obligation set is empty, the view transitions from a transitional view to a regular view and execution continues as normal.

2.3.1. 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 its 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.

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 process 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 process in the current design of the system.

2.4. ISIS (1989) AND HORUS (1996)

A product of Dr. Kenneth Birman and his group at Cornell, Isis [8] and Horus [10] are two distributed frameworks which are comparable to the DGI. Although these projects are no longer actively developed, Isis is 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 led Birman's group to 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 offers a modular design with a variety of extensions which can change message characteristics and performance as needed by the project.

2.4.1. Group Model. In Horus and Isis, a group is a collection of processes that can communicate with one another to do work. Multicasts are directed to the view, and are guaranteed to be received by all members or no members. 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 that 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' modules 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.

2.5. TRANSIS (1992)

Transis [11] 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 message complexity, where N is the number of processes.) 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 gateways, called Xporters, to deliver messages between the local views. Transis uses a combination of acknowledgements, which are piggybacked 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.

2.6. TOTEM (1996)

Totem[12] 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.

2.7. SPREAD

Spread[13] is a modern distributed framework. It 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 to the gateways in the other distributed toolkits, however, the daemon is a special, dedicated message passing process, rather than a participating process with a special role.

Using a dedicated daemon allows a Spread view 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.

2.8. FAILURE DETECTORS

Failure detectors [14] (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 breadth of work related to different classifications of failure detectors, with different properties. Some of the properties include[14]:

- 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.

One class of failure detectors, Omega class Failure detectors, are particularly interesting because of [15]. 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 $\diamond\mathcal{W}$ [14], $\mathcal{W}$ [16] [17] and Ω (Omega) [15].

[15] studies an Omega class failure detector using OmNet++[18], a network simulation software package. 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, [15] 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

This models the group management module of the FREEDM DGI. The DGI is a smart grid operating system that organizes and coordinates power electronics. It also negotiates contracts to deliver power to devices and regions that cannot effectively facilitate their own needs. DGI leverages common distributed algorithms to control the power grid, making it an attractive target for modeling a distributed system.

The DGI software consists of a central component, known as the broker. This broker is responsible for presenting a communication interface. It also furnishes any common functionality the system's algorithms may need. These algorithms, grouped into modules, work in concert to move power from areas of excess supply to excess demand.

DGI utilizes several modules to manage a distributed smart-grid system. Group management, the focus of this work, implements a leader election algorithm to discover which processes are reachable within the cyber domain. Other modules provide additional functionality, such as collecting global snapshots, negotiating the migrations, and giving commands to physical components.

DGI is a real-time system; certain actions (and reactions) involving power system components need to be completed within a pre-specified time-frame to keep the system stable. It uses a round robin schedulerin which 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 used in this designed around a broker architectural specification. Each core functionality of the system was implemented within a module that was provided access to core interfaces. These interfaces provided functionality such as scheduling requests, message passing, and a framework to manipulate physical devices.

The Broker provided a common message passing interface that all modules could access. This interface was then used to pass information between modules. For example, the list of peers in the group was sent as a message.

Several of the distributed algorithms used in the software required the use of ordered communication channels. DGI provided a reliable ordered communication protocol (The sequenced reliable connection or SRC). It also offered a "best effort" protocol (The sequenced unreliable connection or SUC). This protocol was also FIFO (first in, first out), but provided limited delivery guarantees.

Simple message delivery schemes were used in order to avoid complexities introduced by using TCP. Constructing a TCP connection to a process that either had failed or was unreachable required a considerable amount of time. We elected to use UDP packets which do not have those issues, since the protocol is connectionless. UDP also allowed for the development of multiple protocols with various properties to evaluate which properties are desirable. Lightweight protocols which were best effort oriented were implemented to deliver messages within the requirements. The protocols listed here continued operating despite omission failures. They follow the assumption that not every message is critical to the operation of the DGI and that the channel did not need to halt entirely to deliver one of the messages.

3.2.1. Sequenced Reliable Connection. The sequenced reliable connection was a modified send and wait protocol that had the ability to stop resending

messages and move on to the next one in the queue if the message delivery time exceeded some timeout. The design of this protocol met several criteria:

- Messages needed to be delivered in order. A number of distributed algorithms rely on the assumption that the underlying message channel is FIFO.
- Messages could become irrelevant. Some messages may only have a short period in which they are worth sending. Beyond that time period, they should be considered inconsequential and thus, skipped. Message expiration times were used to address this issue. After a certain amount of time had passed, the sender would longer attempt to write that message to the channel. Instead, he would proceed to the next unexpired message and attach a "kill" value. The kill value was the sequence number of the last message the sender knows the receiver accepted.
- Every effort needed to be made to deliver a message while it was still relevant.

One adjustable parameter, the resend time, controlled how often the system would attempt to deliver a message it had not received an acknowledgment for. A resend function was periodically called in an attempt to redeliver lost messages to the receiver.

3.2.2. Sequenced Unreliable Connection. The SUC protocol was a best effort protocol: it employed a sliding window to deliver messages as quickly as possible. A window size was chosen, the sender can have up to that many outstanding messages at any given time. The receiver would look for increasing sequence numbers, and disregard any message that was of a lower sequence number than was expected. The purpose of this protocol was to implement a bare minimum: messages were accepted in the order they are sent.

The SUC protocol's resend time could be adjusted. Additionally, the window size was configurable. However the window size was left unchanged for the tests presented in this work.

