MY BIG FAT THESIS

by

STEPHEN CURTIS 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

2016

Approved

Dr. Bruce McMillin, Advisor
Dr. One
Dr. Two
Dr. Three
Dr. Four

ABSTRACT

ACKNOWLEDGMENTS

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

The design of stochastic models of distributed system has a long history as a challenging area of interest. Models of distributed systems have to deal with a number of factors. These factors include the various types of failure the system could experience, a lack of tightly synchronized execution, and a large complex state space when there are a high number of agents[23][5]. However, the concept of distributed systems plays a central role in many of the future visions for how critical infrastructure will operate. This critical infrastructures are physical networks whose operation are so vital that if those networks failed to operate correctly it would be highly detrimental to the population that rely on those systems. Cyber-physical systems (CPS) are the integration of computational systems with physical networks. Computational systems already play a critical role in most critical infrastructures, and as demands for security features such as acessibility increase, distributed systems become an increasingly favorable choice for the computational needs for these systems [53].

The Future Renewable Electric Energy Delivery and Management (FREEDM)[38] smart-grid project follows this vision. The FREEDM center, an NSF funded ERC envisions a future power-grid where widely distributed renewable power generation and storage is closely coupled with a distributed system that facilitates the dispatch of power across those areas. Other systems like Vehicular Ad Hoc Networks (VANET)[31][44][24] and Air traffic control [48][4] also propose similar control systems where many computers must cooperate to ensure both smooth operation, and the safety of the people using those systems. As a consequence, ensuring the behavior of the computer systems that control these infrastructures behaves correctly during failure is critical, especially when those computer systems rely on their interaction with other computers to operate.

A robust CPS 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, processes 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 a smart-grid system, misbehavior during fault conditions could lead to critical failures such as a blackout or voltage collapse. In a VANET or air traffic control system, vehicles could collide, injuring passengers or destroying property. Additionally, since these systems are a part of critical infrastructure, protecting them from malicious entities is an important consideration.

This work was motivated by observations on the effects of network unreliability on the group management module of the DGI used by the FREEDM smart-grid project. These original observations confirmed the need to explore more well defined models for the behaviors of CPS in order for them to better serve the people that use them.

We present a framework for reasoning about inferable state in the context of a distributed system. To do this we exploit existing work in the field of information flow security. Information flow security has been used to reason about how attacks like STUXNET can manipulate operators beliefs while disrupting the system[20]. In particular, these approaches reason about how the operator in a STUXNET attack has no avenue to verify the reports from a compromised computing device. Using existing modal logic frameworks and using information flow security models[26][20][27], one can formally reason about where information that is not normally known to a domain can be inferred.

We will show in this work that in a system with the correct information flows, an agent in a distributed system can infer the state of other agents in the system.

With this information, that agent can then construct a reasonable model of the system to determine if the current behavior could lead to an undesirable situation with either the cyber or physical network. We formalize how this flows are created using information flow security models.

Using this framework we present a leader election that is markov modelable for a set of known omission rates. The presented algorithm maintains the Markov property for the observations of the leader despite omission[16] failures. This approach to considering how a distributed system interacts during a fault condition allows for the creation of new techniques for managing a fault scenario in cyber-physical systems. These models produce expectations of how much time the DGI will be able to spend coordinating and doing useful work. Using these measures, the behavior of the control system for the physical devices can be adjusted to prevent faults, like blackouts and voltage collapse, in the physical network.

We also propose a technique to inform distributed processes of communication network congestion. These processes act on this information to change their behavior in anticipation of message delays or loss. This behavior allows them to harden themselves against the congestion, and allows them to continue operating as normally as possible during the congestion. This technique involves changing the behavior of both the leader election[22] and load balancing algorithm during congestion.

To accomplish this, we extend existing networking concepts of RED, ECN[43], and ICMP source quench[6]. When a network device detects congestion, it notifies processes that the network is experiencing congestion and they should react appropriately. We propose a practical application for these techniques in scenarios where the message delay is a primary concern, not the rate at which data is sent. With this information, they can adjust their behavior to compensate for the detected congestion.

Additionally, we demonstrate an implementation of the FREEDM DGI in a NS-3 simulation environment[15] with our congestion detection feature. The DGI operates normally until the simulation introduces a traffic flow that congests the network devices in the simulation. After congestion has been identified by the RED queueing algorithm, the DGIs are informed. We show that when the congestion notifications are introduced, the DGI maintains configurations which they would normally be unable to maintain during congestion. Additionally, we show a greater amount of work can be done without the work causing unstable power settings to be applied.

2. BACKGROUND

2.1. DISTRIBUTED SYSTEMS

Distributed systems are a computing paradigm characterized by independence of computational units and no universal clock. Components in a distributed system may not directly share computational resources or memory. Instead, computers in a distributed system typically interact through a message passing interface.

As a result, distributed computing is a challenging area of research. In a distributed system, since processes do not share a universal clock, the ordering of messages and events needed to be carefully considered to ensure correct operation of the system. Additionally, since individual components fail, determining which components fail is also difficult in a distributed system. Different types of failures can cause different kinds of information to be withheld from processes or changed to disrupt those processes.

2.2. EXECUTION AND COMMUNICATION MODELS

In this work we consider a distributed system where no processes in the system share an address space. All processes must use a message passing interface in order to communicate with other processes to exchange information. As a result of the complexities of distributed systems, various execution models have been developed to define how the execution of a distributed system proceeds. Different execution models exchange how easy it is to reason about the system's execution and the types of algorithms that can be executed for how complicated they are to implement. We describe the avenues of communication between processes as channels. A channel is classified as reliable if it meets 3 axioms:

Axiom 1. *Every message sent by a sender is received by a receiver and every received message was sent by a sender in the system. [23]*

Axiom 2. *Every message has an arbitrary but not infinite propgation delay.[23]*

Axiom 3. *Every channel is a First In First Out (FIFO) channel. If process P sends a messages x and y to Q (in that order) then Q receives the messages in order (x then y).[23]*

These are often referred to as synchronous channels. In this work however, channels are not assumed to be perfectly reliabile. Instead, we respect Axiom 3, partially fulfill Axiom 1, and disregard Axiom 2. Therefore, we assume the following about communication channels in the systems modelled (replacing Axiom 1)

Axiom 4. *Every message received by a receiver was sent by a sender in the system.*

As well as Axiom 3. Without the constrained propogation delay from 2, this type of communication channel is typically referred to as an asynchronous channel.

Clocks are considered synchronized if every clock in the system reads the same time. Since it is impossible for independent clocks to tick at the same rate, weak synchronization is used to describe when clocks in the system have an upper bound on drift rate from each other.

The communication model can be synchronized or asynchronous. In the synchronous communication model, processes can only send a message when the receiving process is ready to receive it. Algorithmically, sending in a synchronous model is usually considered a "blocking" operation, meaning that once a process tries to send a message, it cannot proceed until the message is received. In this work, communication is asynchronus, meaning that a process does not wait for the successful delivery of a message. This is known as a non-blocking send.

In a system with synchronous processes, processes execute in lockstep. At each step a process executes its next available action. Synchronous execution requires tight

organization of the processes executing the algorithm. In this work we generally rely on semi-synchronous execution by processes where execution proceeds in rounds or phases. The start of each round or phase is synchronized between processes, using a synchronized clock.

2.3. FAULTS, FAILURES, AND ERRORS

Processes can encounter incorrect behavior or issues during execution. Errors, faults, and failures describe the severity and consequence of the issue.

DEFINITION - ERROR

An error is a difference between what is considered "correct" output for a given component, and the actual output– an incorrect result.

DEFINITION - FAULT

The manefestation of an error in sofware, or an incident where a incorrect, step, or data definition is performed in a computer program.

DEFINITION - FAILURE

The inability for a component or system to perform its required function or within its specified limits.

2.3.1. 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 asynchronous 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.[23]

The fail-stop failure has three properties [23]:

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.

2.3.2. Omission Failure. An omission failure occurs when a message is never received by the receiver. Omission failures can occur when the communication medium is unavailable or when the latency of message exceeds a timeout for its expected delivery. Protocols like TCP do not tolerate omission failures: a packet is resent until it is acknowledged by the receiver. If the acknowledgment 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. [23]

2.3.3. Failure Detectors. Failure detectors [11] (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[11]:

- 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 [25]. 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 $\diamond\mathcal{W}$ [11], $\mathcal{W}$ [10] [12] and Ω (Omega) [25].

2.3.4. Byzantine Failures. In a Byzantine failure, processes in the distributed system intentionally send information that is incorrect or misleading to other processes. Contraints for detecting processing that exhibit Byzantine behavior is a famous result in distributed systems.[33]

2.4. PROBABILITY

The expected value represents the long-term average output of a probability distribution.

$$E[X] = x_1p_1 + x_2p_2 + ... + x_kp_k \tag{2.1}$$

Conceptually, the expected value is the weighted average of the outcomes of some stocastic system.

Availability is the probability a system or component is operational and accessible when required for use, denoted A.

$$A = \frac{E[uptime]}{E[uptime] + E[downtime]} \tag{2.2}$$

Reliability is the ability of a system or component, in specified conditions, to be able to perform its required functions for a specified period of time.

$$R(t) = \Pr(T > t) = \int_t^\infty f(x)dx \tag{2.3}$$

Where $R(t)$ is the probability that the system or component functions up until at least time t and $f(x)$ is the probability density function for the component's survival.

2.5. MARKOV MODELS

A Markov chain is a collection of states and probabilistic transitions between those states. States in a Markov chain are mutually exclusive. In a Markov chain, when a system is some state i it has some probability of transitioning to some other state j at the next time-step. A Markov chain is a first order chain if the probability of transitioning from i to j does not depend on the history of transitions that lead to state i. First order chains are described as having a memoryless or Markov property. This formalizes the independence of the next state from the history of previous states. The Markov property describes a Markov chain as a sequence of random variables $X_1, X_2, X_3, ...$ and states the value of X_{n+1} only depends X_n: [7]

$$\Pr(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n)$$
$$= \Pr(X_{n+1} = x \mid X_n = x_n). \tag{2.4}$$

An ergodic Markov chain is a chain where it is possible, in some finite number of steps, to go from any state to any other state. A stationary Markov chain is one where the transition probabilities do not change over time. In a stationary Markov chain, the nth visit to a state is indistinguishable from the $n+1$th visit to a state.

2.5.1. Discrete Time Markov Chain. A Discrete Time Markov Chain (DTMC) is one where the transitions between states happen at discrete time steps. A DTMC with m states can be represented by a $m \times m$ matrix. For simplicity when creating the model, matrices in this work are 1-indexed. In a matrix P, the value of P_{ij} represents the probability of the transition from i to j. It should be obvious the sum of each row in the matrix is equal to one:

$$\sum_{i=1}^{m} P_{ij} = 1. \tag{2.5}$$

A useful companion to the transition matrix is a state distribution vector. While the transition matrix describes how system will transition between states, the state distribution vector describes the probability of observing a given state.

Definition 1. *A state distribution vector is an m-dimensional vector composed of probability of observing each state in the system at a given instant:*

$$[P_1 \quad P_2 \quad \ldots \quad P_m]$$

Where P_i corresponds to the probability of observing state i.

A Markov chain is a suitable model for a memoryless random process with a finite number states which is observed at fixed time intervals. By utilizing a Markov chain, a variety of statistical analyses can be performed on these modeled system. For example, a Markov chain with the stationary and ergodic properties can be analyzed for its steady state probabilities. The steady state is a state distribution vector that describes the probability a random observation of a long-running process will observe some state i. The steady state can by found via a system of equations: [7]

$$0 \leq \pi_j \leq 1.0 \tag{2.6}$$

$$\sum_{j=1}^{m} \pi_j = 1.0 \tag{2.7}$$

$$\pi_j = \sum_{i=1}^{m} \pi_i p_{ij} \tag{2.8}$$

In the following sections, the computation of the steady state will be noted as $Steady()$. A Markov chain can also be used to predict what state a process will be in at some point in the the future. Given a initial state and a number of time-steps a matrix operation will yield the likelihood of the process being in each state after the time interval has passed. The mean passage time, a measure of how many time-steps will pass before a process returns or arrives to some state, can also be calculated.

We model a leader election algorithm with a closed form representation of the behavior of the algorithm. This closed form representation is a profile Markov chain (noted as P). The profile Markov chain is validated against a chain generated from execution of the algorithm. The chain constructed from sampled data is known as a test chain (noted as T).

2.5.2. Continous Time Markov Chain. Transitions in a Continous Time Markov Chain (CTMC) depend on the amount of time spent in a given state. 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).

$$P\{X(s+t) = j | X(s) = i\} = P\{X(t) = j | X(0) = i\} \tag{2.9}$$

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: [42]

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2.10}$$

As a result, the expected or mean value of an exponential distribution, is a function of the parameter λ: [42]

$$\mathrm{E}[X] = \frac{1}{\lambda}. \tag{2.11}$$

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 [39]

$$\sum \lambda(x,y) = \sum \lambda p_{x,y} = \lambda(x) \tag{2.12}$$

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.

2.6. INFORMATION FLOW

2.6.1. Modal Logic. Kripke frames[32][9] play a critical role in the development of this work. The essential concept of this work is that each global state of the distributed system at any given instant can be captured as a countably infinite set of propositional variables. A Kripke frame is a pair $< W, R >$[21] such that W is a set of of possible versions of the global state over time, where each element in W is known as a world. Each element of R describes a binary relationship for how the described system can move from world to world as events occur in the described system.

In the case of a distributed system, a world could be described as one the possible combinations of values of all boolean state variables $S = \{s_0, s_1, ...s_n\}$ in that system. As execution occurs, messages, time, or events cause these variables to change. Each change in boolean variables corresponds to a relationship in R[34]. Therefore, a world w is one possible valuation of all the variables in S and a transition from w to another w' (with its own valuation) can be noted as wRw'. Without loss of generality, each relationship in R must result in the change of at least one variable in S. Additionally, the set of world is complete: every possible combination

Table 2.1: Logical Statement Formulation Rules

1. if φ is a wff, so are $\neg\varphi$, $\Box\varphi$, and $\Diamond\varphi$.
2. if φ is a wff, so are $B_i\varphi$ and $\neg B_i\varphi$
3. if φ is a wff, so are $T_{i,j}\varphi$ and $\neg T_{i,j}\varphi$
4. if φ is a wff, so are $I_{i,j}\varphi$ and $\neg I_{i,j}\varphi$
5. if φ and ψ are both wff, so are $\varphi \wedge \psi$
6. if φ and ψ are both wff, so are $\varphi \vee \psi$

is represented in the set of worlds. No relationship can lead to a world that does not exist.

Additionally, we can define a set of valuation functions, $\{V\}$. Each function $V^i_{s_x}$ in V describes the value observed by an agent i of a boolean state variable s_x. If a valuation function for a particular state variable is not defined for an agent, that agent cannot determine the value of that state variable, and cannot determine the value of any logical statement based on that variable. In the case of a distributed system, this concept is analogous to the isolation of memory for each agent. For example, an agent i, cannot simply determine the value of a variable for agent j.

The combination of a Kripke Frame $< W, R >$ and a set of valuation functions V is a Kripke Model $M = \{W, R, V\}$ sometimes known as a modal model. The complete model describes all the possible worlds, the relation between those worlds and the information available in the domains of the system.

Let $\varphi \in \Phi_0$ be an atomic proposition in a set of countably many propositions. The set of well-formed formulas (wffs) as defined by the formulation rules in 2.2 is the least set containing Φ_0. Additionally, we use the modal operator $\Box$ as an abbreviation for $\neg\Diamond\neg\varphi$. The complete axiomatic system is outlined in 2.1. For the uninitiated, the modal box operator ($\Box$), "it is necessary that" states (in the case of $\Box\varphi$) that in every world w, φ is true. As its dual, the diamond operator ($\Diamond$) states, that it is not the case that in every world, φ is true.

Table 2.2: The Axiomatic System

Definition of logical and modal operators (abbreviations)

D1. $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$
D2. $\varphi \oplus \psi \equiv (\varphi \vee \psi) \wedge \neg(\varphi \wedge \psi)$ (exclusive or)
D3. $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
D4. $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
D5. $\Diamond\psi \equiv \exists w \in W : w \vdash \varphi$
D6. $\Box\varphi \equiv \neg\Diamond\neg\varphi$
D7. $B_i\varphi$ agent i believes the truth of φ
D8. $I_{i,j}\varphi$ agent j informs i that $\varphi \equiv \top$
D9. $T_{i,j}\varphi$ agent i trusts the report from j about φ

Axioms

P. All the tautologies from the propositional calculus.
K. $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$
M. $\Box\varphi \rightarrow \varphi$
A1. $\neg\Box\varphi \rightarrow \Box\neg\Box\varphi$
A2. $\Diamond(\varphi \vee \psi) \rightarrow \Diamond\varphi \vee \Diamond\psi$
A3. $\Box\varphi \wedge \Box\psi \rightarrow \Box(\varphi \wedge \psi)$
B1. $(B_i\varphi \wedge B_i(\varphi \rightarrow \psi)) \rightarrow B_i\psi$
B2. $\neg B_i\bot$
B3. $B_i\varphi \rightarrow B_iB_i\varphi$
B4. $\neg B_i\varphi \rightarrow B_i\neg B_i\varphi$
I1. $(I_{i,j}\varphi \wedge I_{i,j}(\varphi \rightarrow \psi)) \rightarrow I_{i,j}\psi$
I2. $\neg I_{i,j}\bot$
C1. $(B_iI_{i,j}\varphi \wedge T_{i,j}\varphi) \rightarrow B_i\varphi$
C2. $T_{i,j}\varphi \equiv B_iT_{i,j}\varphi$

Rules of Inferrence

R1. From $\vdash \varphi$ and $\vdash \varphi \rightarrow \psi$ infer ψ (Modus Ponens)
R2. $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ (DeMorgan's)
R3. From $\vdash \varphi$ infer $\vdash \Box\varphi$ (Generalization)
R4. From $\vdash \varphi \equiv \psi$ infer $\vdash \Box\varphi \equiv \Box\psi$
R5. From $\vdash \varphi \equiv \psi$ infer $\vdash T_{i,j}\varphi \equiv T_{i,j}\psi$

2.6.2. Non-Deducible (MSDND) Security. In the domain of security there are a wide variety of aspects worth protecting in every system. These are grouped into the core security concepts of integrity, accessibility and privacy. Many traditional security approaches rely heavily on cryptography to provide privacy. However accidental information leakage can still occur, which compromises the privacy of the system. For cyber-physical systems, the leakage is difficult to control. Unlike

their cyber counterparts, the actions taken by the physical components cannot be hidden from a casual observer. For example, a plane changing altitude or a car turning or changing speed cannot be hidden from an observer. Other, more complicated systems, like the power grid, have actions that are more difficult to observe, but a well motivated attacker can potentially collect critical information about the behavior of the cyber components with observations of the physical network[46].

Information Flow security models are invaluable for assessing what information, if any, is leaked by either the cyber of physical components of the CPS. There are a number of information flow security models, all based off similar concepts. Typically, these models partition the system into two domains: the high security domain and the low security domain. However, the MSDND security model allows the system to partitioned into any number of domains. The MSDND model has been used to describe how the STUXNET attack was able to hide its malicious behavior from the operators. The MSDND security model is expressed using modal logic to determine what information in a domain is deducible to an observer in another domain. MSDND security exploits the possible worlds of modal logic to determine if there are worlds where the value of a logical atom is deducible by someone outside the domain.

