DECENTRALIZED COORDINATION BUILDING BLOCKS (DCBLOCKS) FOR DECENTRALIZED MONITORING AND CONTROL OF SMART POWER GRIDS

By

SHWETHA NIDDODI

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2015

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of SHWETHA NIDDODI find it satisfactory and recommend that it be accepted.

David E Bakken, Ph.D., Chair

Carl Hauser, Ph.D.

Adam Hahn, Ph.D.

ACKNOWLEDGEMENTS

My experience at Washington State University during my graduate study has been very rich, extremely valuable, and rewarding. I am especially thankful and indebted to my adviser Dr. David E Bakken for his support, guidance, encouragement, and opportunities he provided during my graduate study. I would like to also express my gratitude to Dr. Carl Hauser and Dr. Adam Hahn for serving as my committee members.

I would like to thank Dr. Anurag Srivastava for his constant guidance in understanding power system applications. I also wish to thank Ryan Goodfellow, HyoJong Lee, Ren Liu, Tanvi Ashwarya, Hyesun Cha and Aravind Mallikeswaran for all the discussions and help during the course of my thesis work. Additionally, the completion of my research has been supported in part by Réseau de Transport d′Èlectricité (RTE) France. I would also like to acknowledge Patrick Panciatici and Thibault Prevost for discussions and thoughtful inputs on power system applications.

Lastly, I'm most grateful to all the love and support of my family during the course of my study.

DECENTRALIZED COORDINATION BUILDING BLOCKS (DCBLOCKS) FOR DECENTRALIZED MONITORING AND CONTROL OF SMART POWER GRIDS

Abstract

by Shwetha Niddodi
Washington State University
December 2015

Chair: David E Bakken

Decentralized Coordination (DC) algorithms for distributed systems are widely used in many applications such as online banking, online shopping, the internet and many more. Here, the processes spread apart in a network coordinate with each other to achieve a common application specific task. DC algorithms are also applicable to wide range of large scale engineering system infrastructure. One such critical engineering infrastructure is the power grid. The common control architecture in traditional power grids for power system monitoring and control has been central control center based (CC), complemented with limited local control. However, with the ongoing and future transitions to the smart electric power grid, the CC based control is becoming unsuitable for the grid's changing demands. Smart field sensors, renewable energy sources and more Information and Communications Technology methods are being used for monitoring and control. The need for supplementing the conventional CC based power system control to a more decentralized power system control is being advocated. However, the research is mostly in developing decentralized power algorithms. This change need to be supported with robust decentralized computing and communication infrastructure.

Many DC algorithms along with different failure models supported by them have been studied and applied for more than three decades. These algorithms can used to provide a robust, fault tolerant solution for developing decentralized power algorithms. This research involves survey of existing and emerging decentralized power applications and survey of various DC algorithms available in computer literature that would be most suitable for these decentralized power applications. Design of Decentralized Coordination Building Blocks (DCBlocks) based on these DC algorithms which can act as a software platform for developing robust power applications is discussed. Few use cases of decentralized power applications using DCBlocks have been provided to show applicability of DCBlocks to a wide spectrum of power system scenarios. A prototype implementation of one of the decentralized power application using DCBlocks platform is demonstrated. Experiments are conducted using wide scale test bed environment like DeterLab for profiling the performance of DCBlocks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Distributed Computing

Distributed computing is a discipline of computer science that essentially tries to answer the question "Given that we have a computer network, how can we best use it to support applications with logic that is spread across the network? In other words, how can we coordinate, synchronize, replicate, and make it simpler to program with lower-level building blocks called middleware?". The components in the distributed systems can be heterogeneous in terms of computer hardware, operating system, programming language used and yet interact with each other using middleware. Middleware provides a portal that masks the separation of components, such that the collection of independent components appears as one integrated system. The applications running on different systems are not aware of their exact physical location and "discover" each other using some sort of discovery service. This allows for transparency of location and access where in remote and local resources are accessed using identical operations.

The advantage of using distributed computing architecture is that a complex computational problem can be split into many smaller well coordinated tasks run at individual computers. This moves the burden of performing all the computation at a single place which can easily lead to performance bottleneck, single point of failure, single point of attack etc.