The SUC protocol was developed because early hypothesis omission failures supposed that a lighter weight protocol might be more advantageous. The lightweight protocol also presented a more complex behavior to model using Markov chains, since the nature of the protocol allowed for a race condition were a packet is lost.

3.2.3. Real Time. The DGI's specifications also called for real time reaction to events in the system. These 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 was given a phase which grants it a specific amount of processor time. Modules used this time to complete any tasks they had prepared. When the allotted time was up the scheduler changed context to the next module. This interaction is illustrated in Figure 3.1

Modules informed the scheduler of tasks they wish to perform. The tasks could be scheduled for some point in the future, or scheduled to executed immediately. When a tasks became ready was inserted into a ready queue for the module it was been scheduled for.

When that modules phase was active, tasks were pulled from the ready queue and executed. When the phase was complete, the scheduler stopped pulling tasks from the previous module's queue and began pulling from the next module's queue.

This allowed enforcement of upper bound message delay. Modules had a specific amount of processing time allotted. Modules with messages that invoked responses typically required the responses to be received within the same phase. Round numbers enforced that the message was sent within the same phase.

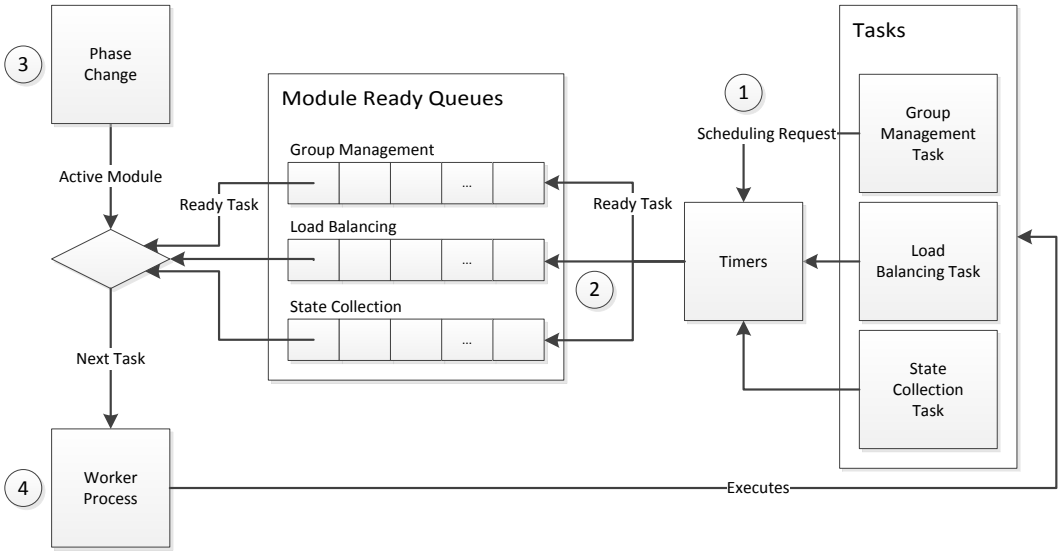

Figure 3.1: The real time scheduler used a round robin approach to allot execution time to modules.

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

2. When the timer expires the task is placed into the ready queue.

3. Modules were assigned periods of execution (called phases) which were a predetermined length. After the specified amount of time had passed, the module's phase ends and the next module in the schedule began to execute.

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

Modules were designed and allotted time to allow for parameters such as maximum query-response time (based on the latency between communicating processes). This implied that a module that engaged 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 [19]. Originally published in 1982, this algorithm provides a robust election procedure that allows for transient partitions. Transient partitions are formed when a faulty link between two or more clusters of DGIs causes the groups to divide temporarily. These transient partitions merge when the link becomes 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 [19], a large number of election algorithms have been created. Each algorithm is designed to be well-suited to the circumstances it will deployed in. Specialized algorithms exist for wireless sensor networks [20][21], detecting failures in certain circumstances [22][23], and of course, transient partitions. Work on leader elections has been incorporated into a variety of distributed frameworks: Isis [8], Horus [10], Totem [12], Transis [11], and Spread [13] all have methods for creating groups. Despite this wide array of work, the fundamentals of leader election are consistent across all work. Processes arrive at a consensus of a single peer that coordinates the group. Processes that 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 an up-to-date list of peers for the members of his group. Group members monitor 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 nodes will enter a recovery state and operate alone until they can identify another coordinator. Therefore, a leader

and each of the members maintain a set of processes which are currently reachable, a subset of all known processes in the system.

Leader election can also be classified as a failure detector [15]. Failure detectors are algorithms which detect the failure of processes within a system; they maintain a list of processes that they suspect 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 the 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 within the system. The member sends a query its leader. The member will only suspect the leader, and not the other processes in their group. Of course, simple modifications could allow the member to suspect other members. However, that modification is not implemented in DGI code.

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

3.4. NETWORK SIMULATION

Network unreliability was 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.

"Lost" messages are dropped on the receiver side. Code was inserted in the datagram processing code of the broker. The broker would not deliver the message to the modules if the message is selected to be dropped. On receipt of a message, the broker's examinesdthe source and randomly selected based on the prescribed reliability to drop a message. A dropped message was not delivered to any of the modules and was not acknowledged by the receiver. This method emulated 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 it.

The behavior of the DGIs acting togethere was be modeled as a collection of states. Each state described configuration of the system, and that the transitions between those states model failure events or election events. These transitionas are probabilistic, and as a result it is a natural extension to model the distributed system as a Markov Chain.