This information flow security model can be used to determine what an agent in a distributed system can determine about another agent. The exact specification of timing the distributed system becomes unnecessary as the modal model can express any combination of logical atoms in one of its worlds. [26][20][27]

The MSDND security model can be expressed as follows[20]. Consider a pair of state variables s_x and s_y which may or may not be in the same security domain. The value of s_x and s_y have a logical xor relationship: if s_x is true, s_y must be false. Given an agent i that does not have a valuation function for either of those two variables, the system is MSDND secure for that agent and pair of variables. Written formally:

$$MSDND = \exists w \in W : w \vdash \Box[(s_x \vee s_y) \wedge \neg(s_x \wedge s_y)]$$
$$\wedge[w \vDash (\not\exists V_x^i(w) \wedge \not\exists V_y^i(w))] \tag{2.13}$$

Of particular interest is the special case where s_x and s_y are relation on the same wff: ($s_x = \varphi$ and $s_y = \neg\varphi$):

$$MSDND = \exists w \in W : w \vdash \Box[\varphi \oplus \neg\varphi]$$
$$\wedge[w \vDash (\not\exists V_\varphi^i(w))] \tag{2.14}$$

In a system where the above logical relationship holds, the agent i cannot determine the value of s_x or s_y. However, if the relationship does not hold, there is some world where the agent can determine the value of s_x and s_y.

2.6.3. BIT Logic. Belief, Information transfer, Trust (BIT) was developed by such and such to formalize a modal logic about belief and information transfer. BIT logic has typically been applied to distributed systems, but has also played roles in CPS security. The operations of the BIT logic allow formal definition of how entities pass information, and how they will act on the information passed to them. BIT logic utilizes several modal operators:

- $I_{i,j}\varphi$ defines the transfer of information directly from agent j to an agent i.
- $T_{i,j}\varphi$ defines trust an agent i has in a report from j that φ is true.
- $B_i\varphi$ defines the belief that an agent i has about φ. The actual value of φ is irrelevant: the agent i believes it to be true.

These operators allow reasoning about information transference between entities. In the context of a distributed system, these operators allow the division of the actual state held by some agent i to what some other agent j believes agent i's state is.

2.7. ECN

ECN is a techinque for managing congestion in IP networks. When an ECN capable network device detects congestion, it can drop the packets or it can signal senders using flags in the packet headers that the network is congested. For a TCP application, the result of the dropped packets causes the slow-start congestion control strategy to reduce the rate packets are sent. A more advanced implementation, using ECN, sets specific bits in the TCP header to indicate congestion. By using ECN, TCP connections can reduce their transmission rate without re-transmitting packets.

UDP applications have not typically utilized ECN. Although the ECN standard has flags in the IPv4 header, access to the IPv4 header is not possible on most systems. Furthermore, there is not a "one size fits all" solution to congestion in UDP algorithms.

2.7.1. Random Early Detection. The RED queueing algorithm is a popular queueing algorithm for switches and routers. It uses a probabilistic model and an Exponentially Weighted Moving Average (EWMA) to determine if the average queue size exceeds predefined values. These values are used to identify potential congestion and manage it. This is accomplished by determining the average size of the queue, and then probabilistically dropping packets to maintain the size of the queue. In RED, when the average queue size avg exceeds a minimum threshold (min_{th}), but is less than a maximum threshold (max_{th}), new packets arriving at the queue may

be "marked". The probability a packet is marked is based on the following relation between p_b and p_a where p_a is the final probability a packet will be marked.

$$p_b = max_p(avg - min_{th})/(max_{th} - min_{th}) \tag{2.15}$$

$$p_a = p_b/(1 - count * p_b) \tag{2.16}$$

Where max_p is the maximum probability a packet will be marked when the queue size is between min_{th} and max_{th} and *count* is the number of packets since the last marked packet. With RED, the probability a packet is marked varies linearly with the average queue size, and as a function and the time since the last packet was marked. If avg is greater than max_{th}, the probability of marking trends toward 1 as the average queue size approaches $2 * max_{th}$ In the event the queue fills completely, the RED queue operates as a drop-tail queue. In a simple implementation of the RED algorithm, marked packets are dropped.

2.8. DISTRIBUTED GRID INTELLIGENCE

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. Algorithms employed by the DGI and 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 prespecified time-frame to keep the system stable. It uses a round robin scheduler in 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.

The DGI uses the leader election algorithm, "Invitation Election Algorithm," written by Garcia-Molina[22]. This algorithm provides a robust election procedure which allows for transient partitions. Transient partitions are formed when a faulty link inside a group of processes causes the group to divide temporarily. These transient partitions merge when the link becomes more reliable.

2.8.1. Real Time. The DGI's specifications also called for real time reaction to events in the system. These real-time requirements were 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 were 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 2.1

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.

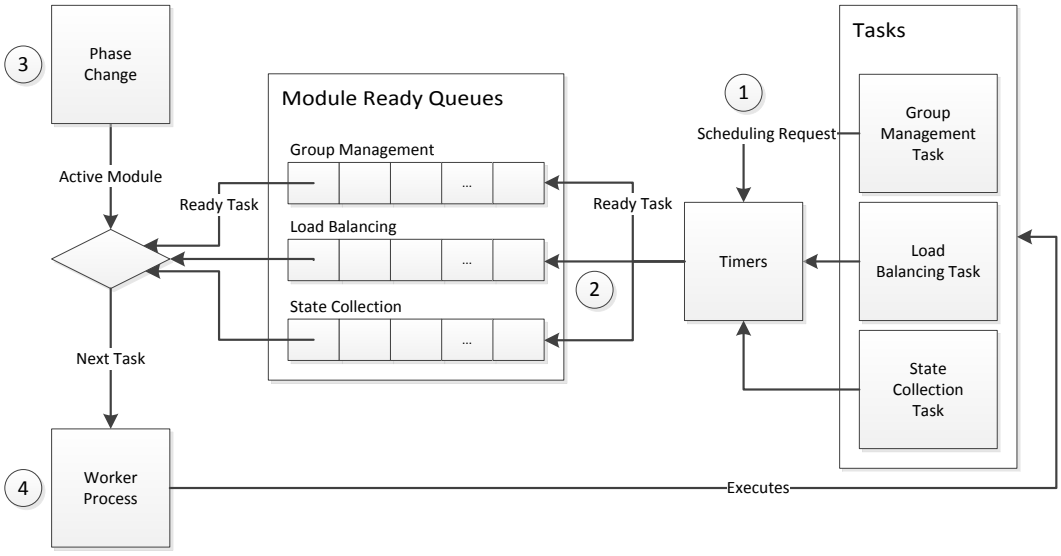

Figure 2.1: The real time scheduler used a round robin approach to allot execution time to modules.

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.

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.

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.

2.8.2. Group Management Algorithm. The DGI uses the leader election algorithm, "Invitation Election Algorithm," written by Garcia-Molina [22]. 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 [22], 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 [49][18], detecting failures in certain circumstances [52][35], and of course, transient partitions. Work on leader elections has been incorporated into a variety of distributed frameworks: Isis [8], Horus [45], Totem [37], Transis [3], and Spread [2] 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 [25]. 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 to 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.

Using a leader election algorithm allows the FREEDM system to autonomously reconfigure rapidly in the event of a failure. Cyber components are tightly coupled with the physical components, and reaction to faults is not limited to faults originating in the cyber domain. Processes automatically react to crash-stop failures, network issues, and power system faults. The automatic reconfiguration allows processes to react immediately to issues, faster than a human operator, without relying on a central configuration point. However, it is important the configuration a leader

election supplies is one where the system can do viable work without causing physical faults like voltage collapse or blackouts[13].

A state machine for the election portion of the election algorithm is shown in Figure 6.1. In the normal state, the election algorithm regularly searches for other coordinators to join with. When another coordinator is identified, all other processes will yield to their future coordinator. The method of selecting which process becomes the coordinator of the new group differentiates the modified algorithm from other approaches.

Processes determine the optimal coordinator (the one with the lowest process ID) through the exchange of Are You There (AYT) and Are You Coordinator (AYC) messages. A process will only accept invites from processes more optimal than itself, and more optimal than its current leader. Once a timeout expires, the coordinator will send a "Ready" message with a list of peers to all processes that accepted the invite. To prevent live-lock the processes do not leave their previous group until the new Ready message arrives, unlike the original "Invitation Election Algorithm." The invited processes have timeouts for when they expect the ready message to arrive.

Once a group is formed it must be maintained. To do this, processes occasionally exchange messages to verify the other is still reachable. Coordinators send AYC messages to members of its group to check if the process has left the group. Group members send AYT messages to the coordinator to verify they haven't been removed from the group, and to ensure the coordinator is still alive. If processes fail to reply to received message before a timeout, they will leave the group. Leaving the group can either be caused by the coordinator removing the process, or the process can enter a recovery state and leave the group, forming a new group by itself.

This version also removes the potential live-lock due to crash failure, although crash-failure is not considered in this work. Other algorithms could be more efficient

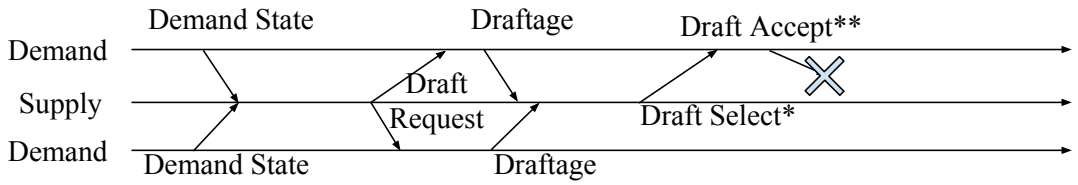

Figure 2.2: Example of a failed migration. (*) and (**) mark moments when power devices change state to complete the physical component of the migration. In this scenario, the message confirming the demand side made the physical is lost, leaving the supply node uncertain.

and less bursty during normal operation, however they are more inconsistent during network congestion and omission.

2.8.3. Power Management. In this work we utilize the load balancing algorithm from [1]. The load balancing algorithm performs work by managing power devices with a sequence of migrations[51]. In each migration, a sequence of message exchanges identify processes whose power devices are not sufficient to meet their local demand and other processes supply them with power by utilizing a shared bus. To do this, first processes that cannot meet their demand announce their need to all other processes. Processes with devices that exceed their demand offer their power to processes that announced their need. These processes perform a three-way handshake. At the end of the handshake, the two processes have issued commands to their attached devices to supply power from the shared bus and to draw power from the shared bus. An example of how the power system is affected by migrations is depicted in Figure **??**. In the chart, processes with net generation (generation ¿ 0) share power with processes with excess loads. As processes with excess loads are satisfied, both supply processes and demand processes trend toward 0 net generation.

The DGI algorithms can tolerate packet loss and is implemented using UDP to pass messages between DGI processes. Effects of packet loss on the DGI's group

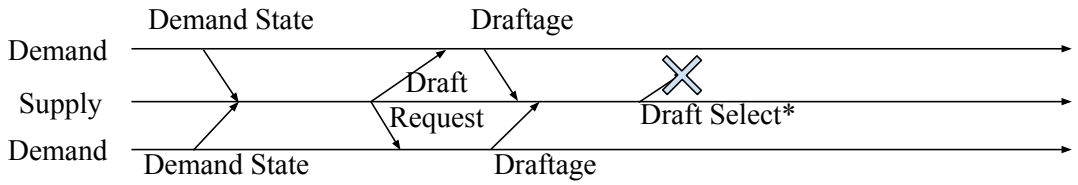

Figure 2.3: Example of a failed migration. (*) marks a moment when power devices change state to complete the physical component of the migration. In this scenario, the supply process changes its device state, but the demand process does not.

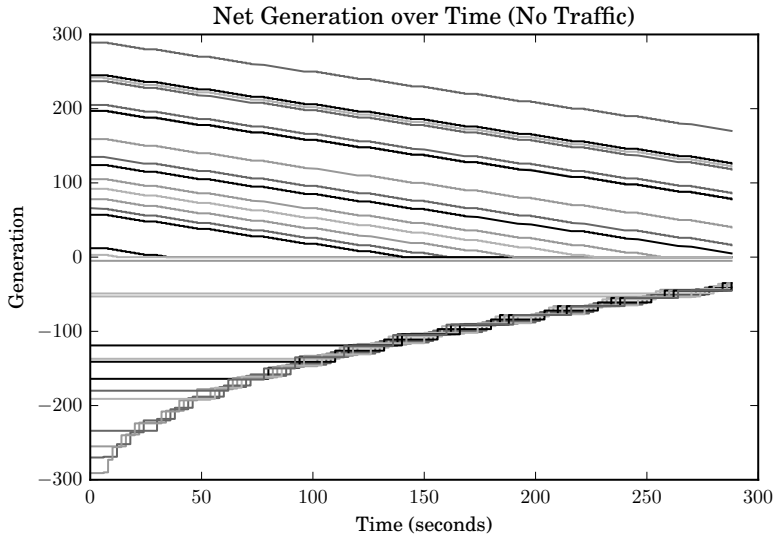

Figure 2.4: Each migration consumes excess generation capability and removes excess demand.

management module have been explored in [28] and [29]. The load balancing algorithm can tolerate some message loss, but lost messages can cause migrations to only partially complete, which can cause instability in the physical network. A failed migration is diagrammed in Figures 2.2 and 2.3. With this power migration algorithm, uncompensated actions may occur in the power system. These actions can eventually lead to power instability through issues such as voltage collapse. Additionally, the supply process may not always be certain if the second half of the action was completed or not. If the "Draft Accept" message does not arrive from the demand process, the supply process cannot be certain of whether or not its "Draft Select" message was received. If the supply process takes action to compensate by reversing the migration and the confirmation arrives later the system will also be driven towards instability because another process completed an uncompensated action. Processes could potentially confirm the number of failed migrations with a state collection technique. It is therefore desirable to manage the processes to minimize the number failed migrations.

3. PROBLEM STATEMENT AND MOTIVATION

The entry point to research in this area was asking the question "How does the software managing critical infrastructure like FREEDM behave when a critical component, the communication infrastructure, is not operating correctly?" Historically, leader elections have had limited applications in critical systems. However, in the smart grid domain, there is a great opportunity to apply leader election algorithms in a directly beneficial way. [1] presented a simple scheme for performing power distribution and stabilization that relies on formed groups. Algorithms like Zhang, et. al's Incremental Consensus Algorithm [54], begin with the assumption that there is a group of nodes who coordinate to distribute power. In a system where 100% up time is not guaranteed, leader elections are a promising method of establishing these groups. A strong cyber-physical system should be able to survive and adapt to network outages in both the physical and cyber domains. When one of these outages occurs, the physical or cyber components must take corrective action to allow the rest of the network to continue operating normally. Additionally, other nodes may need to react to the state change of the failed node. In the realm of computing, algorithms for managing and detecting when other nodes have failed is a common distributed systems problem known as leader election.

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

It is important for the designer of a cyber-physical system to consider what effects the cyber components will have on the overall system. Failures in the cyber domain can lead to critical instabilities which bring down the entire system if not handled properly. In fact, there is a major shortage of work within the realm of the effects cyber outages have on CPSs [50] [53]. In this paper we present a slice of what sort of analysis can be performed on a distributed cyber control by subjecting the system to packet loss. The analysis focuses on quantifiable changes in the amount of time a node of the system could spend participating in energy management with other nodes.

Using the DGI as a starting point, the analysis of the leader election algorithm in the DGI began with analysis of its behavior when messages were lost. To do this, the DGI software was subjected to omission faults while the state of the algorithm was captured over a period of examination. In the goal of these experiments was to examine what behavior the DGI would exhibit during the fault conditions. Additionally, we hoped to determine the advantages and disadvantages of using a more complicated, reliabile message retransmission protocol over a simplier one without retransmission.

3.1. INTIAL EXPERIMENTS

The initial experiments were collected from a non-real time version of the DGI code. Expiriments measured the time-in-group for various sets of DGI running the leader election algorithm during omission fault conditions. Additionally, the experiments examined two communication modes, the Sequenced Reliable Connection, and the Sequenced Unreliable Connection. Both methods of communication were valid approaches for the message passing requirements of the DGI.

3.1.1. 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 is too long. When designing this scheme we wanted to achieve several criteria:

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

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

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

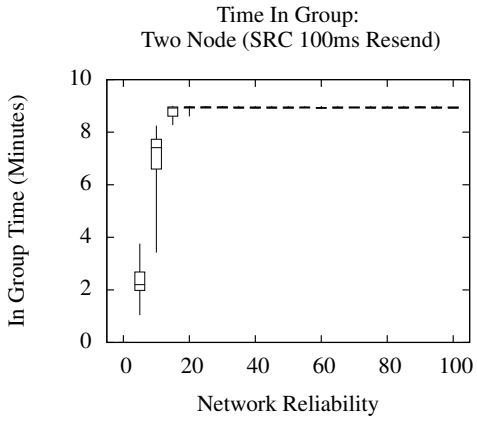

Figure 3.1: Time in-group over a 10 minute run for a two node system with a 100ms resend time

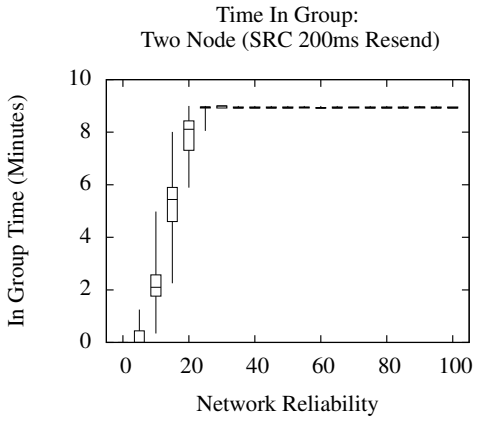

Figure 3.2: Time in-group over a 10 minute run for a two node system with a 200ms resend time

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.

3.2. INTIAL RESULTS

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].

3.2.1. 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 3.1, 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.

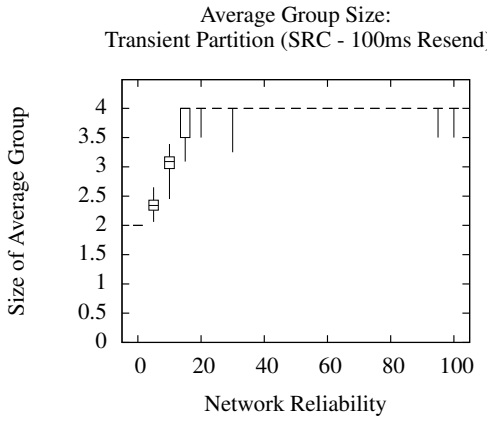

Figure 3.3: Average size of formed groups for the transient partition case with a 100ms resend time

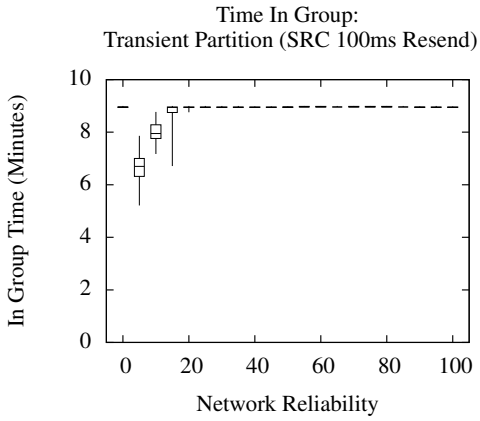

Figure 3.4: Time in-group over a 10 minute run for the transient partition case with a 100ms resend time

Figure 3.2 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 a datagram crossing the partition was increased as the experiment continued. The 100ms case is shown in Figures 3.3 and 3.4.