Robust decentralized coordination algorithms to achieve fast coordination even in presence of failures have been developed over the past two decades.

The collaborating processes in distributed systems are spread over a network. As a consequence, there are numerous ways in which the system can fail. Messages can arrive with variable delays resulting in inconsistent ordering of messages at each processor or they can be completely omitted either due to network or processor failure. Some processors can crash or get partitioned while others are working properly and sending correct data. There may be cases where the failures observed in the communicating system can be arbitrary in nature, in which a process can arbitrarily omit to send a message, or send conflicting messages to different processes, or send duplicate messages. These types of failures can be unintentional (due to software or hardware malfunction) or intentional (malicious). Such type of failures have to be detected and handled properly without impacting the overall system performance significantly. Therefore it is necessary that distributed systems be highly fault tolerant and resilient to security attacks. To do so, appropriate failure detection and failure recovery mechanisms need to be incorporated during the early stages of system design.

Other challenges in distributed system are that they have to be scalable in terms of ever changing load (network or computational load) and resources. The system cost, throughput and latency should be unaffected with the increase or decrease in resources and complexity.

There are various applications using distributed computing like online banking, eCommerce, internet, online gaming and many more. One such application is power system applications. This research is focused on the applicability of distributed computing features to power grid and similar infrastructure. A brief introduction to power grid is provided in following section.

1.2 Power Grid Introduction

The power system network is one of the most complex and critical systems. Power system network typically evolved as a centrally coordinated (CC) control system (Wu et al., 2005), where the measurement data flows from sensors placed at different locations in the field to the control center and control action data flows from the control center to the field equipment. Such control scheme has been successfully deployed for several decades because power generation was predominantly aggregated and distribution system was passive and radial. However, due to the continuous increase in the demand for electricity by the ever growing number of customers, the need to move away from traditional electric power grid system became necessary. The need for moving the electricity generation closer to the load, monitoring the load using smart sensors and reducing the use of fossil fuels for electricity generation necessitated traditional electrical grid systems to start using advanced electronic communication technologies, smart sensors, renewable energy sources like wind farms. in the power grid. Smart grid technology is essential to address these issues.

Smart grid technologies use advanced sensing systems like PMUs, two way high speed communication, monitoring and enterprise analysis software to get location specific real time data for both the system operators (EMS, advanced metering scheme) and end users (smart meters for demand side management). Introduction of SCADA led to wide area monitoring and control of power grid. Time-synchronized measurements of magnitude and phase angle of voltage and current using PMUs and other time-synchronized measurements from protective relays, meters and fault recorders are gathered by Phasor Data Concentrators (PDCs) within each substation. The gathered data is sent to Energy Management System (EMS) installed at the CC based control centers for monitoring and control. This allows for

distributed aggregation of time synchronized measurements with precise time stamps and transmission of data to EMS for accurate system wide monitoring, state estimation and wide area control.

The existing power system applications like state estimation, voltage stability assessment, oscillation monitoring running on central control centers are designed to function reliably with full system topology and complete network observability. As a result, large amount of sensor data measurements are constantly sent to the control center. But, the large flow of sensor data from the field equipment to the control center is leading to high traffic congestion near the control center. Scability is another issue as more sensors and actuators are getting added into the field. Increased penetration of Distributed Energy Resources (DER) for distributed power generation is also adding to the data traffic to the control center. Control centers are getting over burdened with significant increase in the data flow and large set of system variables leading to increased computational time. As a result, the monitoring and control action is getting slow.

The field equipment such as relays, breakers, PMUs are typically geographically spread apart. So, when EMS in the control center, observes an unstable condition at a particular bus or substation, any control action initiated by it can have high latency due to high network load and long distance. By the time it reaches the concerned substation, the operating condition may have become more severe, calling for drastic measures to restore it back to normal operating condition. Hence, fast control response with strict timing requirements is required. Alternatively, local control for fast response is not efficient either because it is based on local disturbances and with limited network visibility without considering the consequences of its action on the nearby regions/areas. It may hinder the overall system performance even though it effectively solves the local problem. Both completely central-

ized architecture or completely local monitoring and control have limitations. An optimal solution is to have a decentralized group based monitoring and control. The power system is divided into multiple groups of substations, and monitoring and control is based on system information and data measurements from PMUs connected to substations within the group. This facilitates fast, well coordinated control action based on group level topological visibility, subsequently leading to better performance from a system wide point of view.