Models can be expressed either in 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 that returns to the same state. To contrast, a continuous time model assumes that the time between transitions are exponentially distributed.

3.5.1. Continuous Time Markov Chains. The model 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 $F_X(s)$ equal the complete history of a Markov chain X up to the state s, and $j \in S$, where S is the complete set of states in the model, then the probability of transitioning to the next state j ($X_{n+1} = j$) only depends on the current state s (X_n): [24]

$$P\{X_{n+1} = j | F_X(s)\} = P\{X_{n+1} = j | X_n = s)\} \tag{3.1}$$

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. Let $X(s) = i$ indicate the model is in state i at time s. If the model is time homogenous, then the probability of transitioning to state j only depends on the time spent in that state (t).

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

Each transition has some expected value or holding time which describes the amount of time before a transition occurs. Continuous time models do not have transitions that 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: [25]

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

As a result, the expected or mean value of an exponential distribution, is a function of the parameter λ: [25]

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

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 [24]

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

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 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 in the state. 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 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. Constructing The Markov Chain. Consider a set of processes, that are linked by some packet-based network protocol. 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 packet loss, a large variety of delivery protocols have been developed. Each protocol has a different set of goals and objectives, dependent on the application. Two protocols, each with different delivery characteristics, were used in this study.

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

The leader election algorithm was centered around two critical events: checking and elections. The check system is used to detect both crash-stop failures and the availability of processses for election. Processes in the system occasionally exchanged messages to determine if the other processes had failed. Leaders exchanged messages to discover new leaders.

The DGI could perform work if in a group, and not in an election state. The group management module instructs other modules to stop during an election. The collected data in the previous sections was based on that assumption. The Markov chains that model those scenarios also use that assumption.

Events in the distributed system were assumed to be distributed exponentially. These were modeled in the chain by $\lambda(x)$ the parameter of the exponential distribution. [25][24] It is important to note that

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

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

where $\lambda(x)$ is the exponential parameter for the total time spent 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 of transitioning from state x to state y.

Failure Detection. When a leader sent its check messages, the processes either responded in the positive (indicating that they are also leaders) or in the negative (indicating that they have already joined a group). This message was sent to all known nodes in the system. If a process replied that it was also a leader, the

original sender entered an election mode and attempted to combine groups with the first process. Members that failed to respond were removed from the leader's group.

In contrast, the member only sent a check message to the leader of its current group. As with the leader's check message, the response could be either positive or negative. A "yes" response indicated that the leader is still available and considered the member a part of its group. A "no" response indicated that either the leader had failed and recovered, or it has suspected the member process of being unreachable. In either case, the member had been removed from the leader's group. The member would enter a recovery state and reset itself to an initial configuration where it was in a group by itself.

When a change in membership occured, either due to recovery or a suspected failure, the leader pushed a list of members to the group. Members cannot suspect other members of failing. Only the leader can identify failed group members.

Figure 3.2 illuststrates the model of the failure detection stage of the leader election algorithm. A set of processes begin in a normal state as part of a group. The leader sent a query to every member, and every member sent a query to the leader. If a response was not received in either direction, the process was considered to be unreachable. If the original message was sent by the leader, the leader removed that member and informed the remaining members. Otherwise, the member that sent the original message entered a recovery state.

The system remained in the original state as long as all nodes completed 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 be the probability that at least one peer failed to complete the exchange. Based on this, the expected amount of time in the grouped state (T_G) was

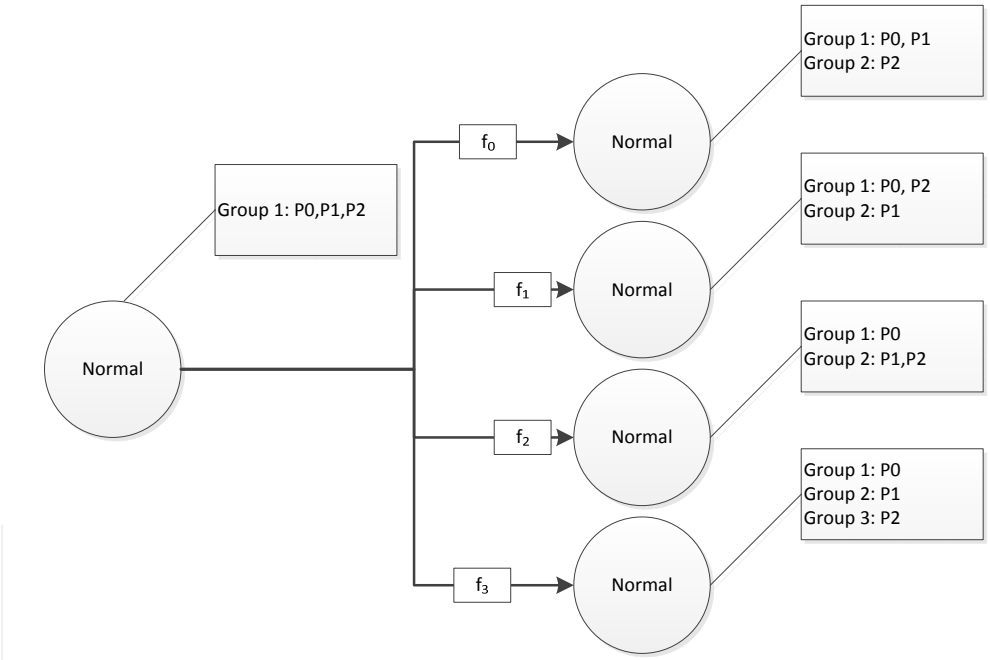

Figure 3.2: A diagram showing a partial Markov chain for failure detection

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

Let λ_G equal the exponential parameter of the exponential distribution for the expected time in some set of groups. The probabilities of each possible transition can then be related to the parameter for that configuration. Let p_i be the probability of transitioning to some set of groups i and let f_i be the exponential parameter for the transition to some other configuration:

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

Leader Election. During elections, a highest priority leader (identified by its process ID) sent invites to the other leaders it had previously identified. If those

leaders accepted the highest priority leader's invite, they replied with an accept messageand forwarded the invite to their members. If the highest priority process failed to become the leader the next highest will sent invites after a specified interval had passed.