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

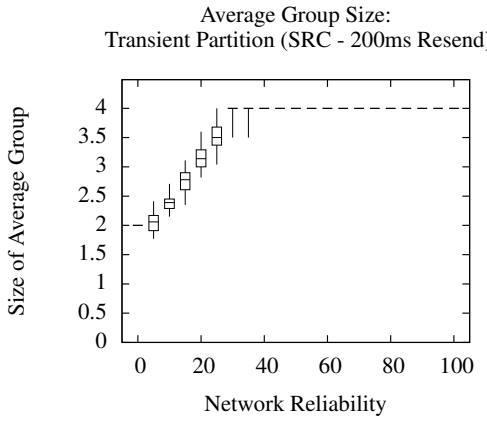

Figure 3.5: Average size of formed groups for the transient partition case with a 200ms resend time

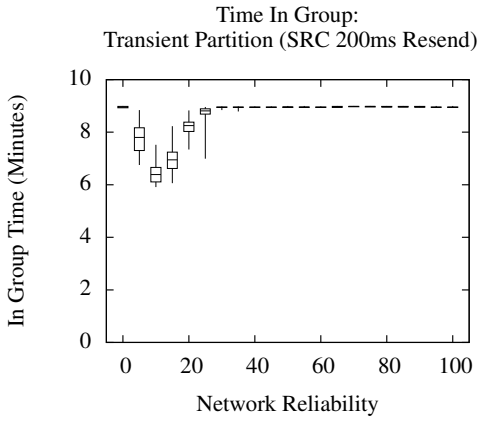

Figure 3.6: Time in-group over a 10 minute run for the transient partition case with a 200ms resend time

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 3.5 and 3.6) suggests a similar behavior to Figures 3.3 and 3.4, with a wider valley produced by the reduced number of datagrams. The mean group size dips below 2 in Figure 3.5, possibly because longer resend times allowed for a greater number race conditions between potential leaders. Discussion of these race conditions is shown and discussed during the SUC section since it was more prevalent in those experiments.

3.2.2. 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 3.7. There is a high amount of variance, however, for any particular trial.

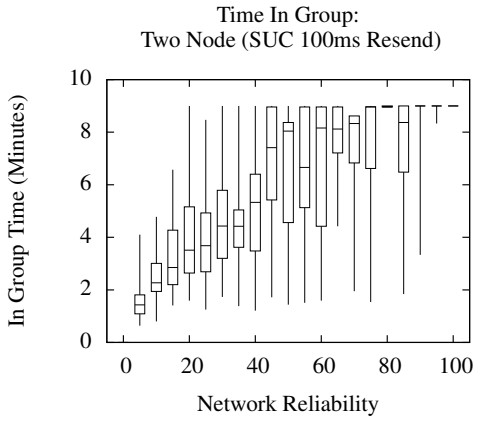

Figure 3.7: Time in group over a 10 minute run for two node system with 100ms resend time

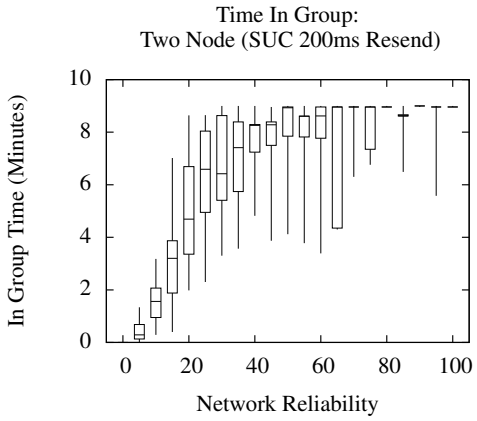

Figure 3.8: Time in group over a 10 minute run for two node system with 200ms resend time

In the 200ms resend case (illustrated in Figure 3.8), a greater growth rate occurred 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.

3.3. MARKOV MODELS

Aftet collecting the results from the intial experiments, we sought to describe the observed behavior through the use of continous-time markov chains. The behavior of the DGI transition between various states of grouping was calibrated with intial results and applied to other scenarios to validate the results. This approach had several short comings. First, for reasons we will demonstrate in subsequent chapters, the leader election algorithm modeled in these chains was not memoryless: the state used in the Markov chain was not sufficent capture the interaction between processes

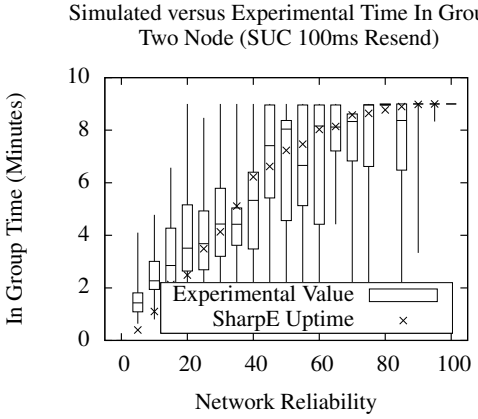

Figure 3.9: Comparison of in-group time as collected from the experimental platform and the simulator (1 tick offset between processes).

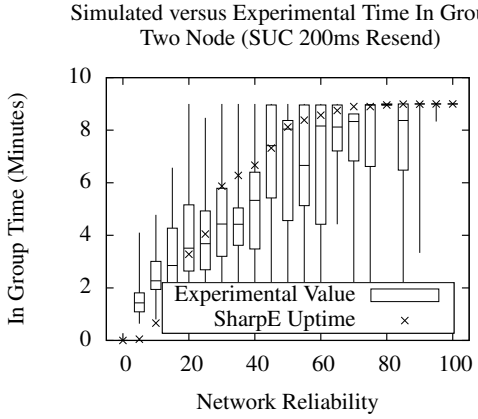

Figure 3.10: Comparison of in-group time as collected from the experimental platform and the simulator (2 tick offset between processes).

that was occuring. Secondly, the continous time model could not accurate capture the execution model of the DGI. In the DGI processes synchronize with each-other to execute in a semi-synchronous manner. As a result, the execution and transition between states is more accurately described with a discrete-time Markov chain.

3.3.1. 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 [30][47] 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 3.9 and 3.10.

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, since the round-robin scheduler synchronized when processes ran their group management modules. The simulator was set to assume that the synchronization between

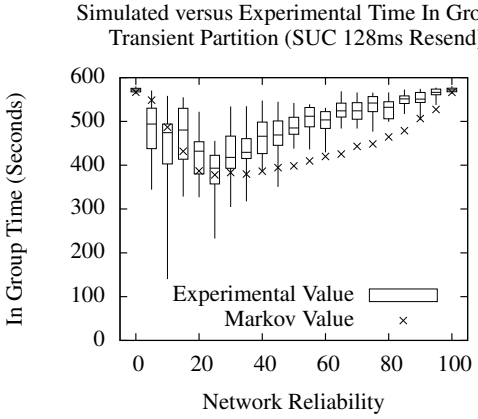

Figure 3.11: Comparison of in-group time as collected from the experimental platform and the time in group from the equivalent Markov chain (128ms between resends).

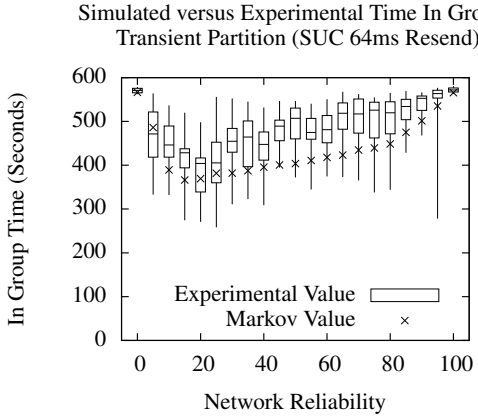

Figure 3.12: Comparison of in-group time as collected from the experimental platform and the time in group from the equivalent Markov chain (64ms between resends).

Table 3.1: Error and correlation of experimental data and Markov chain predictions

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

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 3.11 and 3.12.

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 3.1. These results results were not sufficent to accurately describe the behavior of the system during fault conditions. As a result, we sought to refine the analysis of the model in order to get an accurate portrayl of the behavior of the system during faults.

3.4. REMARKS

In order to ensure that critical infstracture safely and reliabily provides services to those that need the services, it is necessary to understand how the infrastructure behaves during faults. Conceptually, using unvetted critical infrastructure, especially when the infrastructure relies heavily on communication, is the same as stepping into an elevator without a safety brake. Strong analysis of a system's behavior during a failure scenario is paramount to ensuring the safety of those using the infrastructure. We propose knowledge of the correctness of the operation may be more important that the efficency of the operation of that critical infrastructure. Through this work we demonstrate how distributed algorithms can be improved by reasoning about them using information flow security models. By using these models one can analyze where the certainity of the system is placed, as well as ensuring that the algorithms can be modelled without complete and perfect information of the system.

Additionally, we wish to show that these models can be used to allow the models to go beyond use by those administrating the infrastructure. The models created can be analyzed and used while the system is working to provide services ("online"). Using the models created in this way a distributed critical infrastructure system can adjust it behavior, hardening itself against failures that could arise. With this hardening techinque the algorithms can prevent critical failures which could decrease quality of life or endanger that life itself. Subsequent chapters show how information flow methods can be used to determine the memorylessness of aspects of a distributed system, how those aspects can be used to contruct a model, and applications of those models.

4. RELATED WORK

4.1. 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]

4.1.1. 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, 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.

4.2. 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. [36] 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 [36]:

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.

4.2.1. 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 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.

4.3. DISTRIBUTED FRAMEWORKS

4.3.1. Isis (1989) and Horus (1996). A product of Dr. Kenneth Birman and his group at Cornell, Isis [8] and Horus [45] 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.

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.

4.3.2. Transis (1992). Transis [3] 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 acknowledgments, 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 acknowledgment broadcast informing the other members of the view it did not receive that message. In Lansis, the acknowledgment 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.

4.3.3. Totem (1996). Totem[37] 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 multiring 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.

4.3.4. Spread. Spread[2] 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.

4.4. ANALYSIS OF DISTRIBUTED SYSTEMS

[17] focused on examining the emergent behavior as a collection of processes in a grid computing scenario work to process a large dataset. In their work, the authors focus on analysis derived from a discrete time Markov chain discribing a single process completing a task. This chain describes how the process in the grid based system goes through the steps of the process acquiring a task to do, working on that task and subsequently either completing the task or failing. The created chains were “absorbing” chains, meaning that it had one or more states that the process

could not leave once it had arrived in those states (task completed and task failed, for obvious reasons). Their analysis used minimal s-t cuts to determine critical paths for the ideal operation of the sytem. Using this analysis, they considered the Markov chain as a max-flow min-cut problem using the task complete or task failed absorbing states as sinks. By identifying the "critical transitions" of the graph, the areas that would most greatly affect the performance of the system could be identified.

[25] studies an Omega class failure detector using OmNet++[14], 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, [25] only considers the system to be in a complete and active state when all participants have consensus on a single leader.

4.5. PHYSICAL FAULTS CAUSED BY CYBER ENTITIES IN CPS

Faults in CPS can originate from many locations. First and most obviously, the traditional physical system being agumented by the CPS is subject to its own failures, either from component failure or the actions of an attacker. Secondly, the CPS must employ sensors to detect the state of the physical components in the system. Again, like the physical components, these sensors are subject to component failure or the actions of an attacker. Lastly, if the CPS communicates between entities using a communication network, the network can be distrupted by any number of isssues, including DDoS, attackers, or congestion caused by other users in a shared network.

Several works have shown, that for a computer controlled smart grid, that failures originating at sensors or the communication network have the potential to cause the cyber entities controlling the physical networks to take incorrect actions.

Work has been undertaken to identify faulty sensing components in a network, but the identification of bad sensing equipment may not always be possible. Additionally, it is not always possible to identify if the origin of an issue is a faulty sensor or an outside attacker.

If the cyber entity itself has been compromised, it could potentially exhibit Byzantine behavior, causing it to try and trick other components into bad actions, or it may try to disrupt the physical network directly. Work has been done to try and identify when an entity in a cyber network is actively working to compromise the physical network by using the underlying physical invariants. However, even if a cyber entity is trying to behave correctly, disruptions to the communication network or the sensors it uses can cause it to take actions similar to a process that is actively attempting to destabilize the system. As before, these actions can be identifed by using the underlying invariants of the physical network after they have occured. However, it would be ideal to avoid situations where a "good" entity is forced to act badly.

5. INFORMATION FLOW ANALYSIS OF DISTRIBUTED COMPUTING

5.1. METHODS

A model created for a distributed system must have sufficient information to create an accurate model. This information is difficult to obtain because of the circumstances many distributed systems run on. Without exact synchronization, an accurate global snapshot of the system cannot be taken. Instead of attempting to capture exact global snapshots, our approach relies on allowing an agent to reason about the state of the other agents in the system in order to allow that agent to construct a model which can then be distributed to other agents.

To do this we propose the following structure for the execution environment of the distributed system: Each agent has some set logical atoms which it manipulates as its algorithms execute. Each agent belongs to domain unique to that agent (agent i is the only member of logical domain D_i) No agent can directly access a logical atom outside of its domain. The authenticity of any information transfer (Using modal operator $I_{i,j}$) is never not trusted. However, the Trust ($T_{i,j}$) operator is still used to describe a message that is lost in transit: in all logical formulas presented the Trust operator describes this circumstance. agents do not exhibit Byzantine failure, nor do they crash, only messages may be omitted.

If no information is passed between agents, they are MSDND secure (ignoring any sort of leakage from interactions in the physical world). As information is passed, aspects of the agents state are leaked. However, depending on when messages are sent, the agent can be left in doubt as to the state of the other agent. This has a common analogy to the two armies problem.

In order to create algorithms that can be modeled with a Markov chain, first the algorithm must allow at least one agent participating in the algorithm to determine an "Image" of the current state of the algorithm. This image is descriptive enough to allow the agent to determine the probabilistic likelihood at arriving at the the next "image" of the state of the algorithm. Secondly, to create modelable algorithms, we restrict it to a class of algorithms where the sequences of worlds that lead from one image to the next is equally likely.

5.2. TWO ARMIES PROBLEM

First, we will show that information flow analysis can be used to determine what state information is deducible to a particular agent in the system. To begin, we use the common two-armies problem to begin an analysis into what states can be determined in a distributed system.

In the two armies problem, two agents, which are generals of their respective armies must cooperate to attack an enemy city. However, the two armies are physically separated by the enemy city and must send messengers to each other to coordinate their plan. If the generals do not make an agreement on the attack, the attack will fail. However, the generals must come to an agreement when their channel for communication (a messenger) is unreliable (they can be captured by the enemy).

After one message has been sent, to one of the two generals, state of the other is MSDND secure. Let φ_0 be a logical atom that indicates "General A will attack at dawn." For simplicity, we assume that after the time to attack is decided by General A, the agents will not reconsider the plan.

Theorem 1. *If no messages are exchanged, the state of the two generals is mutually MSDND secure.*

Proof: If no messages are exchanged and no information is leaked from the physical world, the two generals have no way of determining the other's state.

Theorem 2. *Once at least one messenger delivers a message to one of the Generals, one of the generals is not MSDND secure.*

Proof: Let $\{\varphi_i : i \in 1, 2...n\}$ describe the state that a general has received φ_{i-1}.

Case 2.1. *One messenger is sent by General A and arrives at General B.*

If no confirmations are sent, then General A clearly cannot deduce if General B has received the message and to General A, then to A, B is MSDND secure because it has no way to valuate $B_B\varphi_0$. However, to B, if B believes A's message then A is not MSDND secure to B, because B believes that φ_0 is true.

1.	φ_0	General A decides to attack at dawn.
2.	$I_{B,A}\varphi_0$	General A sends a messenger to B informing them of their army's intent.
3.	$B_B I_{B,A}\varphi_0 \wedge T_{B,A}\varphi_0$	General B believes the message from general A.
4.	$B_B\varphi_0$	By C1.
5.	$B_B\varphi_0 \rightarrow \varphi_1$	General B knows the plan.
6.	$w \vDash V^B_{\varphi_0}(w)$	$V^B_{\varphi_0}(w)$ always returns true.

Therefore, A is not MSDND secure to B. However, $V^A_{\varphi_1}(w) \notin V$, so B is secure to A.

However, to agent A, $V^A_{\varphi_0}(w)$ is undefined, so MSDND holds in that security domain.

Case 2.2. *Any number of messengers are sent and deliver their message, alternating from General A or General B to the other general.*

As each messenger arrives, the receiving general will trust the message and believe, resulting in that general assigning value to φ_i.

In the case $n = 1$, B is now unsure that A has received φ_1 and cannot deduce if $B_A\varphi_1$. B is unsure of A's state and as a consequence A is MSDND secure to B.

1.	$B_B\varphi_0$	Continuing from Case 2.1
2.	$B_B\varphi_0 \rightarrow \varphi_1$	General B decides to follow A's plan.
3.	$I_{A,B}\varphi_1$	General B sends a messenger to A informing them of their army's intent.
4.	$B_A I_{A,B}\varphi_1 \wedge T_{A,B}\varphi_1$	General A believes the message from general B.
5.	$B_A\varphi_1$	By C1.
6.	$B_A\varphi_1 \rightarrow \varphi_2$	General A agrees.
...		The same logical chain repeats.
7.	$w \vDash V^x_{\varphi_n}(w)$	$V^x_{\varphi_n}(w)$ is always true. x is A or B depending on the value of n.

Therefore, either A or B is not MSDND secure to the other for φ_n.

However, B is not MSDND secure to B because φ_1 is known to A. By extension For $i = 2, 4...n$ B is secure to A, but not A to B. For $i = 3, 5...n$, A is secure to B, but not B to A.

Corollary 3. *Every message exchange where some atom φ_0 is sent, followed by any number n successful exchanges results in one agent being insecure to the other.*

Theorem 4. *If a messenger is captured, if the message is not resent, both agents will be secure on the last successfully delivered message atom φ_n or φ_0 if the first messenger is captured.*

Case 4.1. *One messenger is sent and captured by the enemy.*

It should be obvious and direct that if the messenger does not arrive, it is equivalent to the messenger never being sent. (Theorem 1)

Case 4.2. *If $n-1$ messengers successfully deliver their message, but messenger n is captured, both are secure on φ_n.*

Suppose General A sends φ_{n-1} to B. It should be obvious that on the delivery of the message φ_{n-1} to B, the value of φ_n is secure in B to A, as A has no way of knowing if φ_{n-1} was delivered, unless B sends φ_n with a messenger. When B does send φ_n, the messenger never arrives. As a consequence, General A has no way of

assigning value to $B_A\varphi_n$ ($V^A_{\varphi_n} \notin V$). However, as before, with φ_{n-1} at A is not secure to B.

5.3. BYZANTINE GENERALS

If General A is attempting to coordinate with multiple armies, the problem becomes more complex. If we extend the messenger analogy to cover faulty generals (ones that send incorrect information or omit messengers), the generals can reach consensus if for every faulty general, there are three generals that work correctly. This is a well known result known as the Byzantine Generals problem.

Theorem 5. *In any message exchange that conforms to the constraints of the Byzantine Generals problem, all agents are MSDND insecure on the consensus'd plan φ.*

Proof: Suppose agent i decides to use plan φ to attack. Suppose that there is some set of Byzantine Generals T and some set of loyal generals G ($i \in G$). If $|G| > 3|T|$, the algorithm executes successfully, and $B_x\varphi : \forall x \in G$. Therefore, every general in G can valuate φ and the variable is insecure.

It is worth noting, however, that if all generals intend to behave well (they are non-byzantine) and messages are lost in transit, consensus can only be reached if the initial distribution of φ is delivered to all parties, and each general still receives enough messages to determine the majority consensus. In general, this would be impractical to ensure in an actual application.

State Determination. When an agent uses the information transfer operator ($I_{i,j}$) to pass information to another agent in the system, it intends for that agent to believe the passed wff. When an agent distributes a wff to many agents with the information transfer operator, it leads to a set of beliefs about the beliefs of the receiving agent. This set N_i is the set of beliefs agent i can have if all the wffs it passed to the other agents are believed. For example, if an agent distributes a wff φ

to a set of agents Ag ($i \notin Ag$), then $N_i = \{B_i B_j \varphi : \forall j \in Ag\}$. Since the belief of each $B_j\varphi$ is outside of the domain of i, the agent i can only valuate a wff in N_i which has been leaked to i.

Let the set L_i be the subset of N_i for which a valuation function exists in a domain i. L_i can be populated either by direct information transfer or information leakage from interactions with agents. We can similarly define a set M_i which is the subset of N_i and superset of L_i if the agent i had perfect knowledge of the beliefs of information it passed to other agents ($L_i \subseteq M_i \subseteq N_i$).

Theorem 6. *Each member of L_i is MSDND insecure to i.*

Proof: Each wff in L_i has a valuation function in the domain i.