Increasing penetration of DERs, battery chargeable loads like Electric Vehicles (EVs) are posing new challenges because of their intermittent characteristics and uncertainty of availability. It is becoming increasingly difficult for CC based power management applications to handle the sudden fluctuations in power generation and load. Therefore demand side response to help in mitigating some of the problems such as frequency deviations is being considered. Typically, DERs such as wind farms are geographically distributed in remote areas because of which there is high latency in control action reaching them. Instead, if fast control action can be coordinated with relevant system information locally, it can reduce some of the burden on CC based EMS.

Also, the central control centers can become vulnerable to malicious attacks as it is single point of failure. This vulnerability can be reduced if critical decision making logic is spread across multiple substations.

1.3 Problem Statement

Existing power system infrastructure operate with a central or local monitoring and control architecture. The current trend in power systems is to move away from traditional central or local architecture to a more decentralized architecture. In decentralized architecture, part of the computations are implemented in each communicating component. The

computing entities coordinate periodically to achieve a common application goal. Building decentralized applications is non trivial due to several factors. Some of them are:

- Variable network delay.
- Variable computational delay.
- Different messages arriving in different order at each destination.
- Different failure types seen at each coordinating process.
- Perturbations due to security attacks such as DDoS attacks, hijack computers etc.

Therefore implementing distributed control algorithms are harder than a traditional single process one.

Although there is a lot of knowledge about algorithms for optimal electrical performance of power systems, it is highly **non trivial** to develop and implement these algorithms in distributed network environment. There is a lack of deep understanding and experience in power community in developing distributed computing applications for such decentralized architecture. An ad-hoc implementation will lead to low reliability and failures. Therefore, it is critical to provide a robust fault tolerant distributed computing infrastructure that addresses the various realistic scenarios (and unexpectedly difficult boundary cases) in a distributed network that the power community did not have to contend with in a central or local control architecture.

Fortunately, the problem of distributed coordination has been studied extensively by computer scientists since the late 1970s. However, the research literature and even applied development frameworks such as ISIS^2 (Isis2, 2015) and JBOSS (JBoss, 2015) are at a relatively low level and hence difficult and time consuming (and possibly impossible) for the

power community to adapt. The goal of this thesis is to leverage this vast body of theoretical and pragmatic research and build a collection of decentralized coordination algorithms that is most suitable for decentralized power application scenarios and to make them more robust and resilient to failures. These algorithms need to be implemented as separate building blocks with simple interfaces to make them more user friendly for power engineers.

1.4 Contributions of Thesis

The contributions of the thesis are:

i. A survey of decentralized coordination algorithms applicable to tightly-managed (synchronous) distributed systems.

ii. A survey of present and emerging decentralized power algorithms with the focus on coordination and communication.

iii. Design of Decentralized Coordination Building Blocks (DCBlocks) for necessary power applications that hide much of the complexity from non-dc specialists.

iv. Develop decentralized power applications use cases using DCBlocks in collaboration with power engineering students. This is done in collaboration with researchers of Smart Grid Demonstration and Research Investigation Lab (SGDRIL), WSU.

v. Prototype implementation of Decentralized Voltage Stability application using DCBlocks to demonstrate the features of DCBlocks. This is done in collaboration with researchers of SGDRIL lab, WSU (Lee et al., 2015), *under review*.

vi. Experimental analysis in a wide-scale test bed environment.

1.5 Organization of Thesis

The thesis is organized in seven chapters. Chapter 1 provides an introduction of distributed computing and power systems. The contributions of this thesis is also discussed in this chapter. Chapter 2 describes various sub problems in distributed computing and corresponding decentralized coordination algorithms for each sub problem. The performance and various failure models supported by them is also described. Chapter 3 describes various present and emerging decentralized power applications. This chapter establishes the trend of moving from traditional centralized monitoring and control in power systems to more decentralized monitoring and control. Chapter 4 describes the design of Decentralized Coordination Building Blocks (DCBlocks) for decentralized power applications. The failure handling mechanism of each building block is also described. Chapter 5 delineates five use cases of decentralized power applications using DCBlocks. Chapter 6 describes a prototype implementation of Decentralized Voltage Stability application using DCBlocks. The simulation results of DCBlocks building blocks profiling its performance in various case scenarios such as increasing group size, network loss, message latency and different failure conditions (crash, omission) are provided in the same chapter. Chapter 7 states the summary of the research work, conclusions, and future work.