Therefore, the membership of the system was affected in two ways: 1) election events that changed the size of groups and 2) failure suspicions (via checks) which decreased the group's size. Note that elections coulf decrease the group's as well as increase them. If a round of forwarding invites fails, the group size could decrease.

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

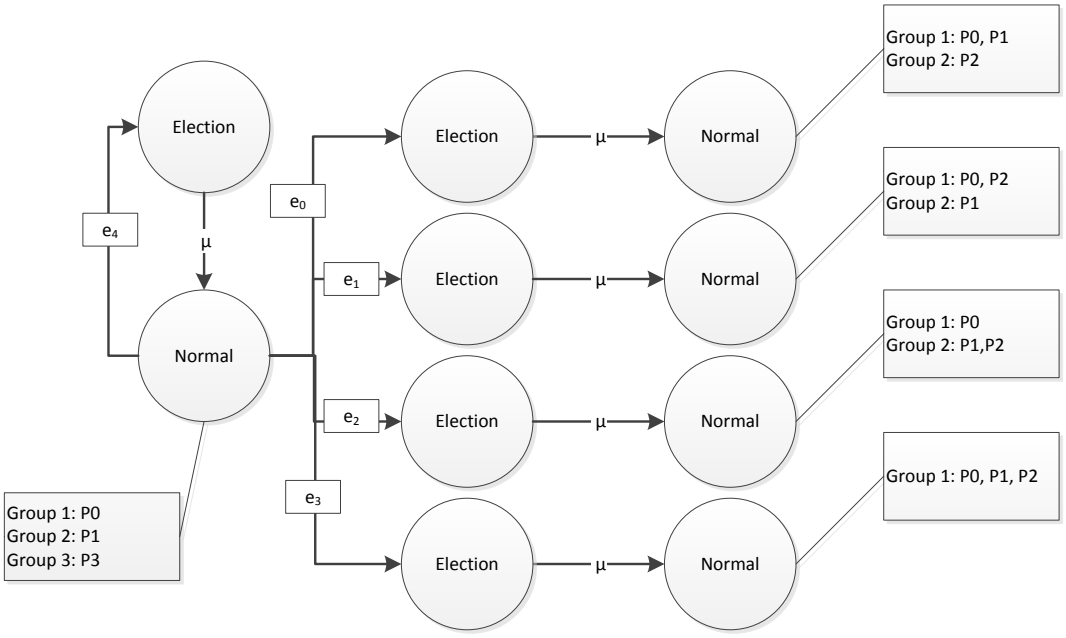

Figure 3.3: A diagram showing a partial Markov chain for an election

A continuous time Markov model of a single election is presented in Figure 3.3. Here, 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 that do no discover any leaders. This 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 be the probability that the exchange discovers a leader.

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

The parameters (e_x) in Figure 3.3 are a function of T_D and the probability an election results in configuration x (p_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, in which the peers hold an election to determine a new configuration. An election can either succeed or fail, as illustrated in Figure 3.3. This results in a new system configuration, or each involved process returing 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. Each state has a combination of election transitions and failure transitions. States where all reachable nodes are in the same group do not have election transitions. Likewise, states where there are no reachable leaders do not have election transitions. The combined model is predictive of the system's

overall characteristics. The time spent in a particular configuration is a function of λ's for all events that can cause the system to transition away from a configuration.

Simulations of individual events were performed to construct the Markov chain. The circumstances for the events were assumed to be homogeneous; processes only differws by their process ID. Under this assumption, the simulation of events was broken down into a series of scenarios that represented of the system's events. Because each scenario is independent of other scenarios, each scenario ran independently. Additionally, because the circumstances were assumed to be homogeneous, scenarios that are similar such as ones where two processes swapped roles were simulated only once. The results were transformed from one scenario to another with a simple mapping. This mapping scheme and parallizability helped keep the state space explosion of the potential states under control.

3.5.3. Assumptions. The inter-arrival time between events was assumed to be exponentially distributed. Furthermore, it was assumed that the system would be relatively well synchronized, with most elections occurring at the same time. This assumption was valid for the 2-node cases in the non-real-time code. An increased number of processes violated this assumption for the non-real-time code. With the use of the round-robin scheduler with synchronization it was possible enforce the synchronization assumption with the real-time code.

All participating peers were assumed to be on the same schedule; all peers began executing the model simultaneously. Synchronization was accomplished using Choi's work [26]. It was also assumed the clocks are synchronized. If the network has faulted, process clocks would not drift noticeably from their last synchronization. A production system would likely use GPS time synchronization to obtain certain power system readings [27].

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 so that any transients in the test could be removed. The tests were run for ten minutes, however the maximum result was 9 minutes of in group time. These graphs first appeared in [28].