1.	$I_{j,i}\varphi$	i informs some agent j of φ
2.	$B_j I_{j,i}\varphi \wedge T_{j,i}\varphi$	j receives φ and believes its authenticity
3	$B_j\varphi$	By C1.
4.	$I_{i,j}\varphi_{ack}$	j acknowledges $B_j\varphi$
5.	$B_i I_{j,i}\varphi_{ack} \wedge T_{i,j}\varphi_{ack}$	i recieves φ_{ack}
6.	$B_i\varphi_{ack}$	By C1.
7.	$B_i\varphi_{ack} \rightarrow B_i B_j\varphi$	Because j acknowledged φ, i believes j believes φ.
8.	$w \vDash V^i_{B_i B_j \varphi}(w)$	i does believe φ
9.	$w \vDash V^j_{B_j \varphi}(w)$	Is always true.

Therefore, $j \in L_i$, and $B_j\varphi$ is MSDND insecure to i.

Theorem 7. *Each member of M_i and N_i but not L_i are MSDND secure to i.*

Proof: Each wff in M_i and N_i have no valuation in the domain i.

Case 7.1. *The case where j receives some wff φ and is in M_i but not L_i.*

Case 7.2. *The case where j does not receive some wff φ and is in N_i but not M_i.*

The complete set of atoms and beliefs available in a particular domain may not be necessary to construct the desired model nor may it be the state the model uses. Let $Im(D_i, w)$ define an image function for a particular domain i, and world. The

1.	$I_{j,i}\varphi$	i informs some agent j of φ
2.	$B_j I_{j,i}\varphi \wedge T_{j,i}\varphi$	j receives φ and believes its authenticity
3	$B_j\varphi$	By C1.
4.	$I_{i,j}\varphi_{ack}$	j acknowledges $B_j\varphi$
5.	$\neg(B_i I_{j,i}\varphi_{ack} \wedge T_{i,j}\varphi_{ack})$	i does not recieve φ_{ack}
6.	$w\neg \vDash V^i_{\varphi_a ck}(w)$	i is uncertain if $B_j\varphi$
7.	$w \vDash V^j_{\varphi}(w)$	Is always true

Therefore, $j \in M_i$, and $B_j\varphi$ is MSDND secure to i.

1.	$I_{j,i}\varphi$	i informs some agent j of φ
2.	$\neg(B_j I_{j,i}\varphi \wedge T_{j,i}\varphi)$	j does not receive φ
3.	$w \nvDash V^i_{B_j\varphi}(w)$	i is uncertain if $B_j\varphi$
4.	$w \nvDash V^j_{\varphi}(w)$	j is uncertain of φ

Therefore, $j \in N_i$, and $B_j\varphi$ is MSDND secure to i.

image function produces a state suitable for use in a Markov chain. We assume that in our system, any beliefs an agent hold stem from information transfer from another agent. Therefore, we can assert that the beliefs in L_i for any process i must have a traceable history that stems from a process having a valuation for the referenced atom that aligns with the belief process i holds about that wff. Furthermore, wffs in N_i but not L_i do not have valuation in domain D_i and cannot be used in the construction of the image.

Definition 2. *If for any sequence of worlds $w_0, w_1, w_2, ...w_n$ and pair of images $Im(D_x, w_0) = I_1$ and $Im(D_i, w_n) = I_2$ where I_2 is an immediate successor of I_1 in a Markov chain constructed based on the system. A wff φ is stable for i if after i informs any other process of the value of φ in any world w_k the value of φ does not change in any subsequent world up to, but not including w_n. Expressed logically, $V^i_{w_k}\varphi = V^i_{w_x}\varphi \forall x \in \{k, k+1, ...n-1\}$*

Does this definition work? Suppose we consider the model of our group management system. The image is taken initially in the groupsize=1 state (I_1). Lets skip forward and call φ_j a logical variable that says j is a part of i's group. Ready

acknowledge is $I_{i,j}\varphi_j \rightarrow B_i\varphi_j$ the cardinality of $B_i\varphi_j$'s that i has. This value is not transmitted by i, but if it is stable for j because j doesn't crash, then by the next theorem its also stable for i. After i collects all the φ_j, i takes image I_2. At this point, i can do whatever it likes: it can decide not to believe φ_j at some point and force j to prove φ_j to i again, before it takes I_3.

Theorem 8. *If φ is stable for i, and i is not byzantine and $I_{j,i}\varphi \rightarrow B_j\varphi$, then $B_j\varphi$ is stable for j.*

Proof: Since φ is stable, the output of the valuation function for that wff at i does not change after i has informed j of φ. Since i is not byzantine, i can only transfer the same value to j as it did in w_k. Additionally, the restriction $I_{j,i}\varphi \rightarrow B_j\varphi$ precludes j from arbitrarily changing belief. Therefore, once j believes φ it will continue to believe φ until at least w_n.

Definition 3. *A process is stable if for all wff's used in $Im(D_i, w_0)$ every wff is stable.*

Any action taken by agent i's algorithm based on L_i or N_i by the algorithm should consider two components: N_i at worst overestimates the state and L_i at worst underestimates the state. In a consensus algorithm, the consequences of an overestimate are often worse than those of an underestimate. In the case of an underestimate, the agent distributing the atom can simply redistribute the atom to those agents again. If the algorithm moves forward with the overestimate N_i, those agents not in M_i can only advance if the algorithm does not depend on $B_j\varphi$ or the receiving agent can deduce φ from subsequent messages, without interrupting another agent's operation. However, that deduction can only occur if additional information arrives at the agent.

This feature is necessary for constructing useful models. A good model for online analysis in a distributed system should be simple, to limit the size of the state

space: complete exact knowledge of another agent's state is impractical. If the model is constructible by an agent it should rely on information which can be inferred by information leakage from an MSDND insecure agent. Any knowledge of an agents state, inferred or otherwise by an agent should have some permanence to make the model created suitable for any sort of long term analysis.

For example, if an agent i can infer some state for agent j, the state that i infers about j will remain the state of j for an amount of time sufficient for whatever predictive needs j uses the inferred information for. Of course, if agent B crashes, and the value was stored in volatile memory then the information inferred by A is no longer correct. Therefore, crash failures are largely ignored in this work.

Theorem 9. *For any pair of worlds w_1 and w_2 where $Im(D_i, w_1) = I_1$ and $Im(D_i, w_2) = I_2$ if I_2 is an immediate successor of I_1 in the constructed model, the model will have the memorylessness property if for every world $w_k \in W$ where $Im(D_i, w_k) = I_1$ and every world $w_l \in W$ where $Im(D_i, w_l) = I_2$ every possible sequence of worlds that goes from w_k to w_l is equally as likely as the sequence that leads from w_1 to w_2.*

If this theorem is true, then the arrival at an imaged state I_2 depends only on the worlds which yield the Image I_1. Since the world contains the complete depiction of the state of the system, it is necessary that any additional information not used in the imaging function does not affect the likelihood of arriving at the next image in the model.

Let $\Pr(w_i R w_{i+1})$ be the probability of transitioning from world i to world $i+1$. Let $\Pr(w_i R w i+1 \wedge w_{i+1} R ... \wedge ... R W_n)$ be the probability following a sequence of worlds from some world i to some world n. There may be multiple sequences that lead from world w_i to world w_n. Let $\Pr(w_i \rightarrow w_n)$ be the combined probability of any sequence of worlds that leads from a world w_i to the first encounter of world w_n.

It is worth noting then that with a mapping of the probability of transitioning between worlds, the Kripke Model itself becomes a Markov Chain. Since the Kripke

model is complete representation of the system being modelled, each world must be memoryless wrt to any world that has an R relation onto that world ($w_x R w_y$). If there was any information that would affect the probability of transitioning to the next state in the world, the world would not be complete, and would violate the neccessary completeness property of the model. Some actions may, by necessity, be determinstic as the R relation of the Kripke model obligates that only one state variable may change between worlds.

An imaging function for a Kripke-Markov model is then actually an application of lumpability CITE. The "lumps" produced by the imaging function are restricted to the class of information that is MSDND insecure to an agent in that domain. By necessity of being insecure, all information used in an image must have a valuation function for the domain the image is being taken from.

To hold the memoryless property, given a pair of images I_1 and I_2, where I_2 is an immediate successor of I_1 in the Markov chain. Let $W_1 = \{w_x | Im(D_i, w_x) = I_1\}$ and $W_2 = \{w_x | Im(D_i, w_x) = I_2\}$. $I_1 \neq I_2 \rightarrow W_1 \cap W_2 =$, since the images must not be equal. Furthermore, there exists a sequence of worlds for every $w_x \in W_1$ that leads to a member for W_2.

Then, for every world $w_x \in W_1$ and for every $w_y \in W_2$ that can be reached from w_x before any other member of W_2, $\Pr(w_x \rightarrow w_y) = \Pr(w_x` \rightarrow w_y`)$. Where w'_x and w'_y are another pair of world that fulfill the same requirements for w_x and w_y.

Additionally, for any $\Pr(w_x \rightarrow w_y)$, $\Pr(w_x \rightarrow w_y) = \Pr(w_x \rightarrow w_y | w_z R ... R w_x)$, where w_z is some other world that can reach w_x through some sequence of worlds. As a consequence of this property the probability of encountering a sequence of the transitions is equally as likely regardless of the sequence that led up to that point.

Corollary 10. *Any sequence of worlds described by a Kripke Model that is complete fulfills the obligation that* $\Pr(w_x \rightarrow w_y)$, $\Pr(w_x \rightarrow w_y) = \Pr(w_x \rightarrow w_y | w_z R ... R w_x)$

Proof by contradiction: If with this property, the model is not memoryless, then there must be information not contained in the worlds that affects the next world in the sequence of the system. However, the world must be complete, and if the there is information that is not in the world, the world must not be complete.

For the sequences of worlds that connect the two worlds w_x and w_y, we suppose there are two reasons for the differing available paths: 1) global message and execution ordering and 2) probabilistic logical departures. For type 1, we generally ignore these effects: messages that arrive within a deadline imposed by the agent running the algorithm, or the global total ordering of reactions to messages are not important. We define these then as equally likely transpositions of single path. The second type can be avoided by limiting non-determinism in the algorithms, and avoiding execution decisions influenced by type 1 sequences.

Theorem 11. *Any process that acts based on N_i cannot be memoryless, unless the imaging function produces the same image for every world or no messages are lost.*

Case 11.1. *The case where the imaging function produces the same image for every world.*

If the imaging function produces the same image for every world, there is no other image to transition to and every transition must arrive at the same image.

Case 11.2. *The case where $L_i = M_i = N_i$ (No messages have been lost).*

If no messages are lost, the sequence of worlds that a process goes through between states depends only on the ordering of events in the system.

Case 11.3. *The case where $L_i \subsetneq M_i \subsetneq N_i$*

Since the output of the imaging function is limited to the knowledge of the domain of a given process, the output of the imaging function does not assign value to wff's in N_i but not L_i. Since these wff's have no value they cannot be used to

construct a valid image, as the image may not correspond to a world w in the set of worlds W.

5.3.1. Example With Two Armies. If we enforce a requirement that a General cannot reconsider a plan (in this case, φ_0) because they are stable. Then, while the receiver of the last message, φ_n, can construct the same model of the probability to attack as the sender, the recipient can construct a more accurate model given that the message delivery event has occurred. In fact, if the sender is committed to a proposed plan, for this problem the recipient the recipient can be certain the attack will succeed.

Instead, consider a simple algorithm to reach consensus for a system with no byzantine generals, but with omission failure (message loss). Again, each message has the probability p of being delivered. General A distributes the plan φ_0 to k other generals. If General A expects no confirmations, the probability that the message was delivered to all k generals is p^k. If the first message is delivered, the receiving agents can construct the same model as A if they know how many generals there are and the probability that the messages are delivered.

However, for a longer algorithm with multiple exchanges and states, it is not sufficient to use the consensus probability to determine the state, for an overall view of the system because the state of the system is a probability distribution and not a fixed state.

Instead we obligate the design of the algorithm to rely on its underestimate of the system state L and not on the overestimate for determining the actions taking back the algorithm in the next step. For a leader election this is a good construct: with the correct message passing, the other agent can immediately infer that it was not a part of L if it receives φ_0 again.

Consider the simple consensus algorithm from before. While all the agents that receive φ can construct the same model, that model is not sufficient for any

agent to determine the state of the system. However, if all the agents are stable, and a confirmation message $\varphi_{1,i}$ is sent by each agent that received φ_0 then the original sender knows for certain the state of each agent that sent $\varphi_{1,i}$. Therefore, the original agent (a) is certain of a portion of the agents ($i = 1, 2...n$) that have leaked $\varphi_{1,i}$ to it, assuming that $B_a\varphi_{1,i} \rightarrow w \vDash V^i_{\varphi_{1,i}}(w)$ for the current world, w.

However, this approach is limited because of the set of $\varphi_{1,i}$ that a believes is potentially a subset of all the agent that have a valuation function for $\varphi_{1,i}$ if any of the messages are lost in transit. Based on the information flows presented here and previously, if there is no response mechanism in the algorithm, after the transmission of φ_0, a has a set of agents which believe might have received φ_0, but can make no determination of the actual state of the system. If there is an acknowledgment mechanism then a can determine a subset of the total system state.

Constructing a model based on a subset of the complete state may seem like a mistake, but recall that A does not believe that those other entities believe φ_0. Assume a simple system where A is attempting to ensure every agent believes φ_0. If the actions of the agents are divided into discrete steps, then each step A distributes φ_0 to each of the agents that A has not received $\varphi_{1,i}$ from. A markov chain can easily be constructed for the probability of eventually arriving at state where every agent believes φ_0. Indeed, this algorithm is a special case of the coin flipping leader election algorithm, an algorithm for leader elections in anonymous networks.

Each round, the transfer of φ_0 to active agents is line 3 of the algorithm. The receive step is equivalent to the expected response. The other agent's $b(i)$ value is decided by the receipt of φ_0 from A. Agents go idle on the receipt of φ_0, as if they had randomly selected a 0. If the other agents do not receive φ_0 it is logically equivalent to them holding a 1. If their confirmation is not received by A, it is logically equivalent to that agent successfully sending a 1 to A. However, this approach has the advantage of being able to evaluate, (given the omission rate) approximately how many rounds

Algorithm 1 Anonymous Coin Flipping Leader Election

1: $b(i) \leftarrow random(0, 1)$
2: Send b to every active neighbor
3: Receive b from every active neighbor
4: **if** $b(i) = 1 \wedge \forall j \neq i : b(j) = 0$ **then**
5: i is the leader.
6: **else if** $(b(i) = 1 \wedge \exists j \neq i : b(j) = 1 \vee (\forall k : b(k) = 0))$ **then**
7: $b(i) = random(0, 1)$
8: Go to next round
9: **else if** $b(i) = 0 \wedge \exists j : b(j) = 1 : b(j) = 1$ **then**
10: Become passive.
11: **end if**

it will take for every agent to acknowledge φ_0. This construction is presented below in Figure 5.1, which shows the structure of the chain A could generate for the election. In the figure, each state records the set of "passive" processes.

5.4. ELECTION IN AN ANONYMOUS COMPLETE NETWORK

Consider a version of the "coin-flipping" leader election, expressed in terms of BIT logic. This algorithm is functionally identical, but contains additional gaurds on the conditionals as an additional expression of the knowlege of the agents executing the algorithm. Additionally, note that the construct of labeling each φ that an agent receives is simply assigning labels to make the algorithm easier to reason about. The algorithm only considers the complete collection of φs for execution and not the source of any of the φ used.

Additionally, let ψ_i be the state where an agent completes the algorithm as the leader, and γ_i is the state where the an agent has terminated the algorithm. Correct operation is where $\{\gamma_i : \forall i\}$ is true in the same round of execution and exactly one ψ is true ($\sum_i \psi_i = 1$).

Theorem 12. *The coinflipping algorithm may not terminate correctly unless there is detectable, perfect information transfer to all parties in the algorithm.*

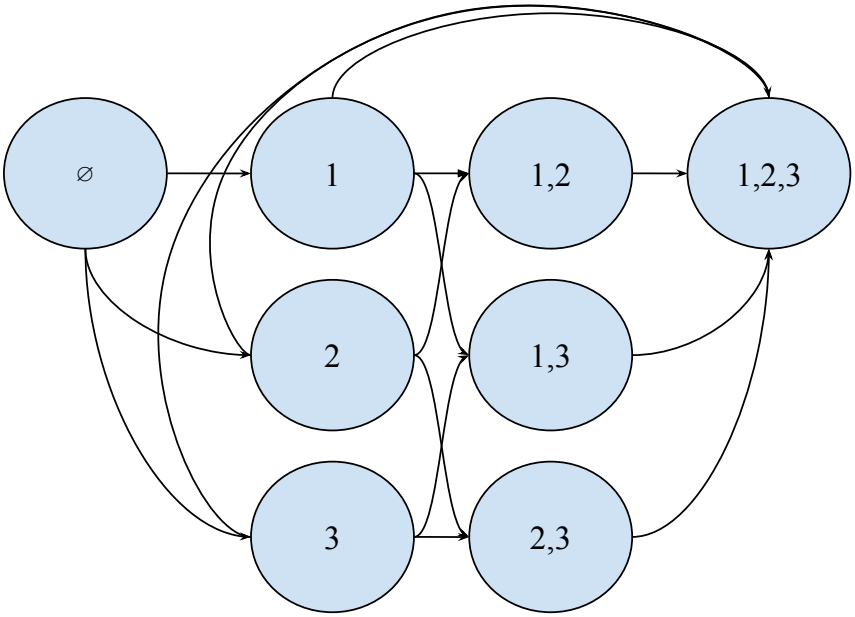

Figure 5.1: Example of a Markov chain constructed for an election algorithm from information flows.

Algorithm 2 Anonymous Coin Flipping Leader Election Expressed in BIT logic

$\varphi_i \leftarrow random(T, F)$
$\psi_i \leftarrow F$
Send φ_i to every active neighbor $(\forall j \neq i \ : I_j, i\varphi_i)$
Receive φ_j from every active neighbor
if $\varphi_i \wedge (\forall j \neq i, w \vDash V^i_{\varphi_j}(w) : \neg\varphi_j)$ **then**
    $\psi_i \leftarrow T$
    $\gamma_i \leftarrow T$
else if $(\varphi_i \wedge \exists j \neq i, w \vDash V^i_{\varphi_j}(\varphi_j)(w) : \varphi_j) \vee (\forall k \text{ s.t. } w \vDash V^i_{\varphi_k}(w) : \neg\varphi_k$ **then**
    $\varphi_i \leftarrow random(T, F)$
    Go to next round
else if $\exists j \neq i, w \vDash V^i_{\varphi_j}(w) : \varphi_j$ **then**
    $\gamma_i \leftarrow T$
end if

Proof: Let i be the agent that would correctly terminate as the leader. Let j be a process that has selected $\varphi_j = F$.

1.	$\varphi_i = T, \varphi_j = F$	Initial conditions
2a.	$I_{j,i}\varphi_i$	i sends φ_i to j
2b.	$I_{i,j}\varphi_j$	j sends φ_j to i
3a.	$\neg(B_j I_{j,i}\varphi_i \wedge T_{j,i}\varphi_j)$	j does not receive φ_i.
3b.	$B_i I_{i,j}\varphi_j \wedge T_{i,j}\varphi_i$	i receives φ_j.
4a.	$\neg w \vDash V_{\varphi_i} j(w)$	j cannot valuate φ_i.
4b.	$B_i\varphi_j$	By C1.
5a.	$\nexists V^j_{\varphi_i}(w) \wedge \neg\varphi_j \rightarrow (\varphi \leftarrow T)$	j Cannot determine that i can terminate and incorrectly changes φ.
5b.	$\varphi_i \wedge w \vDash V^i_{\varphi_j}(w) \wedge \neg\varphi_j \rightarrow \psi_i$	i incorrectly terminates as the leader.

Therefore, i will terminate before j decides to go passive.

Which agrees with results from similar analysis CITE. As with the Byzantine Generals problem, algorithms similiar to Byzantine agreement can be constructed for the anonymous networks. These algorithms terminate if the number of agents that omit information are less than a specific threshold.