CHAPTER 2

OVERVIEW OF DECENTRALIZED COORDINATION ALGORITHMS

2.1 Introduction

The objective of Decentralized Coordination (DC) algorithms is - Given a set of processes spread across a computer network, how best to coordinate their actions to achieve a common goal even in the presence of variable message delays, heterogeneous components, variety of system failures, security attacks etc. Numerous DC algorithms have been proposed for synchronous systems such as power systems, where there is a definite bound on the computational time and message transmission delay. Information about definite bounds on network delay and processor computational time, allows us to use timeouts to detect some of the failures like crash failures of remote processes, message omission failures and message timeout failures.

It is important to understand the pros and cons of DC algorithms and its applicability to power systems. This chapter presents an overview of the DC algorithms along with the salient features of each algorithm. The chapter discusses some of the important sub problems in DC. For each sub problem, a few applied and theoretical DC algorithms are discussed. A table consisting of summary of DC algorithms for each sub problem with various failure models supported by them and their performance characteristics are also provided. Before describing each sub problem in DC, a brief description of various possible

failure models and criteria for measuring the performance is given.

2.2 Failure Models

Failure models in DC can broadly be classified based on the domain type (time or value), nature (non-arbitrary or arbitrary) and intent (non-malicious or malicious) of failures. Timing failures are failures where messages arrive either too early or too late with respect to specific time interval (or timeout). Non malicious failures are unintentional or benign failures occurring due to software or hardware malfunction. Malicious failures are intentional failures with the intention of disrupting the flow of the application at the worst opportune time. Classification of failures based on the nature of failures is given below. All these type of failures need to be detected and handled properly.

- Non Arbitrary Failures - The failures of this type remain consistent and appear until it is properly handled by the software. They are usually non malicious or benign failures.

    i. Crash (C) - This is special type of timing failure where the process halts and yet "cleanly" without externally visible mistakes. This can be detected using heartbeat and timeout mechanism.

    ii. Omission (O) - Omission failures are also timing failures, where messages do not arrive at the receiving side. This happens either due to the sending process omitting to send the message or due to failure in communication network midway between the sending and receiving process.

    iii. Value (V) - Semantic failures are value type of failures. A semantic failure is an interaction with an incorrect meaning such as a temperature sensor below

absolute zero or otherwise incorrect in the semantics of the application.

- Arbitrary Failures - The failures of this type are arbitrary in nature, i.e., any combination of errors in the value and time domain can occur at any time. For example, a process can arbitrarily omit to send a message, or send conflicting messages to different processes, or send duplicate messages etc. These failures are difficult to detect and handle and hence is most severe of them all. They can be malicious or non-malicious.

  Byzantine (B) - Byzantine failures are malicious failures that involve sending inconsistent (and sometimes bad) data, lying about ones identity, displaying two-faced behavior by forwarding conflicting messages to different processes or lying about what messages were received.

2.3 Computational Complexity

The performance of DC algorithms can be measured in terms of following two parameters.

- Message Complexity - Total number of messages exchanged by all processes by the end of the algorithm.
- Time Complexity - Total time in terms of number of rounds, needed to complete a successful execution of the algorithm.

2.4 Decentralized Coordination Algorithms

The following sections provide a brief description of each sub problem in DC and corresponding DC algorithms (both applied and theoretical) for each sub problem.

2.4.1 Consensus

Here, the communicating processes agree on one or more values from a set of values proposed by each communicating process. Consensus algorithms should adhere to following properties.

- Termination - Eventually, all processes make a decision.
- Agreement - The decision value/s of all correct processes is the same.
- Integrity - If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

Some variants of consensus algorithms have a relaxed rule for Agreement property.

i. Simple Consensus (SC) (G. Coulouris and Blair., 2011; Raynal, 2010) - Here processes agree on same decision (scalar) value from list of proposed values. The algorithm is executed in multiple rounds.