4.1.1. Sequenced Reliable Connection.

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

Figure 4.2 demonstrates that as the rate at which lost datagrams were re-sent was decreased to 200ms, the in-group time decreased. This behavior was expected. Each exchange had a time limit for each message to arrive and the number of attempts was reduced by increasing the resend time.

Transient Partition Case. The transient partition case is a simple example in which 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 this study, two processes were present on each side of the partition. In the experiment, the probability of

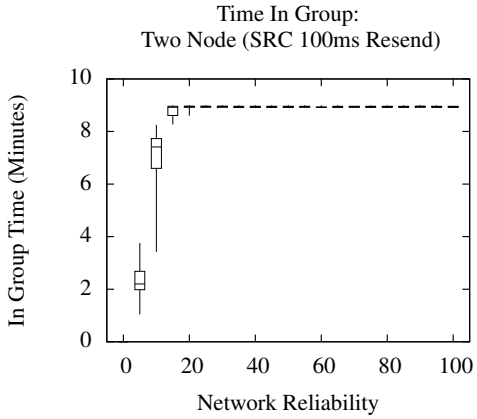

Figure 4.1: Time in-group over a 10 minute run for a two node system with a 100ms resend time

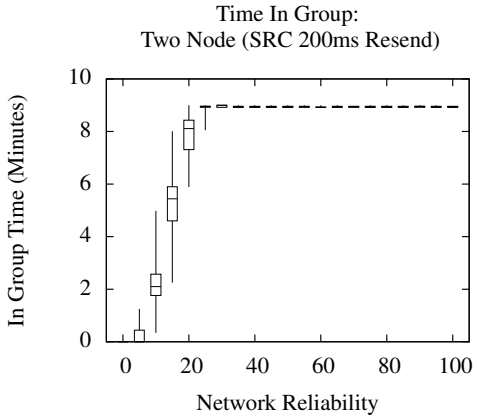

Figure 4.2: Time in-group over a 10 minute run for a two node system with a 200ms resend time

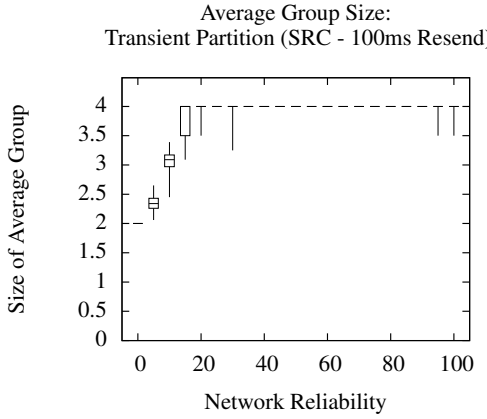

Figure 4.3: Average size of formed groups for the transient partition case with a 100ms resend time

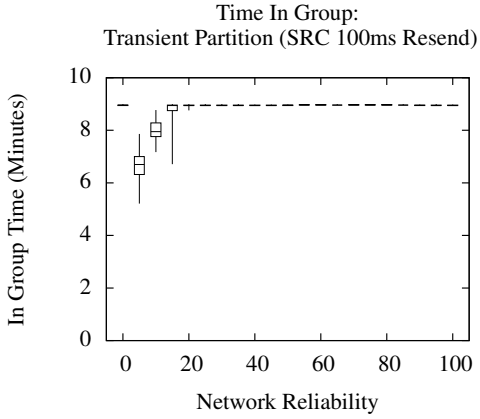

Figure 4.4: Time in-group over a 10 minute run for the transient partition case with a 100ms resend time

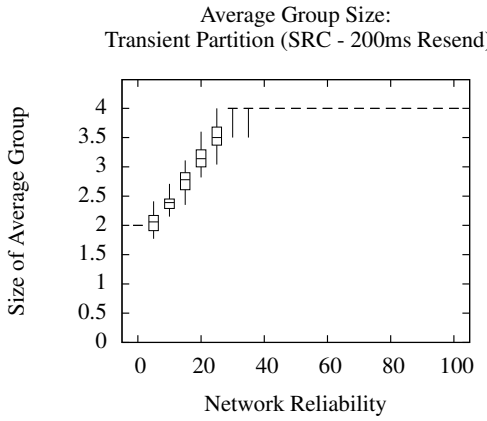

Figure 4.5: Average size of formed groups for the transient partition case with a 200ms resend time

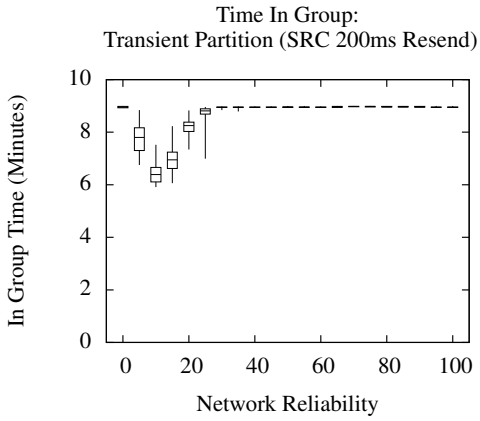

Figure 4.6: Time in-group over a 10 minute run for the transient partition case with a 200ms resend time

a datagram crossing the partition was increased as the experiment continued. 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 possible). As packets began to cross the partition (as the reliability increases), DGI instances on either side attempted to complete elections with the nodes on the opposite partition and the in group time began to decrease. During this time, however, the mean group size continued to increase. Thus, while the elections were decreasing the amount of time that the module spent in a state where it can actively do work, it typically did not fall into a state where it was in a group by itself. As result, most of the lost in group time comes from elections.