6. ALGORITHM AND MODEL CREATION

6.1. EXECUTION ENVIRONMENT

Execution occurs in a real-time partially synchronous environment. Processes synchronize their clocks and execute steps of the election algorithm at predefined intervals. Processes with clocks that are not sufficiently synchronized cannot form groups. For this work, process execution occurs in an environment where the clocks are sufficiently synchronized to consistently form groups.

The execution environment is subject to omission failures. In an omission failure, a message sent by one process to another process is lost in transit. Omission failures can occur for many reasons. Network congestion is a typical culprit. Routers may drop packets or delay their delivery when there is a large amount of traffic or a network issue. In a real-time environment, messages that are delayed and miss their real-time deadlines will have the same appearance as an omitted message. Process delay can also cause a similar effect. The execution environment for the election algorithm has a omission failure occurrence modeled as a Bernoulli trial. In this model, each message has some probability p of being delivered within the timing constraints imposed by the real-time schedule. For the purpose of analyzing the effects of omission failures, processes are not subject to other faults.

6.2. ELECTION ALGORITHM

A state machine for the election portion of the election algorithm is shown in Figure 6.1. In the normal state, the election algorithm regularly searches for other coordinators to join with. When another coordinator is identified, all other processes will yield to their future coordinator. The method of selecting which process becomes

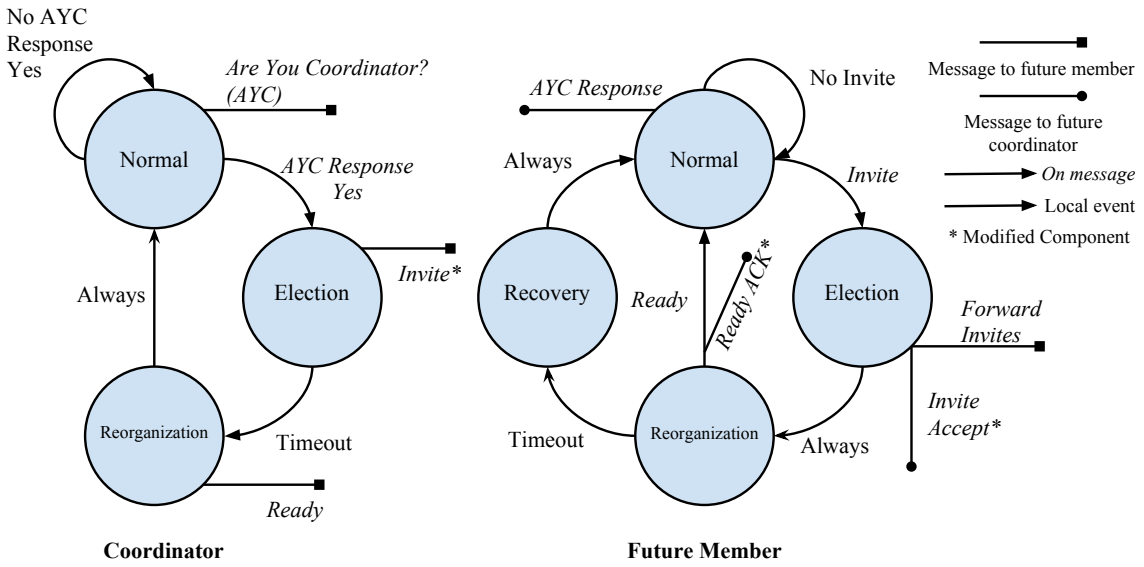

Figure 6.1: State machine of a leader election. Processes start as coordinators in the "Normal" state and search for other coordinators to join with. Processes immediately respond to "Are You Coordinator" (AYC) messages they receive. The algorithm was modified by adding a "Ready Acknowledgment" message as the final step of completing the election. Additionally, processes only accept invites if they have received an "AYC Response" message from the inviting process.

the coordinator of the new group differentiates the invitation election algorithm from other approaches. Regardless of the algorithm, the selection of the coordinator is always a race-condition and is difficult to model.

In the invitation election algorithm, processes are assigned a priority based on their process ID. The coordinator with the highest priority is the first to send invites. After a brief delay, if it appears that coordinator did not send their invites, the next highest process will send their invites. Coordinators that receive invites will forward the invite to its group members. Those processes will accept the invite. Once a timeout expires, the coordinator will send a "Ready" message with a list of peers to all processes that accepted the invite. The invited processes have timeouts for when they expect the ready message to arrive. If the message does not arrive in time, the process will enter the recovery state where it resets to a group by itself.

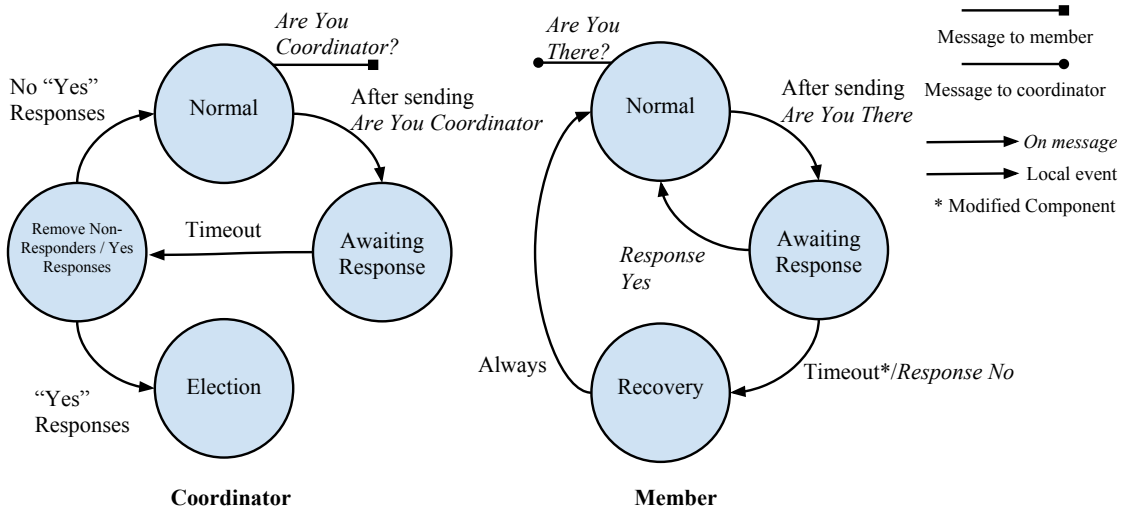

Figure 6.2: State machine of maintaining a group. The "Are You Coordinator" (AYC) messages are the same as those in Figure 6.1. AYC and "Are You There" (AYT) are periodically sent by processes, and responses to those messages are immediately sent by the receiving process. In the modified algorithm, the member does not enter the recovery state if they do not receive an AYT response before the timeout expires.

Once a group is formed it must be maintained. To do this, processes occasionally exchange messages to verify the other is still reachable. This interaction is shown in Figure 6.2. Coordinators send "Are You Coordinator" messages to members of its group to check if the process has left the group. Group members send "Are You There" messages to the coordinator to verify they haven't been removed from the group, and to ensure the coordinator is still alive. If processes fail to reply to received message before a timeout, they will leave the group. Leaving the group can either be caused by the coordinator removing the process, or the process can enter a recovery state and leave the group, forming a new group by itself.

For a variety of reasons modeling this algorithm is very difficult. In a distributed system information cannot be instantaneously spread throughout the system. As a consequence, an omission failure will cause a process to miss out on information distributed by other processes. In particular, an omission of a "Ready" message

sent to group members cannot be known to the coordinator without an additional message. This missing information causes the coordinator's observation of the group to be out-of-date. If the observed state of the group is incorrect, it cannot be used in a Markov chain with perfect information. When the coordinator's observation of the group is out-of-date, a state observed by the coordinator would conflict with one observed by a member. In this case, the state observed the member is closer to the true state of the system. This situation is not favorable to modeling the formation of groups. This is true because, for the purpose managing the group, the observations of the coordinator are much more valuable than those of the members.

In the modified algorithm, members have uncertainty with respect to their membership, and the coordinator has perfect information about members that definitely are a part of the group. The original "Are You There" approach for detecting failure (Figure 6.2) also causes a similar effect. Additionally, the election algorithm has an inherent race condition as part of its normal operation. This race condition can make the selection of the group leader inconsistent: the highest priority process may not always win the election. For the purpose of analyzing the system, having a consistent leader is important for capturing a consistent view of grouping. Additionally, the resulting leader of a normal election is difficult to model since it would necessitate capturing a race-condition in the model.

In this work, we modeled what a process will observe as a result of omission failure. Therefore, it is important a process's observations hold to the Markov property. The original Garcia-Molina algorithm has been modified so the observations of the coordinator process have the Markov property. The following sections state the portions of the algorithm where the observation of the leader process does not yield the probability of next transition. Changes to the algorithm relieved race conditions when selecting a leader, and made the state of the members a certainty for the coordinator.

Analysis is presented in the context of the FREEDM DGI. The FREEDM DGI is a portion of the FREEDM project focusing on creating a distributed intelligence for controlling power electronics. As a consequence of managing the physical network of a critical infrastructure, the DGI must execute in real-time: certain actions and tasks performed by the DGI have to be completed in a specific time from.

To organize these agents, the system relies on existing clock synchronization techniques to put the system into a semi-synchronous execution environment. The DGI is divided into several modules which focus on various tasks like autonomous configuration, managing power devices and collecting data from the system. DGIs rely on synchronized clocks to begin the execution of the various tasks at the same time. This allows the message passing portions to specify deadlines to ensure that well-behaved participating agents are all working on similar tasks at the same time. For example, under the execution model as part of the autonomous configuration module, all agents begin the Leader Election Invitation Algorithm at the same time.

When reasoning about the system within the structure of the Markov Chains and MSDND security, this is a boon. An agent which is too out of sync will not be able to participate effectively and so is ignored for the purpose of this work. This effectively limits the state space of the complete naive model (as all agents are in roughly the same step of the algorithm). Furthermore, an agent can be sure that if message deliveries are timely (which is enforced by the real-time constraints) they can be certain of the lifetime of the inferred state.

This execution model has one major benefit: if, in order to participate in the algorithm, the agent must synchronize their actions (even if that synchronization is loose), the lifetime of a model is well established. In the case of a leader election algorithm, the agent that becomes the leader and constructs the model of the lifetime of the group can know for certain that another agent cannot come online and attempt to take over it's group until the next round of execution begins.

In the following portions of the paper we present an analysis of the information leakage for the original Garcia-Molina Invitation election algorithm and present how our modified version uses the MSDND information leakage to place certainty at one agent in order to construct a Markov model. In particular, we wish to use the MSDND model to show that the changes show that the coordinator can infer if each agent in the group considers itself a part of that agent's group.

Since any coordinated actions by a group must be initialized by the coordinator, having a model of the system constructed by the coordinator is the most valuable in the context of the system. The model constructed by this approach follows the L state of the system. The L state logically underestimates the state of the system, which is more accurate with respect to determining the current state for the next step of the algorithm. For the formulas presented, the coordinator of each group constructs a model of the system based on the cardinality of the the L set based on the information flows in the algorithm.

6.3. GROUP MAINTENANCE AND LEADER DISCOVERY

The original and modified Garcia-Molina invitation election algorithms both use a common set of variables at each agent to track the state of the system.

Additionally, both versions use a recovery algorithm to abandon failed elections and groups.

Members cannot leave a group without the leader's permission. Members do not suspect the coordinator has failed, only the coordinator may suspect the members. This change is shown in Figure 6.2, denoted by the elements marked with an asterisk. For the purpose of starting an election, an "Are You There" message and its negative response are considered equivalent to a "Are You Coordinator" message and a positive response. On receipt of the negative response, the member will immediately recover

and become a leader. This assumption relies on "Are You Coordinator" and "Are You There" messages being sent at roughly the same time. This change leads to a live-lock situation in a crash failure, where the group's leader crashes and does not return and as a consequence the remaining members are trapped in a group without a leader. For the purpose of this work, we have disregarded these live lock scenarios.

6.3.1. Original Algorithm. The AYC message serves as a failure detector for crashed agents. In the models presented in this paper, crash failures are ignored as part of the modeled system.

The AYT message serves as a check for the "Ready" message sent by the group leader at the end of the election algorithm. AYC verifies the leader considers the sending process a part of its group. A negative response to the AYT message would preven a process from participating in the current round of leader elections. This violated the memorylessness property as a process would be excluded from the current round of elections based on an event that occured the previous round.

6.3.2. Modified Algorithm. Algoritm 3 shows the modified components of the Invitation Election algorithm used for group maintenance and leader discovery.

In the maintenance portion of the algorithm, the leader acts on the set L, attempting to invite the agents not in its group into the group. Agents in M but not L will be re-invited. From the messages they receive from the leader the agents in $M - L$ will be able to infer that they are not a part of L. To accomplish this, the membership state of the process can be provided to the recipient as part of the AYC message or it can be inferred from the process based on the response from an AYT message.

Theorem 13. *In a synchronous system a process in M but not L can determine its exclusion and act as though it was not in M, for the purpose of discovering leaders.*

Algorithm 3 Group Maintenance and Leader Discovery Functions

```
1: function CHECK
2:     This function is called at the start of a round by a leader
3:     if State ≠ Normal or Coordinator ≠ Me then return
4:     end if
5:     Expected ← ∅
6:     for j ∈ (AllPeers − {Me}) do
7:         AreYouCoordinator(j)
8:         Expected ← Expected ∪ j
9:     end for
10:    Peers which respond "Yes" to AreYouCoordinator are put into the Coordinators set.
11:    Processes that respond are removed from Expected.
12:    When an AreYouThere response is "No" and this process is a coordinator, the querying process is put in the Coordinators set.
13:    Wait for responses, Peers that do not respond are removed from UpPeers.
14:    UpPeers ← (UpPeers − Expected) ∪ Me
15:    UpPeers ← (UpPeers − Coordinators) ∪ Me
16:    if Responses = ∅ then return
17:    end if
18:    if Me < min(Responses) then
19:        MERGE(Responses)
20:    end if
21: end function
22: function RECEIVEAREYOUTHERE(Sender, Identifier)
23:    if GroupID = Identifier and Coordinator = Me and Sender ∈ UpPeers then
24:        Respond Yes
25:    else
26:        Respond No
27:        Coordinators ← Sender
28:    end if
29: end function
```

An agent's membership in M but not L is secure to the agent in M to the coordinator and from the coordinator to the agent. Let φ_i indicate j's membership in i's group. The proof is identical to Theorem 4.

However, membership in M is leaked by the next "Are You There Message" the agent sends.

1.	$I_{i,j}\varphi_{AYT}$	j sends an AYT message to i.
2.	$B_i I_{i,j}\varphi_{AYT} \wedge T_{i,j}\varphi_{AYT}$	i Trusts the message from j.
3.	$(j \notin L \wedge j \in N \wedge \varphi_{AYT}) \rightarrow j \in M$	Valuation membership.

i knows j is in N_i, but recieving the AYT message leaks that j considers itself part of L_i. Since j is not a part of L_i, but considers itself to be, j must be in M. Any message that implies j belief in it's membership of L_i is sufficent to include j in M.

‘

Algorithms can potentially update L_i based on information flow to include those agents in M_i. However, in this instance, to ensure memorylessness, the process in M_i must act in accordance with its exclusion from L_i.

Theorem 14. *A process in M_i but not L_i has the same likelihood of being in the next group as a process in not in M_i and not in L_i.*

A process in neither M_i nor L_i must not have the AYC message it sends and the response nor and have its own AYC and that response omitted to reach the next step of the algorithm. Successful completion depends on the delivery of 4 messages total. If a process is in M_i but not L_i, the same obligation follows, but that process sends and AYT and expects that response, in addition to the AYC from the coordinator. Successful completion still depends on the delivery of 4 messages.

This approach allows the memorylessness property for the imaging to be upheld. If an agent is not in L that agent will be able to determine it is not in L. This allows the same number of messages to be exchanged regardless of an agent's membership in L. If the probability that any given message sent to a process has a

consistent probability of being delivered, the sequences of worlds that merge two coordinators has the same distribution of probable outcomes as the sequences of worlds that a process in M but not L is invited to the group.

6.4. COORDINATOR PRIORITY

To alleviate the inherit race condition of leader selection, without loss of generality, a fixed leader process was selected. This process was the only one that could become the leader of a multiprocess group. This is shown in Figure 6.1, where only the selected process can send invite messages. This simplification was applied because the configuration of the system with a larger number of processes depended on the configuration of the other processes. Without this simplification, the state of the rest of the system would not have the memoryless property. The state of the processes that are not in the observers group would change each round. As a consequence the state of the rest of the system and the likelihood of forming a specific group size would change each step if other processes could become leader. A probabilistic prediction would depend on how long the processes have been separated. After a long enough period of time the distribution of states for the processes could be found using a steady-state analysis. However, when processes enter the "Recovery" state they are deterministically placed in the alone state. As a consequence the state of the system depends on the number of steps since the last reconfiguration, which violates the memoryless property.

Suppose P1 fails to detect P2 and P3 multiple times. The probability P2 has grouped with P3 could then be modeled as a two process sub-case of this three process case. Let this two process sub-case be described with a profile chain P_{sub}. The probability of P2 and P3 joining P1's group depends on the state of the sub-case. Additionally the probability distribution of an observation of the sub-case depends

how many steps the sub-case has completed. If P_{sub} is ergodic, then $Steady(P_{sub}) \neq$ [1.0 0]. Each step of the sub-case (n) will move its distribution asymptotically closer to the steady-state:

$$[1.0 \quad 0] * \lim_{n \to \infty} P_{sub}^{n} = Steady(P_{sub}) \tag{6.1}$$

Since the initial state of the algorithm in the sub-case must be un-grouped ([1.0 0]), and the process is ergodic, the probability distribution of the sub-case depends on the number of steps (n). As a consequence, the distribution for any n must be different from the distribution of $n-1$. Since the probability of P1 completing an election of P2 and P3 depends on the grouping of P2 and P3, the behavior of the full case cannot be memoryless, since it is a function of n. Therefore, to make the algorithm memoryless, only a selected process may lead a multiprocess group.

Because elections can be triggered simultaneously, (and they always are in a synchronized execution model), a race condition exists in selecting the coordinator. In the original algorithm, the race condition was resolved using the agent identifier and a wait sequence. The agent that believed it had the lowest identification would immediately send invites, all other agents would wait some period of time before before sending theirs, in case the sender crashed. Our approach uses the maintenance and discovery phase to determine if an agent is the highest priority.

6.4.1. Original Algorithm. In the original algorithm, the break the race condition, each coordinator that has identified other coordinators to merge will determine if it should wait or immediately invite each agent it has identified as another coordinator. This action allows the agents to avoid sending all their invites at the same time.

The original text is vague on how this condition should be determined: if the coordinators should determine their wait time based on the global list of agents that

could potential become leaders or if should be determined from the set of coordinators that that agent has identified as a fellow coordinator. In either case, the selection of when to send the invites influenced heavily by timing differences between the agents leading to non-deterministic behavior in many scenarios.

6.4.2. Modified Algorithm.

Theorem 15. *With perfect information transfer, agents can determine the exact coordinator priority for each participating coordinator.*

If every agent has a unique identifier, then an algorithm that sends that identifier to each other process and receives and identifier from each other process, then that agent will know every identity in the network and will be able to determine it's priority relative to other participants.

Theorem 16. *Deterministic selection allows the likelihood of a particular invite being accepted to be computable for each agent.*

Querying an agent to see if they are a coordinator implies that the sending agent is a coordinator and implies that if that agent is not in a group with that agent and the querying agent is a coordinator, the receiving agent should expect to receive an invite from that agent, and will accept it if it is not already in a higher priority group, and it does not receive a message from a higher priority agent.

This approach of identifying priority during the group discovery phases allows the agents to determine when to send invites based on a deterministic measure rather than a probabilistic one based on execution ordering. A process can decide which invite it wishes to accept before the invites are sent out. There is a potential for live-lock in this approach if the highest priority process always crashes after other agents have determined they should accept that agents invite, but a similar scenario also exists in the original algorithm.