   Round 1: Each process proposes a local value.

   Round 2 to $f + 1$: Each process tries to collect all the proposed values from other processes in the group sent in the previous rounds. It then transmit only those values which it has not received in the previous rounds.

   After $f + 1$ rounds, where f is number of failed nodes, each process decides on the same scalar value. The algorithm reaches consensus/agreement even in the presence of failures and can tolerate at most of $f = (n - 1)/2$ faulty nodes where n is total number of coordinating nodes. The decision criteria can be any function like - computing the minimum, maximum, average etc. of proposed values.

ii. Interactive Consensus Algorithm (ICA) (Pease et al., 1980; Gascn and Tiwari, 2014; Raynal, 2010) - Processes agree on a vector of values (one sent by each process). The algorithm states that at the beginning, every process proposes a value and at the end of the algorithm i.e., after $f + 1$ rounds, all the processes reach a decision and agree upon same vector of values. This vector is called decision vector. The vector contains one value per process i.e. if P_i is a process that proposes value V_i then the decision vector will contain V_i as ith component of the vector.

iii. K-Set Agreement (Chaudhuri et al., 1993; Raynal, 2010) - This problem is a variant of consensus problem. A maximum of k different values can be decided by the communicating processes. Each process can decide on a single value from the k-set. The algorithm is similar to Simple Consensus, except that agreement property is relaxed i.e., the decision set is relaxed from a single value to a k set of values. K-set agreement is reached in the maximum of $(t/k + 1)$ rounds where t is total number of failures.

iv. Paxos (Lamport, 2005) - Paxos is a highly used consensus protocol in many of the commercial software for distributed database management and state machine replication type of applications. Some of the common software using Paxos are Google's Chubby distributed lock service, Amazon web services, ISIS^2 for virtual synchrony, XtreemFS etc. The algorithm described in the original paper is very vague and difficult to understand. However, there have been many papers since then simplifying the steps of the algorithm (Rajsbaum, 2001). There are basically three participants in the algorithm: Proposer (can also be the leader), Acceptor and the Learner. There are four types of messages exchanged.

- *Prepare* Message sent to all Acceptors requesting them to accept the proposal.

The proposal message contains new proposal number and proposal value.

- *Ack/Nack* − Message sent by Acceptor in response to Prepare/Accept message request from the Proposer accepting/rejecting the request
- *Accept* Message sent by Proposer to Acceptors to accept the proposal.
- *Learn* Message sent to the Learners informing the accepted proposal.

The algorithm is executed in 2 phases, each phase has 2 rounds. They are:

(a) **Phase 1** - This is Prepare phase. In this phase, Acceptors guarantee that they will not accept any other proposal with a lower proposal number.

- **Round Phase 1a**: The Proposer selects a proposal number n (highest it has seen so far) and sends a *Prepare* request message to all the Acceptors.
- **Round Phase 1b**: When an Acceptor receives a *Prepare* request message with a proposal value higher than it has seen so far, it will reply with an *ACK* type of message with a highest valued proposal value it has seen so far. After this, the Acceptor will not accept any *Prepare* request message having a proposal number $n' < n$. However, if it later receives a *Prepare* request message having a proposal number $n' > n$, it will respond with an *ACK* message accepting the higher ordered Prepare request, abandoning the earlier Prepare request with proposal number n.

(b) **Round Phase 2** - This is Accept phase. In this phase, Acceptors commit to one proposal and accept it.

- **Round Phase 2a**: When a Proposer receives a response for its *Prepare* request message from a majority of Acceptor, it sends out an *Accept* message for same proposal number n with a value v, where v is the value of the

highest-numbered proposal among the responses, or any value of its choosing if no proposals are received.

- **Round Phase 2b**: When an Acceptor receives a *Accept* request message, it responds with an *Ack* message if it has not seen a *Prepare* message with proposal number $n' > n$.