The 200ms case (Illustrated in Figures 4.5 and 4.6) suggests a similar behavior to Figures 4.3 and 4.4, with a wider valley produced by the reduced number of datagrams. The mean group size dips below 2 in Figure 4.5, possibly because longer resend times allowed for a greater number race conditions between potential leaders.

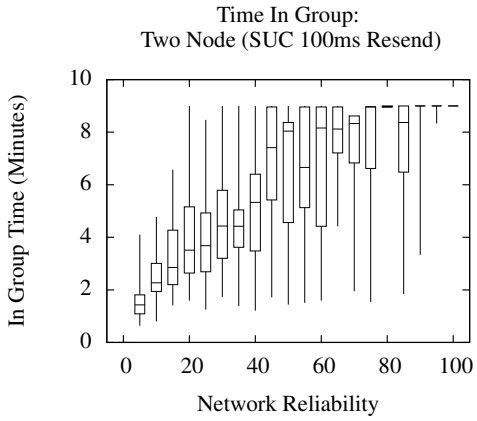

Figure 4.7: Time in group over a 10 minute run for two node system with 100ms resend time

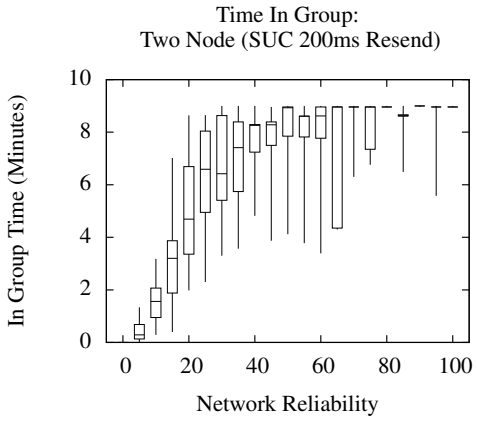

Figure 4.8: Time in group over a 10 minute run for two node system with 200ms resend time

Discussion of these race conditions is shown and discussed during the SUC section since it was more prevalent in those experiments.

4.1.2. Sequenced Unreliable Connection.

Two Node Case. The SUC protocol's experimental tests revealed an immediate problem. There is a general increasing trend for the amount of time in-group shown in Figure 4.7. There is a high amount of variance, however, for any particular trial.

In the 200ms resend case (illustrated in Figure 4.8), a greater growth rate occured in the in group time as the reliability increased. When an average was taken across all of the collected data points from the experiment, the average in group time is higher for the 200ms case than it was for the 100ms case (6.86 vs 6.09). There large amount of variance in the collected in group time, however. As a result, it is not possible to state with confidence that the there is a significant difference between the two cases.

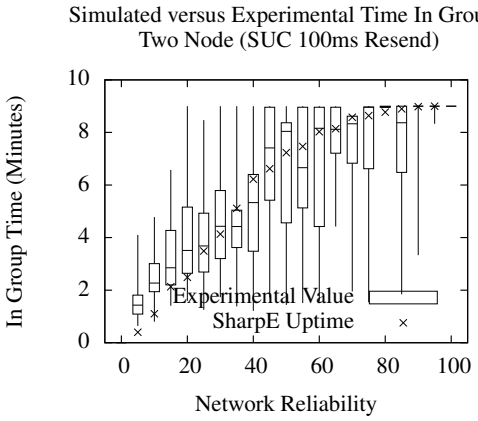

Figure 4.9: Comparison of in-group time as collected from the experimental platform and the simulator (1 tick offset between processes).

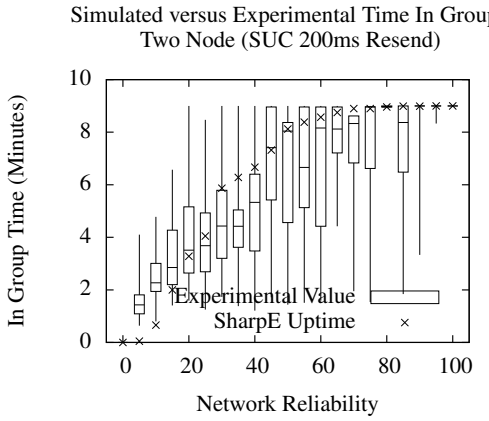

Figure 4.10: Comparison of in-group time as collected from the experimental platform and the simulator (2 tick offset between processes).

4.2. MARKOV MODELS

There was high amount of variance in the collected data. As a result it was difficult to make any sort of prediction about other configurations from the data. Markov chains were employed to model the system.

4.2.1. Initial Model Calibration. The presented methodology of constructing the model was initially calibrated against the original two-node case. This calibration used a non-real-time version of the DGI codebase. The resulting Markov chain was processed using SharpE [29][30] made by Dr. Kishor Trivedi's group at Duke University, a popular tool for reliability analysis. SharpE measured the reward collected in 600 seconds, minus the reward that was collected in the first 60 seconds. Discarding the reward from the first 60 seconds emulated the 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. This synchronization was the time difference between when each of them would search for leaders. The exchange of messages, particularly during an election, had a tendency to synchronize nodes during elections. Nodes could synchronize even if they did not initially begin in a synchronized state. The simulation results aligned best for the 100ms resend case with 1 tick (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 two processes had a poor fit. This is presumed to be a combination of several factors. The major source of fault included the structure of the chain. Construction of the chain assumes that all processes enter the election state a roughly the same time. This was not typically true for more than two processes. Additionally, the simulator could only assume that the synchronization between processes was fixed. 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. The election states with only one outbound transition uncovered a bug in the SharpE software. To circumvent that, issue, SharpE was replaced by a random-walker which generates exponentially distributed numbers and follows the paths of the chain. Time in group data for models which SharpE cannot process were collected across several hundred trials.