6.5. INVITATIONS

6.5.1. Original Algorithm. In the original algorithm, a process accepts the first invitation it receives and does not accept any subsequent invitations.

6.5.2. Modified Algorithm.

Memorylessess.

State Determination. In the invitation portion of the algorithm, the sender of the invite is confident the agent has accepted the invite because the accept message arrives. The group list that the new leader prepares for the ready message is an N set of the potential group members. The list is distributed to the new members. Agents that do not receive N or fail to send a receipt message are not in L. Algorithms that depend on the the set of agents in the group will act on N. There is a potential that to the leader the agents in N but not L will act as though they are part of the group. Algorithmically, the the leader can then expand L to cover the information leaked by that agent acting as though it is holds a state that makes it part of L.

An agent can determine the likelihood that its group will form in a particular configuration if it is the highest priority agent. The determination of coordinator priority is determined during the check phase, then an agent can determine if it is the highest priority agent. If an agent determines it is the highest priority, it can then easily determine the probability that any given agent receives it's invite message and that the response to that invite message returns to that agent.

An agent can determine the likelihood that its group will form a particular configuration if it can determine the likelihood that agents with higher priority do not form a group with with the agents in that configuration.

If the determination of the coordinator priority reveals that an agent has a lower priority than 1 other agent, the probability that an agent accepts and invite

Algorithm 4 Invitation Functions

```
1: function MERGE(Coordinators)
2:     This function invites all coordinators in Coordinators to join a group led by Me
3:     State ← Election
4:     Stop work
5:     Counter ← Counter + 1
6:     PendingID ← (Me, Counter)
7:     Coordinator ← Me
8:     Pending ← UpPeers − Me
9:     for j ∈ Coordinators do INVITE(j,Coordinator,PendingID)
10:    end for
11:    Wait for responses, Peers that accept the invite are added to Pending.
12:    State ← Reorganization
13:    GroupID ← PendingID
14:    UpPeers ← Pending
15:    for j ∈ UpPeers do READY(j,Coordinator,GroupID,UpPeers)
16:    end for
17:    Expected ← UpPeers
18:    Wait for responses, Peers that acknowledge are removed from Expected
19:    UpPeers ← UpPeers − Expected
20:    State ← Normal
21: end function
22:
23: function RECEIVEINVITATION(Sender,Leader,Identifier)
24:    if State ≠ Normal then return
25:    end if
26:    if Sender < Coordinator then return
27:    end if
28:    Stop Work
29:    PendingID ← Identifier
30:    State ← Election
31:    Accept(Coordinator, Identifier)
32:    State ← Reorganization
33:    if Ready is not received then
34:        Recovery()
35:    end if
36: end function
37:
38: function RECEIVEACCEPT(Sender,Leader,Identifier)
39:    if State ← Election and PendingID = Identifier then
40:        Pending ← Pending ∪ Sender
41:    end if
42: end function
43: function RECEIVEREADYACKNOWLEDGE(Sender)
44:    Pending ← Pending ∪ Sender
45: end function
```

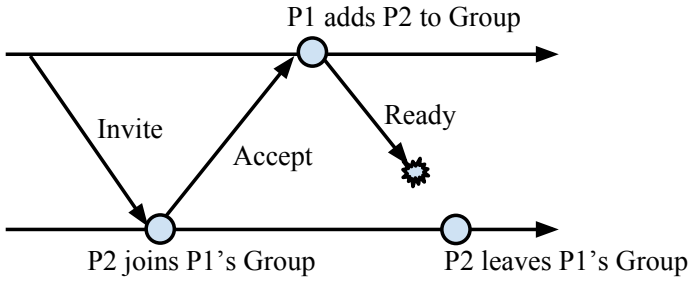

Figure 6.3: The grouping processes effects the outcome of elections

from the agent is the probability the other agent does not recognize the other agent's priority (which is based on the delivery of the check messages).

Alternatively, if the determination of the coordinator priority reveals that an agent has a lower priority than n ($n > 2$) other agents, the probability that some other agent will accept that agents invite is the probability that for each of the other agents ($n - 1...0$) the invited agent does not recognize that agent's priority.

6.6. READY STATE

The changes added a third message to completing an election – a "Ready ACK" message, shown in Figure 6.1. This message is sent by a member after receiving the ready message from the coordinator. This message induces a causality that allows the coordinator to be certain of the member's status before the next round. Without the ready acknowledgment, the member may not receive the ready message and the coordinator will observe the member is a part of the group. This interaction is shown in Figure 6.3.

Without the ready acknowledgment message, the probability a member remains in a group in the first round after an election has a different probability than each subsequent round. By adding the extra message, the observation of the coordinator must be the state of the member of the group. As a consequence, the probability

a member remains in a group in the first round after an election is identical to each subsequent round.

6.6.1. Original Algorithm. In the original algorithm the list of agents in the group is distributed as part of the ready message. Based on analysis presented earlier in the paper, since there is no confirmation of receipt for the ready message, the leader cannot determine the actual state of the group has it only has the unvaluateable set of wffs describing the group state N_i.

6.6.2. Modified Algorithm.

Theorem 17. *Ready and Ready Acknowledge result in the leader holding and L state and the members holding a N state.*

Proof: This result is based partially on the non-deducibility results presented previously in the Two-Armies problem. Let $I_{i,j}\psi_j$ be an agent j sending an invite accept to agent i.

1.	$I_{i,j}\psi_k \forall j$	Each agent j that accepts the invite does so.
2.	$I_{j,i}\{\varphi_k \forall k\}$	i sends φ_i to j (Ready Message).
3.a.	$w \nvDash V^i_{B_j\varphi_k}(w)$	i has no means of determining if j received the ready message.
3.b.	$B_j I_{j,i}\varphi_k \wedge T_{j,i}\varphi_k \forall \varphi_k$	j Trusts the contents of the Ready message.
4.b	$B_j B_i \varphi_k \forall \varphi_k$	By C1.
5.	$w \vDash V^j_{B_i\varphi_k}(w)$	j believes that i believes k is part of the group.
6.	$I_{i,j}\gamma_j \wedge B_i I_{i,j}\gamma_j \wedge T_{i,j}\gamma_j$	i receives the Ready Acknowledge.
7.	$B_i\gamma_j \rightarrow w \vDash V^i_{B_j\varphi_k}(w) \wedge B_i\varphi_j$	i knows that j believes the N_i set and knows that j is a part of the L_i set.

The proof repeats for each j that successfully delivers an invite accept message to i. Therefore, j has the N_i set and i has the L_i set.

In the modified algorithm, the agents that receive the ready message reply with confirmation. This allows the leader of the group to determine L_i, a set of agents that have definitely received the ready message. By coupling this with the

other modifications to the algorithm, for the semi-synchronous execution environment used by the DGI, the states observed by the leader are memoryless.

6.7. COMPOSED ALGORITHM

In the specified execution environment, each time before the coordinator runs the the check portion of the election algorithm it produces an image of the current world to use as the first state of the constructed Markov chain. Each agent in the system has the stability property. As part of the stability property any wffs whose information was transferred after the last image was taken cannot change until the process takes the newest image. For the election algorithm, this causes the leader to doubt the values in L.

If a process is not in L it will have to go through the election procedure even though it may be in M. This ensures that the sequence of changes necessary to bring that agent into the group is identical whether regardless of the agent's membership in M, ensuring the memorylessness property. Previous work [29] describes this algorithm and shows a Markov chain accurately describing the behavior of an implementation of this algorithm.

Creation of A Profile Chain. The election algorithm produces and distributes a set of processes $UpPeers$ and a coordinator of that set. $UpPeers$ is distributed via message passing and maintained by the coordinator. Different processes may have different versions of $UpPeers$ for a given coordinator as processes enter and leave the group. However, a process will eventually receive an up-to-date version of $UpPeers$ from the coordinator, or it will leave the group.

To model the behavior of the algorithm, at each time-step the selected process recorded the cardinality of its membership set $UpPeers$. The selected process was always the coordinator of any formed group, and was the only process that could lead

a multi-process group. The cardinality of $UpPeers$ directly mapped to the state in the Markov Chain. For example given the membership set $S = \{P1, P2, P3\}$, then $|S| = 3$ and the state of the system in the Markov chain is also $i = 3$.

The profile Markov chain for the algorithm was constructed as a closed form representation of the algorithm's behavior. In each round, the behavior is described by two components: maintaining a group and inviting other processes into the group. The coordinator will exchange an "Are You Coordinator" message and the peer will respond to verify is still available. To maintain a group of m other processes, the probability is defined as a random variable with the following probability distribution (pdf):

$$\Pr_M(X = k; m) = \begin{cases} \binom{m}{k} p^{2k}(1-p^2)^{m-k}, & \text{if } 0 \leq k \leq m \\ 0, & \text{otherwise} \end{cases} \tag{6.2}$$

Where k is the number of processes remaining in a group selected from m processes. A process will leave a group if, from the considered process's perspective, they do not respond to an "Are You Coordinator" message. Members cannot change their state without coordinating with the leader.

To invite other processes to the group, the two processes ultimately exchange up to 8 messages. In a round, a single process can invite many other processes to its group. From a selection of n other coordinators, the probability distribution for joining a new group with k of the n processes is:

$$\Pr_I(Y = k; n) = \begin{cases} \binom{n}{k} p^{8k}(1-p^8)^{n-k}, & \text{if } 0 \leq k \leq n \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

In the profile chain, in a state s that describes the number of processes in a group, the probability of transitioning from s to s' with n total processes (including the considered process) is:

$$\Pr_T(Z = s'; n; s) = \sum_{i=0}^{s-1} \Pr_M(X = i; s-1) \cdot \Pr_I(X = s' - i; n - s - 1) \tag{6.4}$$

From this distribution, a set of transition probabilities can be calculated for a given omission rate p and number of processes n. This set of transition probabilities forms a profile Markov chain P, which can be evaluated to for any number of processes n and omission rate p. The generated profile chain is ergodic when $0 < p < 1.0$. The profile chain is a stationary Markov chain.

6.8. MODEL VALIDATION

To assert the closed form profile chain accurately represents the implementation of the algorithm, it must be validated. Since T and P are ergodic, they can be checked for equivalence using a goodness-of-fit test. If the goodness-of-fit test indicates the chains are equivalent, they will generate similar sequences and have similar properties when analyzed. Therefore, generated P chains can be used to analyze the behavior of the algorithm during live execution with changing conditions.

To verify the test chain T is equivalent to the profile chain P, a χ^2 goodness-of-fit test is employed. The null-hypothesis of this test (H_0) asserts the profile chain P is equivalent to the test chain T:

$$H_0 : T = P \tag{6.5}$$

with an alternative hypothesis that the two chains are not equivalent:

$$H_1 : T \neq P \tag{6.6}$$

The χ^2 test measures the goodness-of-fit for a complete chain by combining the measurements of goodness of fit for the transitions away from each state. Therefore, the goodness of fit test for the chain is a summation of tests for each state:[7]

$$\chi^2 = \sum_{i}^{m} \sum_{j}^{m} = \frac{n_i (P_{ij} - T_{ij})^2}{P_{ij}} \tag{6.7}$$

Where n_i is the number of times the state i was observed in the input sequence used to construct the test chain T. The summation is distributed as χ^2 with $m(m-1)$ degrees of freedom (DF) if all entries in P_{ij} are non-zero. In this work, all transitions in the profile Markov chain P are non-zero when $0 < p < 1.0$. However, the probability of some transitions may be extremely small. The χ^2 value was compared to a critical value (CV) giving a measure of how likely it was H_0 could not be rejected. This work selected an $\alpha = 0.05$ significance level to reject the hypothesis $T = P$.

If the hypothesis H_0 were to be rejected, it would indicate the test chain and profile chain differ significantly. As a consequence of rejecting the hypothesis, the implementation would have behavior from the generated closed form solution.

To verify the model, it was compared to runs of an implementation of the algorithm. Test data was collected for systems with 3, 4, 5 and 6 processes. Information was collected from sufficiently long runs of the system with an omission rates between 0.05 and 0.95 tested at intervals of 0.05. Table 6.1 shows the measured error and p-value for the worst observed error for each number of processes. Since the measured error is less than the critical value and the p-value is greater than 0.05, we cannot reject H_0. As a consequence, the profile chains (P) are representative of the behavior of the algorithm's implementation.

Processes	DF	CV	Worst Error	Pr($WorstError$)	p-value
3	6	12.6	8.90	0.80	0.18
4	12	21.0	14.55	0.75	0.27
5	20	31.4	23.47	0.65	0.27
6	30	43.8	32.69	0.85	0.34

Table 6.1: Summary of χ^2 tests performed.

6.9. PROFILE CHAIN ANALYSIS

The DGI focuses on managing power resources in a microgrid environment. Resources can only be managed effectively when multiple DGIs coordinate together to manage those resources. Without another DGI to coordinate with, the DGI has a limited range of options to manage power generation, storage and loads. Therefore, the amount of time DGI will spend coordinating with another process is of particular interest. [28] defines an "In Group Time" (IGT) metric to measure the amount of time a DGI process spends coordinating with at least one other process. In this work, we define IGT based on the steady state of the profile chain. Let $\pi = Steady(P)$ for some profile chain. The IGT is the sum of all states in π, save the first state where the process is alone:

$$IGT = \sum_{i=2}^{m} \pi_i \tag{6.8}$$

The IGT is a number between 0 and 1. It represents the probability a random observation sees a group of at least two members. The steady state distributions are presented as stacked bar graphs in Figures 6.4, 6.5, 6.6 and 6.7. Each complete bar in the graph indicates the IGT. Additionally, Figures 6.4, 6.5, 6.6 and 6.7 also contain the average group size (AGS), when the system has reached the steady-state, plotted as a fraction of the total number of processes. Let P be the steady-state distribution vector for some number of processes, n, and a given omission rate:

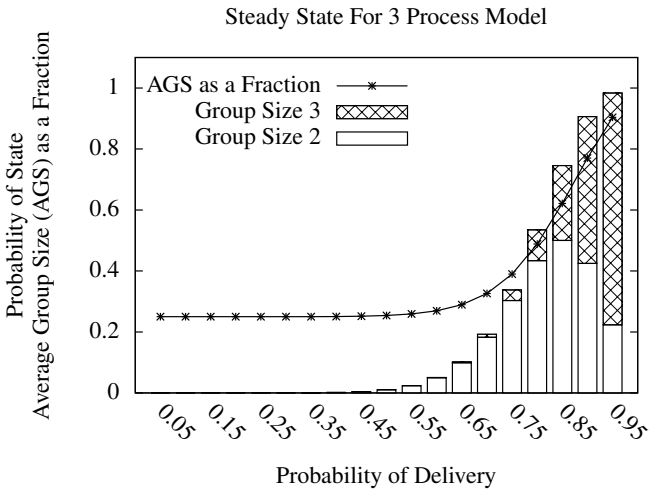

Figure 6.4: Steady state distribution for 3 processes as well as the Average Group Size (AGS) as a fraction of total processes.

$$y = \frac{\sum_{i=1}^{n} P_i * i}{n} \tag{6.9}$$

where y is the plotted AGS as a fraction.

The components of each bar represent the probability the system is in a specific state for a random observation of the system. The height of the component represents the relative probability of observing that state when in a group.

The profile chain can be used to ensure the FREEDM smart grid is able to continue operating despite network issues. The profile chain can be combined with different message sending strategies to maintain service. For example, the DGI can change to a slower mode of operation to ensure operation continues normally despite connectivity issues. By selecting different strategies depending on the message delivery probability the DGI can offer high performance in good network conditions and an acceptable level of service during faults. The profile chain can be extended to an arbitrary number of processes as shown in Figure 6.9. In Figure 6.9, the steady-state of the system is used to compute a weighted average of the group size. To

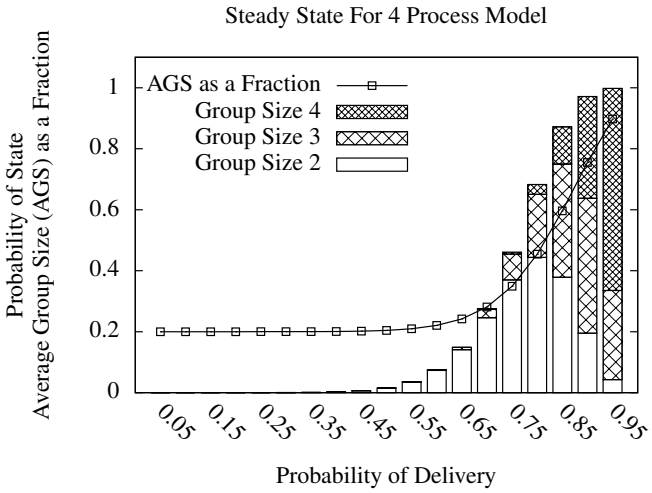

Figure 6.5: Steady state distribution for 4 processes as well as the Average Group Size (AGS) as a fraction of total processes.

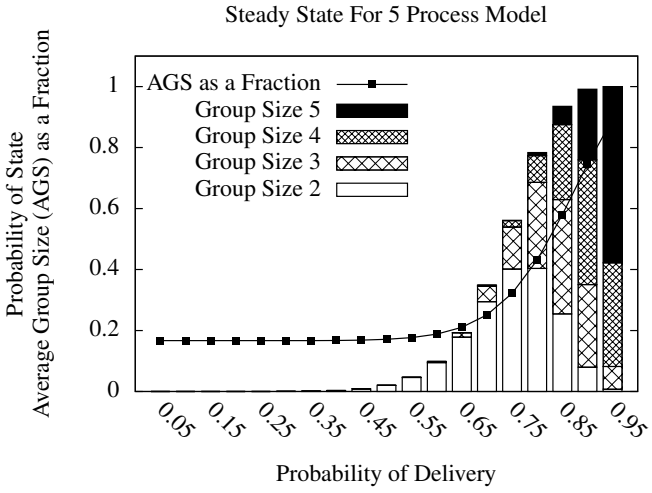

Figure 6.6: Steady state distribution for 5 processes as well as the Average Group Size (AGS) as a fraction of total processes.

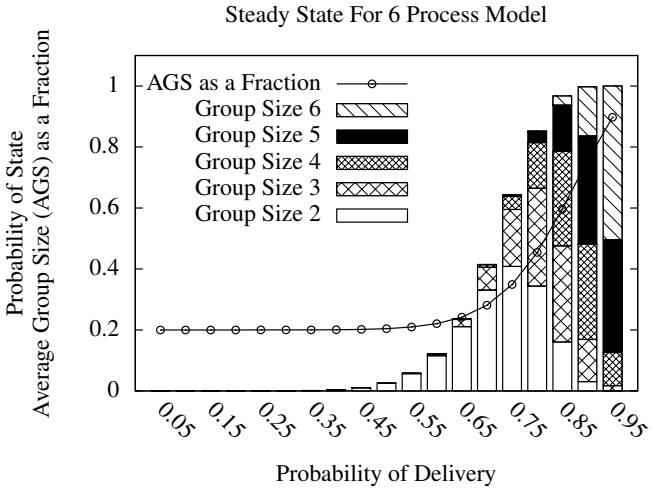

Figure 6.7: Steady state distribution for 6 processes as well as the Average Group Size (AGS) as a fraction of total processes.

compare the produced steady states, the weighted average was plotted as a percentage of all processes in the system. Values in Figure 6.9, were plotted using Equation 6.9. Therefore, Figure 6.9, shows the average percentage of total processes that will be in the group in a steady-state system.