The structure of the Markov Chain assumed that processes enter the election state simultaneously. This was an appropriate assumption for the real-time system,

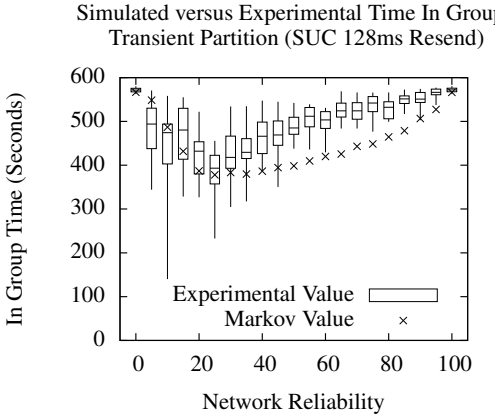

Figure 4.11: Comparison of in-group time as collected from the experimental platform and the time in group from the equivalent Markov chain (128ms between resends).

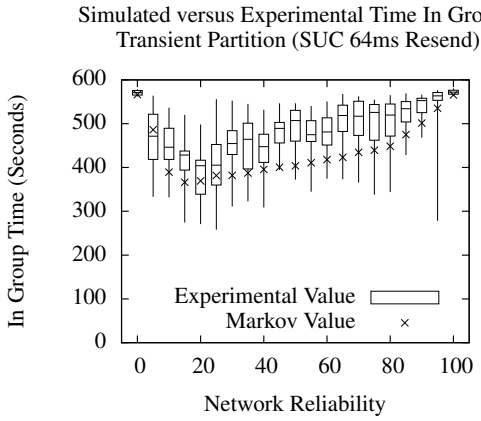

Figure 4.12: Comparison of in-group time as collected from the experimental platform and the time in group from the equivalent Markov chain (64ms between resends).

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

Re-send	Correlation	Error
128	0.7656	11.61%
64	0.8604	11.70%

since the round-robin scheduler synchronized when processes ran their group management modules. The simulator was set to assume that the synchronization between processes was very tight. New experimental data was collected for the 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.

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, 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 test-beds. 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 are the primary focus of this work could cause instability if a sufficient number of these failed exchanges occur. In [6], Choudhari et. al. show that failed transactions can create a scenario where the frequency of a power system could become unstable.

Moving forward, we have identified these areas as targets for improving the research done and creating new contributions.

5.1. SELECTING A SCHEDULE

As shown previously, there is a relationship between congestion and the amount of time a process spends in a group. We have shown that an exponential distribution as part of a continuous-time Markov chain can be used to describe the expected time in a configuration given an omission fault rate. Based on this, we expect to be able

to develop a function which relates the congestion in the network to predict how long a pair of processes will be able to interact. Therefore, when a process observes congestion, either from the network hardware or lost messages they will be able to produce an estimate of their ability to stay in a group with a process. If we consider a set of processes that are in a group together then there is some function which describes the expected amount of time until the group would need to reconfigure:

$$E(c)[X] = \begin{cases} 0 & \text{if } c = 1.0 \\ \infty & \text{if } c = 0.0 \\ 1/\lambda(c) & \text{otherwise} \end{cases} \tag{5.1}$$

Where c is a measure of the congestion in the system: the expected time in group is zero when no messages can be delivered, and infinite when no messages are lost. As congestion is observed by processes, they produce new estimates of λ as a function of the congestion c. The value of $\lambda(c)$ may be based on a series of collected data sampled discretely (using simulation) or may be mappable with a probability distribution.

Additionally, congestion can be used to predict how many migrations will fail. Given a real time schedule and a congestion value c, as with the groups we can define a function $F(c)$ describes the likelihood of a migration failing. These failed migrations contribute to a k-value which can be used in a simplified version of the physical invariant from [6][31][32].

$$\{k < max\_outstanding\_migrations\} \tag{5.2}$$

If the number of failed migrations k exceeds the maximum number of outstanding migrations the system will be unstable. A CPS implementing this invariant will

Table 5.1: Comparison of two proposed schedules, A and B

Schedule A		Schedule B
$E_A(c)[X]$	$<$	$E_B(c)[X]$
$F_A(c)$	$>$	$F_B(c)$
R_{Am}	$>$	R_{Bm}

stop performing migrations when this invariant would be violated. We can definite a related value $remaining\_k$:

$$remaining\_k = max\_outstanding\_migrations - k \tag{5.3}$$

Then we can produce a relationship of when a group is likely to violate the physical invariant in a round, given an upper bound on the number of migrations that will be attempted that round:

$$\{F(c) * max\_migrations \geq remaining\_k\} \tag{5.4}$$

Given this, we can then consider the time that a process may live for, the time it would take to reconfigure, and the likelihood of violating the physical invariant into account as part of selecting a real-time schedule. For example, given that it maybe easier to maintain an existing group rather than elect a new one during network congestion, we can select a new schedule that optimizes the amount of work done and the health of the physical system. Consider a pair of real-time schedules A and B. Schedule A favors high-performance: it expects very little congestion on the network. In exchange, it can complete many more migrations per round (Noted as R_m). Schedule B is slower and safer. It has longer timeout periods that allow for more omission failures. As a result, it can't do a much work: the rate that migrations are performed is lower. The relationship between schedule A and B in terms of the functions $E(c)[X]$ and $F(c)$ in summarized in Table 5.1.