The DGI employed a round-robin schedule where each module is given a predetermined amount of time to execute. This schedule was determined by the number of DGI that could be grouped together and the expected cyber network conditions. Additionally, the Load Balance module, which managed the power resources was scheduled to run multiple times in a long block before the group was evaluated for failure. This schedule is depicted in 6.8. The system began by using group management to organize groups. Next load balancing ran to manage power resources in the created group. Finally, state collection collected a casually consistent state to be used for reporting and offline analysis. If message delays occured, the number of migrations load balancing could perform was reduced. However, by reducing the group size, the number of messages sent by load balancing could be reduced, allowing

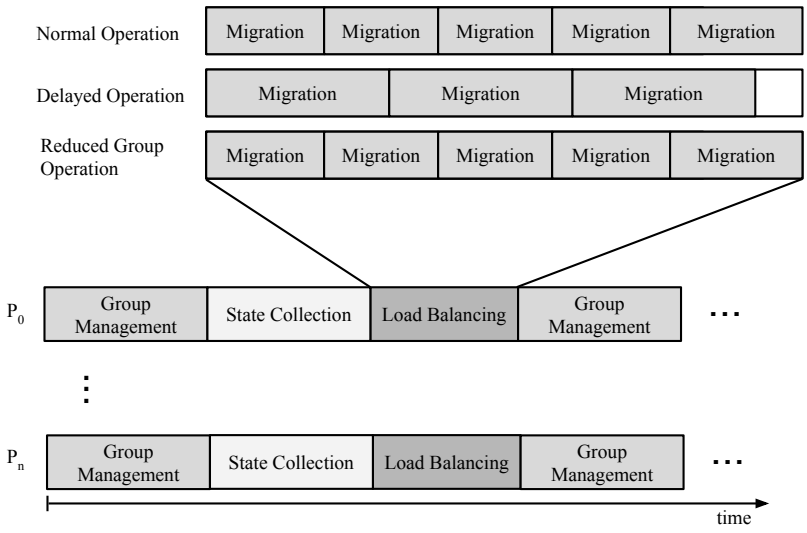

Figure 6.8: Example DGI schedule. Normal operation accounts for a fixed number of migrations each time the load balancing module runs. Message delays reduce the number of migrations that can be completed each round. However, reducing the group size allows more migrations to be completed (because fewer messages are being exchanged) at the cost of flexibility for how those migrations are completed.

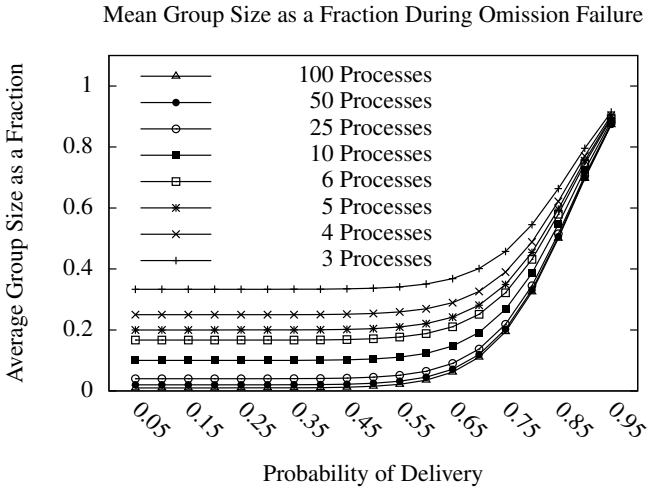

Figure 6.9: Average group size as a percentage of all processes in the system for larger systems.

for a greater amount of work to be done. The outcome of an election had a direct effect on how effectively the DGI can manage resources. Therefore the steady-state analysis gives a good indication of how well the DGI could manage those resources in the future.

This information can also be used to anticipate faults and mitigate them before they occur. Modern routers can supply an expected congestion notification as part of the IP header[19]. When congestion is anticipated, the coordinator can preemptively split the group to reduce congestion. This is possible because the algorithm's message complexity is $O(n^2)$. Future work will combine this analysis with ECN techniques to preemptively change the behavior of DGI to ensure a good level of service. If dropped messages account for an omission rate of even 0.15 from future congestion, performing a coordinated group division can potentially save several rounds of transient states. Breaking the large groups into smaller sets can drastically reduce the number of messages transmitted and help relieve congestion. Additionally, the coordinator can design the split to ensure work can continue when the groups are split, by mixing

supply and demand processes. The division can also account for the placement of congested routers for targeted congestion relief. Furthermore, since the required execution time is a function of group size, the DGI can use the additional execution time within the same real time schedule to use more reliable techniques to deliver messages.

In Figure 6.8, Group Management's execution is broken into four steps: check, merge, ready, and cleanup. Between each step, the DGI waits while messages and their replies are delivered. The leader election algorithm expects that all replies arrive before the next step of the algorithm is executed. If a message does not arrive during the wait period, before the next step of execution the message is considered lost. If it arrives later, it is ignored by the algorithm. While dynamically adjusting the synchronized schedule is not feasible during failure, adjusting the number of messages sent (by sending fewer "Are You Coordinator" messages, for example) is. By sending fewer messages, the number of packets in the communication network is reduced, and the savings in processing can allow multiple delivery attempts in the scheduled wait time.

7. APPLICATION

In this work, we consider the effects of network congestion on a cyber communication network used by the FREEDM smart-grid. We create a model version of a large number of DGI processes in a simple partitioned setup. In the FREEDM smart-grid, the consequences of this congestion could result in several problematic scenarios. First, if the congestion prevents the DGI from autonomously configuring using its group management system, processes cannot work together to manage power devices. Secondly, if messages arrive too late, or are lost, the DGI could apply settings to the attached power devices that drive the physical network to instability. These unstable settings could lead to problems in the power-grid like frequency instability, blackouts, and voltage collapse.

These techniques allow the DGI to anticipate behavior during a fault and allow it to pre-emptively harden itself against the fault.

7.1. APPLICATION: ECN HARDENING

7.1.1. Random Early Detection. The RED queueing algorithm is a popular queueing algorithm for switches and routers. It uses a probabilistic model and an EWMA to determine if the average queue size exceeds predefined values. These values are used to identify potential congestion and manage it. This is accomplished by determining the average size of the queue, and then probabilistically dropping packets to maintain the size of the queue. In RED, when the average queue size avg exceeds a minimum threshold (min_{th}), but is less than a maximum threshold (max_{th}), new packets arriving at the queue may be "marked". The probability a packet is marked is based on the following relation between p_b and p_a where p_a is the final probability a packet will be marked.

$$p_b = max_p(avg - min_{th})/(max_{th} - min_{th}) \tag{7.1}$$

$$p_a = p_b/(1 - count * p_b) \tag{7.2}$$

Where max_p is the maximum probability a packet will be marked when the queue size is between min_{th} and max_{th} and *count* is the number of packets since the last marked packet. With RED, the probability a packet is marked varies linearly with the average queue size, and as a function and the time since the last packet was marked. If avg is greater than max_{th}, the probability of marking trends toward 1 as the average queue size approaches $2 * max_{th}$ In the event the queue fills completely, the RED queue operates as a drop-tail queue.

In a simple implementation of the RED algorithm, marked packets are dropped. For a TCP application, the result of the dropped packets causes the slow-start congestion control strategy to reduce the rate packets are sent. A more advanced implementation, using ECN, sets specific bits in the TCP header to indicate congestion. By using ECN, TCP connections can reduce their transmission rate without re-transmitting packets.

UDP applications have not typically utilized ECN. Although the ECN standard has flags in the IPv4 header, access to the IPv4 header is not possible on most systems. Furthermore, there is not a "one size fits all" solution to congestion in UDP algorithms. However, for the DGI and a class of similar real-time processes, congestion notification has great potential. If processes can adjust the amount of traffic they send based on the anticipated congestion (by disabling features, for example), they can decrease the effects of congestion.

7.1.2. Usage Theory. Since the ECN fields in IPv4 are not available to applications running on the system, the notifications are multicast onto the source interface. This application is responsible for generating the multicast ECN message.

It also keeps a register of hosts running applications that support reacting to the ECN notification.

There are several reasons for this approach. First, related work has shown an ECN strategy without some other queue management scheme is not sufficient to prevent congestion. By allowing real-time applications that decrease the number of messages for congestion special priority in the RED algorithm, we allow those applications to continue operating during congestion. Additionally, in later sections, we demonstrate this strategy is effective for managing congestion.

When the RED algorithm identifies congestion it must notify senders of congestion. Since this approach is non-standard and most UDP applications would not understand the notification, we have opted to create an application that runs on switches and routers. Congestion is detected, the application sends a multicast beacon to a group of interfaces informing the attached devices of the level of congestion. For similarity with the RED algorithm and the NS-3 implementation, this notification is classified as either "soft" or "hard." A soft notification is an indication the congestion in the network is approaching a level where real-time processes can expect message delays that may affect their normal operation. A hard notification indicates the congestion has reached a level where messages are subject to both delay and loss.

7.1.3. Group Management. The group management module's execution schedule is broken into several periods of message generation and response windows. Because the schedule of the DGI triggers the execution of group management modules approximately simultaneously, the traffic generated by modules is bursty. The number of messages sent is $O(n^2)$ (where n is the number of processes in the system), in a brief window, which is dependent on how well the clocks are synchronized in the system. The duration of the response window is dependent on the amount of time it takes for messages to propagate to the hardest-to-reach process the DGI hopes to group with. Additionally, to contend with congestion, an additional slack must be

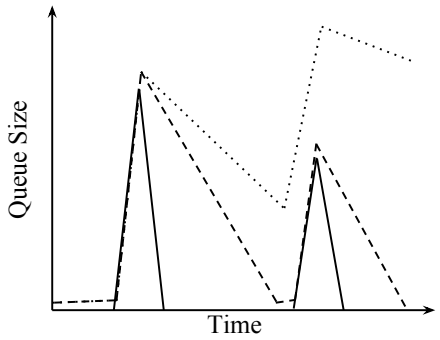

Figure 7.1: Example of network queueing during DGI operation. DGI modules are semi-synchronous, and create bursty traffic on the network. When there is no other traffic on the network (solid line), the bursty traffic causes a large number of packets to queue quickly, but the queue empties at a similar rate. With background traffic (dashed line), the bursty traffic causes a large number of packets to be queued suddenly. More packets arrive continuously, causing the queue to drain off more slowly. When the background traffic reaches a certain threshold (dotted line), the queue does not empty before the next burst occurs. When this happens, messages will not be delivered in time, and the queue will completely fill.

added to allow the RED algorithm to detect congestion before it reaches a critical level. Figure 7.1 depicts typical queueing behavior for a network device serving DGI processes under different circumstances.

Because the traffic generated by DGI modules is very bursty, the queue experiences a phenomena where the bursty traffic mixed with a steady background traffic causes the queue to fill. With no background traffic, the impulse queues a large number of messages, but those messages are distributed in a timely manner. When the background traffic is introduced, the queue takes longer to empty. At a critical threshold, the queue does not empty completely before the next burst is generated by the DGI. In this scenario, the queue completely fills and no messages can be distributed. The RED algorithm and ECN are used to delay or prevent the queue from reaching this critical threshold.

For this work, the algorithm from [29] was used. This algorithm has a higher message complexity when in a group than the Garcia-Molina algorithm it is based

on. However, it does possess a desirable memoryless property that makes it easy to analyze. This work uses an improved version of the algorithm which removes the restrictions in [29] where only one process could become the leader.

Soft ECN.. A soft ECN message indicates the network has reached a level of congestion where the router suspects processes will not be able to meet their real time requirements. The soft ECN message encourages the DGI processes to reduce the number of messages they send to reduce the amount of congestion they contribute to the network, and to allow for reliable distribution techniques to have additional time to deliver messages (since fewer messages are being sent). In the case of potential congestion, the group management module can reduce its traffic bursts by disabling elections during the congestion. When the elections are disabled, messages for group management are only sent to members of the group. Processes do not seek out better or other leaders to merge with. As a consequence, the message complexity for processes responding to the congestion notification reduces from $O(n^2)$ to $O(n)$.

Hard ECN.. In a hard ECN scenario, the router will have determined congestion has reached a threshold where the real-time processes will soon not be able to meet their deadlines. In this scenario, the real-time process will likely split its group. In an uncontrolled situation, the split will be random. It is therefore desirable when this level of traffic is reached to split the group. Splitting the group reduces the number of messages sent across the router for modules with $O(n^2)$ (where n is the number of processes in the original group) message complexity. For larger groups, splitting them provides a significant savings in the number of messages that must be queued by the router, especially since the traffic is very bursty.

Suppose a network like one depicted in Figure 7.2, where processes are divided by a router. In Figure 7.2, there are n processes on one side of the network and m on

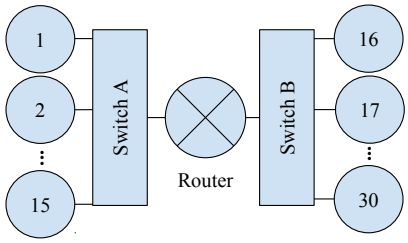

Figure 7.2: Example of process organization used in this paper. Two groups of processes are connected by a router.

the other. In normal operation the omission-modelable algorithm has an $O(n^2)$ message complexity. In Soft ECN maintenance mode, the reduced number of messages reduces the complexity to $O(n)$ by disabling elections.

During elections (and with each group update) the leader distributes a fallback configuration that will coordinate the division of the groups during intense congestion. When the ECN notification is received the processes will halt all current group management operations and enter a splitting mode where they switch to the fallback configuration. The leader of the group distributes a fallback notification to ensure all processes in the group apply their new configuration. The complexity of distributing the notification is linear $O(n)$ and processes that already received the notification will have halted their communication. This approach will ideally avoid the burst/drain phenomena from figure 7.1.

The design of the fallback configuration can be created to optimize various factors. These factors include cyber considerations, such as the likely network path the processes in the group will use to communicate. By selecting the group around the network resources, the group can be selected to minimize the amount of traffic that crosses the congested links in the future. Additionally, considerations from the physical network can be considered. Fallback groups can be created to ensure they can continue to facilitate the needs of the members. This can take into the consideration the distribution of supply and demand processes in the current group. By having a

good mix of process types in the fallback group the potential for work can remain high.

7.1.4. Cyber-Physical System. For a real-time CPS, message delays could affect coordinated actions. As result, these actions may not happen at the correct moments or at all. Since the two-army problem prevents any process from being entirely certain a coordinated action will happen in concert, problems arising from delay or omission of messages is of particular interest. In particular, we are interested in the scenario from [13], where only half of a power migration is performed. Other power management algorithms could have similar effects on the power system based on this idea of a process performing an action that is not compensated for by other processes.

Soft ECN.. In a soft congestion notification mode, the process being informed of the congestion can reduce its affect on the congestion by changing how often it generates bursty traffic. Processes running the load balancing algorithm make several traffic bursts when they exchange state information and prepare migrations. As shown before, if the interval between these bursts is not sufficient for the queue to drain before the next burst occurs, then critical, overwhelming congestion occurs. Since the schedule of the DGI is fixed at run-time processes cannot simply extend the duration of the load balancing execution phase. However, on notification from the leader, the process can, instead, reduce the number of migrations to increase the message delivery interval. This notification to reduce the schedule originates from the coordinator as part of the message exchange necessary for the process to remain in the group. Every process in the group must receive this message to participate in load balancing, ensuring all processes remain on the same real-time schedule. Using this approach, the amount of traffic generated is unchanged but the time period a process waits for the messages to be distributed is increased.

Hard ECN.. When the DGI process receives a hard congestion notification, the processes switch to a predetermined fallback configuration. This configuration creates a cyber partition. By partitioning the network, the number of messages sent by applications with $O(n^2)$ message complexity can be reduced significantly. Each migration of load balancing algorithm begins with an $O(n^2)$ message burst and so benefits from the reduced group size created by the partition.

Suppose there is a network like the in Figure 7.2 with n processes on one half and m on the other. The number of messages sent across the router for the undivided group is of the order $2mn$ as the n processes on side A send a message to the m on side B and vice-versa. Let i_1 and j_1 be the number of processes from side A and side B (respectively) in the first group created by the partition. Let i_2 and j_2 be the number of processes in the second group created by the partition under the same circumstances of i_1 and j_1. The number of messages sent that pass through the router, is then

$$2i_1j_1 + 2i_2j_2 \tag{7.3}$$

For an arbitrary group division, the following can be observed. Suppose i_1 and j_2 are the cardinality of two arbitrarily chosen sets of processes from side A and side B respectively. Following the same cut requirements as before:

$$i_2 = n - i_1 \text{ and } j_2 = m - j_1 \tag{7.4}$$

The the number of messages that must pass through the router for this cut is:

$$2i_1j_1 + 2(n - i_1)(m - j_1) \tag{7.5}$$

The benefits of the cut are minimized when i_1 and j_1 are $\frac{n}{2}$ and $\frac{m}{2}$:

$$2(2\frac{mn}{4} + mn - \frac{mn}{2} - \frac{mn}{2}) = mn \tag{7.6}$$

Which is a reduction of half as many messages. For systems with a large number of participating processes this represents a significant reduction in the number of messages sent across the router. As a consequence, this further extends the delivery window for processes sending messages.

7.1.5. Relation To Omission Model. The synchronization of clocks in the environment is assumed to be normally distributed around a true time value provided by the simulation. The shape of the curve created by plotting the queue resembles that of the Cumulative Distribution Function (CDF) of the normal distribution, noted $F(x)$. A simple description of the traffic behavior can then be described in terms of that curve. First, observe that when the queue hits a specific threshold, even if the queue is drained at an optimal rate, the nth queued packed will not be delivered in time:

$$Qsize - min(Qsize, (DequeueRate * \Delta t)) \geq 0 \tag{7.7}$$

Where Δt is the deadline for the message to be delivered. If the size of the queue exceeds the number of messages that can be delivered before Δt passes, some messages will not be delivered. The size of the queue during the message bursts created by the DGI depends on the message complexity of the algorithm, the number of messages already in the queue, the other traffic on the network, and any replies that also have to be delivered in that interval. Therefore, let c represent the rate that traffic is generated by other processes. Let $init_q$ represent the number of messages in the queue at the start of a burst. Let $init_m$ represent the number of messages sent in the beginning of the burst. Let $resp$ represent the number of messages sent

in response to the burst that must still be delivered before Δt passes. We can then express $Qsize$ as two parts:

$$Qsize = Burst + Obligations \tag{7.8}$$

Where $Burst$ takes the form of the CDF for the normal distribution:

$$Burst = init_m * F(x) \tag{7.9}$$

$$Obligations = c * \Delta t + init_q + resp \tag{7.10}$$

From this we can derive the equation:

$$F(x) \geq \frac{DequeueRate * \Delta t - c * \Delta t - init_q - resp}{init_m} \tag{7.11}$$

Where, from Equation 7.7, $DequeueRate * \Delta t$ is less than or equal to the number of messages in the queue. Solving for $F(x)$ gives a worst case estimate of the omission rate for a specific algorithmic or network circumstance. $DequeueRate$ is affected by the amount of traffic in the system. It should be obvious a greater amount of background traffic corresponds to a greater average queue size. From an relationship between the background traffic and the average queue size and the results presented in [29], Equation 7.11 can be used to select the ECN parameters.

7.1.6. Calibration. Since the distribution of clock synchronization was selected to be a normal distribution, the shape of the curve created by plotting the queue resemebles that of the CDF of the distribution, noted $F(x)$. A simple description of the traffic behavior can then be described in terms of that curve. First, observe that when the queue hits a specific threshold, even if the queue is drained at an optimal rate, the nth queued packed will not be delivered in time:

$$Qsize - (DequeueRate * \delta t) \geq 0 \tag{7.12}$$

Where δt is the deadline for the message to be delivered. If the size of the queue exceeds the number of messages that can be delivered before δt passes, some messages will not be delivered. The size of the queue during the message bursts created by the DGI depends on the message complexity of the algorithm, the number of messages already in the queue, the other traffic on the network, and any replies that also have to be delivered in that interval. Therefore, let c represent the rate that traffic is generated by other processes. Let $init_q$ represent the number of messages in the queue at the start of a burst. Let $init_m$ represent the number of messages sent in the beginning of the burst. Let $resp$ represent the number of messages sent in response to the burst that must still be delivered before δt passes. We can then express $Qsize$ as two parts:

$$Qsize = Burst + Obligations \tag{7.13}$$

Where $Burst$ takes the form of the CDF for the normal distribution:

$$Burst = init_m * F(x) \tag{7.14}$$

$$Obligations = c * \delta t + init_q + resp \tag{7.15}$$

From this we can derive the equation:

$$F(x) \geq \frac{DequeueRate * \delta t - c * \delta t - init_q - resp}{init_m} \tag{7.16}$$

Solving for $F(x)$ gives a worst case estimate of the omission rate for a specific algorithmic or network circumstance.

7.2. PROOF OF CONCEPT

7.2.1. Experimental Setup. Experiments were run in a Network Simulator 3.23[15] test environment. The simulation time replaced the wall clock time in the DGI for the purpose of triggering real-time events. As a result, the computation time on the DGIs for processing and preparing messages was neglected. However, to compensate for the lack of processing time, the synchronization of the DGIs was instead randomly distributed as a normal distribution. This was done to introduce realism to ensure events did not occur simultaneously. Additionally, the real-time schedules used by the DGI were adjusted to remove the processing time that was neglected in the simulation.

The DGIs were placed into a partitioned environment. The test included 30 nodes. Each of the nodes ran one DGI process. Two sets of 15 DGI were each connect to a switch and each switch was in turn connected to the router. This network is pictured in Figure 7.2. Node identifiers were randomly assigned to nodes in the simulation and used as the process identifier for the DGI.

The links between the router and the switches had a RED enabled queue placed on both network interfaces. The RED parameters for all queues were set identically. A summary of RED parameters are listed in Table 7.1. All links in the simulation were 100Mbps links with a 0.5ms delay. RED was used in packet count mode to determine congestion. ARP tables were populated before the simulation began. RED parameters were selected using results from [29].

The relationship between the background traffic and the average queue size was estimated through runs of the NS-3 simulation. Figure 7.3 demonstrates the observed relationship between the total background traffic and the maximum average queue size for that level of traffic. Additionally, the *DequeueRate* was collected from a run of the simulation without traffic, and was found to be 713.08 packets/second.

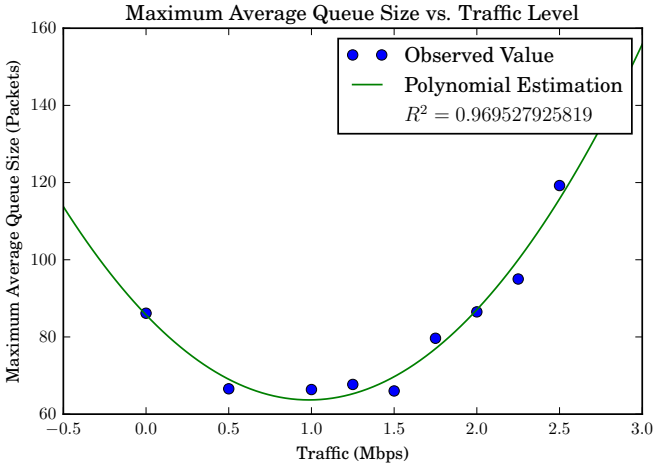

Figure 7.3: Plot of the maximum observed average queue size as a function of the overall background traffic. The polynomial estimate is $y = 22.70x^2 - 44.74x + 85.72$

Parameter	Value	Parameter	Value
RED Queueing Mode	Packet	RED Gentle Mode	True
RED Q_w	0.002	RED Wait Mode	True
RED Min Threshold	90	RED Max Threshold	130
RED Link Speed	100 Mbps	RED Link Delay	0.5 ms

Table 7.1: Summary of RED parameters. Unspecified values default to the NS-3 implementation default value

Therefore, from Equation 7.11, assuming $init_q = 0, resp = 225, init_m = 225$ and $\Delta t = 1$, the maximum traffic rate with no omissions is 263.0 packets/second. The number of packets for the $resp$ and $init_m$ were selected from the worst case of the algorithm in [29]. Based on the traffic parameters in Table 7.1, 263.0 packets/second corresponds to 1.077 Mbps of traffic generated at one switch and 2.1545 Mbps traffic overall. From the polynomial estimate in Figure 7.3, the maximum average queue size for that level of traffic is 94.715, estimated as 90 for the RED Min Threshold in Table 7.1. RED Max Threshold is computed using a similar technique, but using the message complexity for the Load Balancing algorithm, since it maintains its complexity during Soft ECN mode.

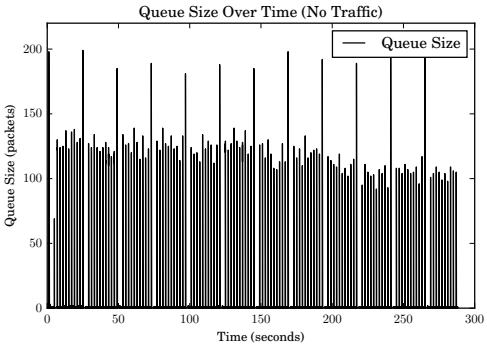

Figure 7.4: Plot of the queue size for a queue from switch A to the router when only the DGI generates traffic.

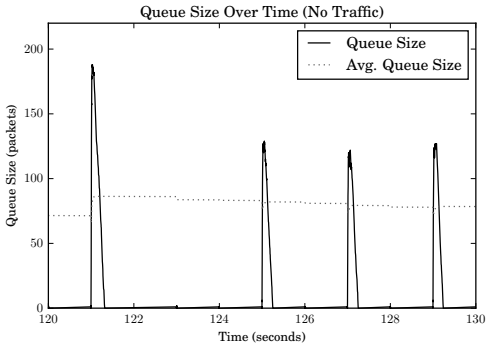

Figure 7.5: Detailed view of Figure 7.4. The left most peak is from Group Management, and the 3 smaller peaks are from power migrations.

To introduce traffic, processes attached to each of the switches attempted to send a high volume of messages to each other across the router. The number of packets sent per second was a function of the data rate and the size of the packets sent. In each simulation, half of the traffic originated from each switch. Due to the bottleneck due to the properties of the network links, the greatest queueing effect occurred at the switches.

7.2.2. Results. Figures 7.4 and 7.5 show the normal operation of the system. In this configuration, there is no congestion on the network. The DGIs start, group together and then begin migrating power between processes. Figure 7.4 plots the queue size over time for a queue used to send packets from a switch to the router. Figure 7.5 is a detailed view of a portion of Figure 7.4. Figure 7.5 shows the queue size during the normal operation of group management as well as the first migration of the load balancing module. The dotted line plots the EWMA of the size of the queue.

From this experiment we establish the min_{th} value used as a RED queue parameter. The traffic generated by each step of the group management algorithm is very bursty. It should be obvious that the tightness of the clock synchronization in

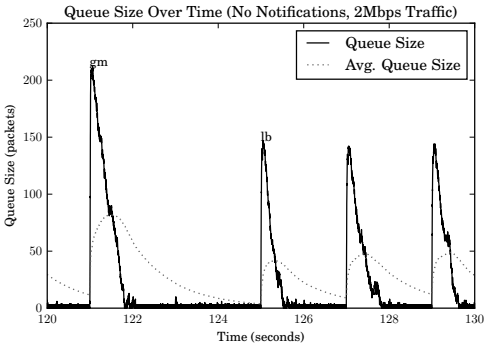

Figure 7.6: Detailed view of the effect on queue size as other network traffic is introduced. Compared to Figure 7.5, the peaks are taller and wider. Background traffic causes the average queue size to be updated more frequently.

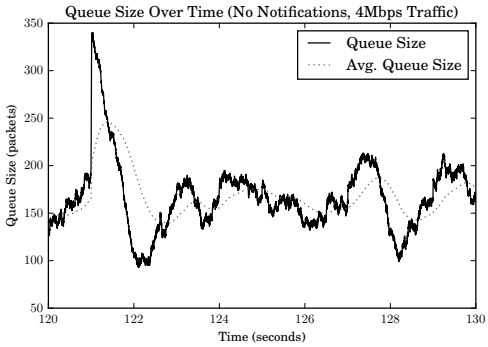

Figure 7.7: Detailed view of the effect on queue size as other network traffic is introduced. With no ECN notifications, the peak from Group Management is much larger. The congestion is sufficient that the Group Management and Load Balancing modules are affected.

the group affect how large this peak is. Like [51], the level of the power at a process is the net sum of its power generation capability and load. As power is shared on the network, processes with excess generation, converge toward zero net power. Demand processes also converge toward zero net power.

Figure 7.6 shows the queue size as the network traffic begins to increase. The DGIs in these experiments use a schedule that allows for some congestion to occur before processes are disrupted. This slack gives the network devices the opportunity to identify when the network congestion will go beyond the acceptable threshold.

Figure 7.7 shows an example of congestion affecting the physical network without ECN. As a result of the congestion in Figure 7.7, processes leave the main group. Additionally power migrations are affected: migrations are lost, or the supply process is left uncertain of migrations completions. Figure 7.11 plots the count of failed migrations over time.

Figure 7.8 shows an example of the ECN algorithm notifying processes of the congestion. Compared to the scenario in Figure 7.7, the ECN algorithm successfully

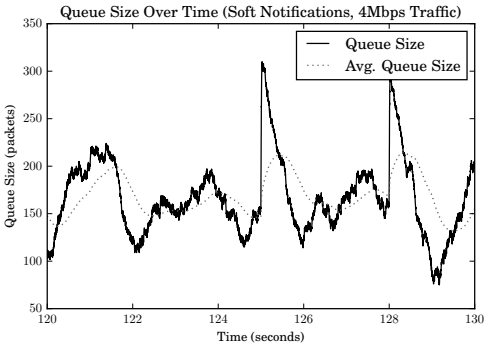

Figure 7.8: Detailed view of the effect on queue size as other network traffic is introduced. In this scenario, the ECN notifications put Group Management into a maintenance mode that reduces its message complexity and switches Load Balancing to slower migration schedule, preventing undesirable behavior.

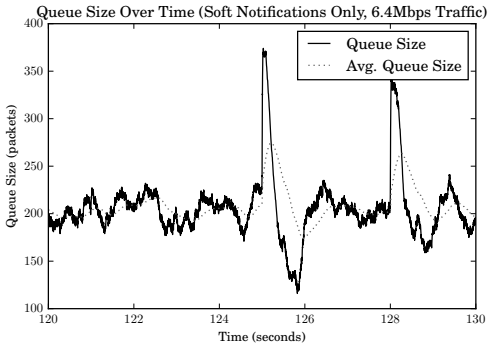

Figure 7.9: Detailed view of the effect on queue size as a large amount network traffic is introduced. Groups are unstable and processes occasionally leave the main group. Some migrations are lost due to queueing delays.

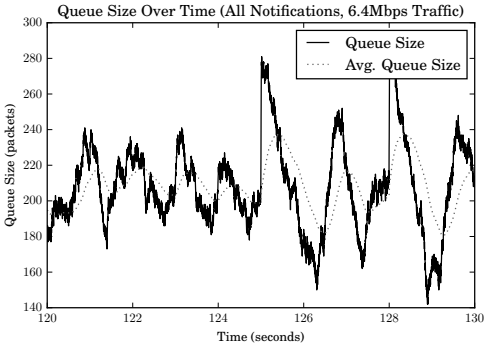

Figure 7.10: Effect on queue size as a large amount of network traffic is introduced. Hard notifications cause the groups to divide. As a result of the smaller groups, the group management and load balancing peaks are smaller than those in 7.9. No migrations are lost.

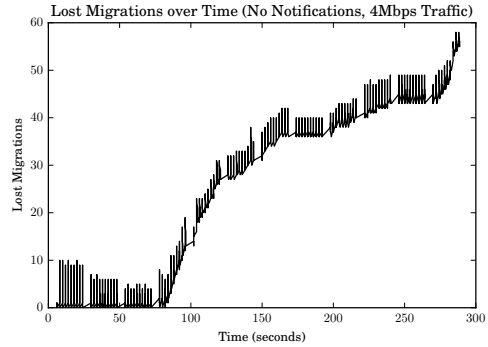

Figure 7.11: Count of lost migrations from all processes over time. Migrations are counted as lost until the second process confirms it has been completed. Without congestion management, a large number of migrations are lost.

prevents the group from dividing, and increases the number of migrations by reducing the number of attempted migrations each round.

Figures 7.10 and 7.9 show an example of a more extreme congestion scenario. In Figure 7.10, the RED algorithm shares a Hard ECN notification. This notification causes the DGI to switch to a smaller fallback configuration. This fallback configuration decreases the queue usage from Figure 7.9 to Figure 7.10. Without this fallback configuration behavior, the system is greatly affected by the traffic. However, with the fallback configuration the system remains stable and no migrations are lost.

8. 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 introducing in hardware-in-the-loop 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 [13], Choudhari et. al. show that failed transactions can create a scenario where the frequency of a power system could become unstable.

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

8.1. SELECTING A SCHEDULE

As shown previously, there is a relationship between congestion and the amount of time a process spends in a group. 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, a function which relates the

congestion in the network to predict how long a pair of processes will be able to interact will be developed. 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. Consider a set of processes that are in a group together. There is some function which describes the expected amount of time until the group would need to reconfigure:

$$E(c)[X] = \begin{cases} 0 & \text{if } c = 1.0 \\ \infty & \text{if } c = 0.0 \\ 1/\lambda(c) & \text{otherwise} \end{cases} \tag{8.1}$$

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 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 [13][41][40].

$$\{k < max\_outstanding\_migrations\} \tag{8.2}$$

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 stop performing migrations when this invariant would be violated. A related value $remaining\_k$ can be defined:

Table 8.1: 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}

$$remaining\_k = max\_outstanding\_migrations - k \tag{8.3}$$

This value can be used to identify 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

$$\{F(c) * max\_migrations \geq remaining\_k\} \tag{8.4}$$

using 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, processes 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 8.1.

The function $migrations(R_m, t)$ 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:

$$\Delta(A, B) = ((1 - F_A(c)) * migrations(R_{Am}, t)) - (1 - (F_B(c)) * migrations(R_{Bm}, t)) \tag{8.5}$$

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:

$$\{\Delta(x, y) \geq 0, \forall y \in S\} \tag{8.6}$$

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: greater understanding is needed of 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.

8.2. 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. Guards will be developed 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, guards will 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. Future work will 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:

$$\bigwedge_{x \in Groups} P_{Ix} \rightarrow P_{IG} \tag{8.7}$$

8.3. ACCURACY AND SCOPE OF THE MODEL

Future work will 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++[14] 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. Future work will consider the correctness of the incorporated algorithms, how omission failures can violate that correctness, and what restrictions can be placed on the configuration and operation of DGIs in order to protect the entire system during failures. To do this, future work 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.

8.4. DELIVERABLES

Therefore, moving forward, future work will expand the models 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, future work will 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. New models of the CPS will be created and validated against simulations and actual hardware. Invariants will be constructed that describe the correct behavior of the groups to ensure safe operation.

In this work, we presented a technique for hardening a real-time distributed cyber-physical system against network congestion. The RED queueing algorithm and an out-of-band version of explicit congestion notification (ECN) were used to signal an application of congestion. Using this technique the application changed several of its characteristics to ready itself for the increased message delays caused by the congestion.

These techniques were demonstrated on the DGI, a distributed control system for the FREEDM smart-grid project. In particular, this paper demonstrated the hardening techniques were effective in keeping the DGI processes grouped together. Additionally, it helped ensure the changes applied to the DGI through cyber-coordinated actions did not destabilize the physical power network.

This technique will be important to create a robust, reliable CPS for managing future smart-grids. However, this technique could potentially be applied to any CPS that could experience congestion on its network, as long as it has the flexibility to

change its operating mode. Potential applications can apply to both the cyber control network and the physically controlled process. For example, in a VANET system, the vehicles could react to congestion by increasing their following distance.

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

[3] 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.

[4] Chongyang Bai and Xuejun Zhang. Aircraft landing scheduling in the small aircraft transportation system. In *Computational and Information Sciences (ICCIS), 2011 International Conference on*, pages 1019–1022, Oct 2011.

[5] J. Baillieul and P. J. Antsaklis. Control and communication challenges in networked real-time systems. *Proceedings of the IEEE*, 95(1):9–28, Jan 2007.

[6] F. Baker. Requirements for IP version 4 routers, 6 1995. RFC 1812.

[7] U. Narayan Bhat. *Elements of applied stochastic processes, 2nd Edition*. John Wiley & Sons, 1984.

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

[9] P. Blackburn, M. De Rijke, and Y. Venema. *Modal logic*, volume 53. Cambridge University Press, 2002.

[10] 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.

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

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

[13] 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.

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

[15] NS-3 Consortium. Network simulator 3.23. `http://www.nsnam.org/`.

[16] Flaviu Cristian. Understanding fault-tolerant distributed systems. *COMMUNICATIONS OF THE ACM*, 34:56–78, 1993.

[17] Christopher Dabrowski and Fern Hunt. Using markov chain analysis to study dynamic behaviour in large-scale grid systems. In *Proceedings of the Seventh Australasian Symposium on Grid Computing and e-Research - Volume 99*, AusGrid '09, pages 29–40, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[18] 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.

[19] A. Dracinschi and S. Fdida. Congestion avoidance for unicast and multicast traffic. In *Universal Multiservice Networks, 2000. ECUMN 2000. 1st European Conference on*, pages 360–368, 2000.

[20] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. `http://goo.gl/dC8VT`, 2010. [Online; accessed December 2011].

[21] Timothy French. *Bisimulation Quantifiers for Modal Logics*. PhD thesis, University of Western Australia, 2006.

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

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

[24] Aniruddha Gokhale, Mark P McDonald, Steven Drager, and William McKeever. A cyber physical systems perspective on the real-time and reliable dissemination of information in intelligent transportation systems. Technical report, DTIC Document, 2010.

[25] 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.

[26] Gerry Howser and Bruce McMillin. Modeling and reasoning about the security of drive-by-wire automobile systems. *International Journal of Critical Infrastructure Protection*, pages 127 – 134, 2012.

[27] Gerry Howser and Bruce McMillin. A Multiple Security Domain Model of a Drive-by-Wire System. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 369–374. Computer Software and Applications Conference, 2013.

[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] Stephen Jackson and Bruce McMillin. Markov models of leader elections in a smart grid system (under review). *Journal of Parallel and Distributed Computing*, 2015.

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

[31] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462, May 2010.

[32] Saul A. Kripke. A Completeness Theorem in Modal Logic. *The Journal of Symbolic Logic*, 24(1):pp. 1–14, 1959.

[33] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[34] Mai Gehrke and Hideo Nagahashi and Yde Venema. A Sahlqvist theorem for distributive modal logic. *Annals of Pure and Applied Logic* , 131(13):65 – 102, 2005.

[35] 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.

[36] 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.

[37] 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.

[38] NSF FREEDM Systems Center. FREEDM, The Future Renewable Electric Energy Delivery and Management Systems Center. `http://www.freedm.ncsu.edu/`.

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

[40] 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.

[41] 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.

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

[43] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to ip, 9 2001. RFC 3168.

[44] D.B. Rawat, C. Bajracharya, and Gongjun Yan. Towards intelligent transportation cyber-physical systems: Real-time computing and communications perspectives. In *SoutheastCon 2015*, pages 1–6, April 2015.

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

[46] Thomas Roth and Bruce McMillin. Breaking Nondeducible Attacks on the Smart Grid. In *Seventh CRITIS Conference on Critical Information Infrastructures Security*. Seventh CRITIS Conference on Critical Information Infrastructures Security, 2012. (to appear).

[47] 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.

[48] K. Sampigethaya and R. Poovendran. Cyber-physical system framework for future aircraft and air traffic control. In *Aerospace Conference, 2012 IEEE*, pages 1–9, March 2012.

[49] 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.

[50] 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.

[51] M.J. Stanovich, I. Leonard, K. Sanjeev, M. Steurer, T.P. Roth, S. Jackson, and M. Bruce. Development of a smart-grid cyber-physical systems testbed. In *Innovative Smart Grid Technologies (ISGT), 2013 IEEE PES*, pages 1–6, Feb 2013.

[52] 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.

[53] 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.

[54] 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.

VITA