If we define a function $migrations(R_m, t)$ that relates a time period t and the rate migrations are attempted to a number of attempted number of migrations over a time period t. With this function a relationship between the number of failed migrations on schedule A to the lower amount of work produced on schedule B, assuming the migration size is constant:

$$\Delta(A, B) = (F_A(c) * migrations(R_{Am}, t)) - (F_B(c) * migrations(R_{Bm}, t)) \quad (5.5)$$

Based on the above equation, schedule B should be selected when Δ is negative, yielding the invariant for the current schedule x and the set of potential schedules S, which validates when is best to apply a given schedule in a group:

$$\{\Delta(x, y) \geq 0, \forall y \in S\} \quad (5.6)$$

The rate that the system should reconfigure is a function of the maximum number of failed migrations that the system can handle before becoming unstable, 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 a 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. This area is deficient because it is strongly coupled to the interactions with the physical component: we must understand how the cyber configuration and physical changes made by that configuration can affect the system, and establish when reconfigurations should occur to keep the system stable.

5.2. CORRECTNESS OF AN INSTALLED CONFIGURATION

The work presented in this document is probabilistic: the results of a leader election are random and based only on responses arriving with in a specified period of time. Other factors can affect what configurations can be installed such as trust in the parties in the group, the underlying physical topology, and the reliability of the peers in that group. We will develop guards on the properties of a configuration that protect the physical topology and the members of the group. These guarantees would also allow processes to better police the configurations they are installed in, in order to protect the system from malicious nodes.

These guards can ensure quantities like trust in the involved properties, the physical organization for verification methods like attestation. Guards could also ensure the capability of the group to do work: a formed group in which all processes are in supply or demand can do no work. Likewise, if the congestion is too high between a pair of processes, it will affect the ability of the group to do work. Perhaps most importantly, we wish to ensure that a partition in the cyber domain will not cause a connected physical topology to become unstable. Interference between groups in the physical domain should not allow a global physical invariant (P_{IG}) to be violated. We wish to develop guards that ensure when all local invariants are true the global physical invariant is true, where P_{Ix} is the physical invariant for a group x:

$$\bigwedge_{x \in Groups} P_{Ix} \rightarrow P_{IG} \tag{5.7}$$

5.3. ACCURACY AND SCOPE OF THE MODEL

We expect to be able to further refine the model and the algorithms and formulas for generating the model. There are some features of the behavior of the DGI

which are not completely encapsulated in the model. Currently, the arrival times are a continuous time estimation of events that occurs discretely in the real-time system. The failure detection checks occur on a specific interval, but the continuous time Markov chain allows these events to occur at any moment. This limitation reduces the accuracy of the model.

With respect to the way the models are generated with the simulator, the way models are specified can be generalized to support more systems of similar design. Additionally, the simulator uses the resend interval as the smallest timestep, which allows models to be evaluated quickly, but limits accuracy. Lastly, it is worthwhile to adapt the simulation to use a more common network simulation software like OmNet++[18] to make the results more portable.

The models presented in this work focus only on the leader election component of a dynamically configured CPS. Additional work would incorporate additional components of the DGI system into the models for a more complete picture of the behavior of the system during failures. We will consider the correctness of the incorporated algorithms, how omission failures can violate that correctness, and what restrictions we can place on the configuration and operation of DGIs in order to protect the entire system during failures. To do this, we will expand the analysis performed here to incorporate algorithms such as state collection and load balancing and define metrics to quantify their behavior during omission failures. This thrust will pair with the correctness and time between reconfigurations: different algorithms will have different amounts of failure that can be allowed before reconfiguration is necessary.

5.4. DELIVERABLES

Therefore, moving forward, we will expand the models we have presented here to include more of the properties of the complete CPS. This model will allow us to better understand what effects the group behavior has on the CPS. Using this, we can establish invariants which allow us to ensure the correctness of a CPS by providing assertions which will not be broken during execution. Creating these invariants will allow us to improve the development of CPSs, especially in their dynamic configuration, which is an area with limited development. These invariants also allow us to create an assertion of correctness which can be validated, during runtime, to ensure the system maintains its stability. We will continue to create and validate models of the CPS against simulations and actual hardware. As we do so we will construct invariants that describe the correct behavior of the groups to ensure safe operation.

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 *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 760–769, July 2013.

[7] S. Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman & Hall, 2007.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[15] 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.

[16] T. D. Chandra, V. Hadzilacos, and S. 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.

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

[18] OMNeT++ Community. Omnet++, May 2012. `http://http://www.omnetpp.org/`.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] P. Olofsson. *Probability, Statistics, and Stochastic Processes, 2nd Edition*. John Wiley & Sons, 2012.

[25] N. Privault. *Understanding Markov Chains*. Springer Singapore, 2013.

[26] 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.

[27] 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*, 2006.

[28] S. Jackson and B. 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.

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

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

[31] T. Paul, J.W. Kimball, M. Zawodniok, T.P. Roth, B. McMillin, and S. Chellappan. Unified invariants for cyber-physical switched system stability. *Smart Grid, IEEE Transactions on*, 5(1):112–120, Jan 2014.

[32] T. Paul, J.W. Kimball, M. Zawodniok, T.P. Roth, and B. McMillin. Invariants as a unified knowledge model for cyber-physical systems. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pages 1–8, Dec 2011.