After the proposal is accepted, the decision is sent to the Learner processes. This round can be avoided if there are fewer processes in the group and all can act as Acceptors. To avoid multiple proposal requests simultaneously, a distinguished process can be chosen which will be the only process (Proposer) issuing the proposals to the Acceptors. Other processes can send their requests to the leader. The proposals can be stored in a persistent storage like a database so that crashed process can recover easily on restart. The failure models supported by Paxos and corresponding performance characteristics is given in the Table 2.1.

v. Raft (Ongaro and Ousterhout, 2014) - Raft is a consensus algorithm specifically designed to maintain replicated logs. Raft is described for a collection (cluster) of servers running replicated state machines. Each replicated state machine (RSM) maintains replicated logs consisting of a set of client commands that needs to be executed sequentially. The cluster of RSMs consist of follower, leader and candidate nodes. A follower accepts log entries (commands) for storage and execution from the leader. An elected leader accepts client requests/commands, determines the order of log entries (commands) and sends it to all RSM for logging into replicated logs and for execution later. The leader will commit the entry into its own log, after receiving acknowledgement of successful storage from a majority of RSM. In this way, the leader helps in

achieving consensus among the cluster of RSMs on the order of log entries into replicated logs. If a follower does not receive any heartbeat message from the leader, it starts a new leader election and proposing itself as leader. RAFT attempts to overcome the complexity inherent with Paxos algorithm.

vi. 2-Phase Commit (Lynch, 1996; Menascé and Nakanishi, 1982) - 2-Phase Commit is database transaction protocol where a set of operations that perform a single logical function are executed in sequence by only one processes at a time (e.g, making airline reservation, withdrawing money from account etc.). The protocol can be considered as a variation of consensus problem with weak termination properties. The weak termination property states that termination is guaranteed only if there are no failures. In 2-phase commit, all communicating processes decide whether to commit (1) or not commit (0). Each processes proposes its local proposal to a distinguished process (leader) to make the decision. If any process proposes 0, then the decision is 0. If all processes propose 1 and if there no failures, then the decision is 1. The algorithm executes in two rounds.

Round 1: All processes send their respective proposals to the leader.

Round 2: The leader makes a decision and broadcasts the decision to all processes.

The protocol does not ensure guaranteed termination if the leader fails, since the follower processes cannot decide on 0 or 1. This is because they do not have knowledge of what the other processes have proposed. This drawback is fixed in the next algorithm.

vii. 3 Phase Commit (Skeen and Stonebraker, 1983) - 3 Phase Commit problem is an extension of 2 Phase Commit problem with strong termination properties. Strong termination ensures that the leader does not commit to a decision until all the pro-

cesses know of the decision (even in presence of faults).

Round 1: All processes send their respective proposals to the leader.

Round 2: Leader broadcasts 0 to all processes, if decision is 0. Else, it broadcasts a "tentative 1" to all processes if receives all 1s.

Round 3: If a process receives decision as 0 from the leader, it records it and algorithm ends here. But if a process receives "tentative 1" from the leader, it records it and sends an acknowledgement to the leader. By end of the round, leader would have received confirmation from all process and decides on 1.

Round 4: Leader broadcasts 1 to all processes.

If the leader fails during any of the first 3 rounds, a new leader gets elected and it continues from step 2. Hence, the algorithm takes anywhere from 3 to 6 rounds.

A summary of Consensus algorithms described above with different failure models supported and performance is provided in the Table 2.1. The parameters of the table are:

- C = Crash Failure
- O = Omission Failure
- B = Byzantine Failure
- N = Number of processes
- F = Number of faulty processes
- K = Max possible decision values in K-set algorithm
- T = Message delay

Table 2.1: Consensus Algorithms Summary

Name	**Failure Model**	**Message Complexity**	**Time Complexity**
Simple Consensus	C, O, B	$N(F+1)$	$F+1$
ICA	C, O, B	$N(F+1)$	$F+1$
K-set	C, O, B	$F/K+1$	$F/K+1$
Paxos	C, O, B	$(2F+1)(N-3)$	4
2 Phase Commit	None	$2(N-1)$	2
3 Phase Commit	C	$3(N-1)+1$	$3-6$

For applications which need to decide on single value, simple consensus is the best consensus algorithm. ICA is best for applications which need all the processors in a group to have consistent view of each proposed value to perform some computation. For example, ICA can be used for leader election to get a consistent view of all the proposed votes in order to make a uniform decision to elect a leader. Suppose a leader needs to be elected based on the process having the least computational (work) load in the group. This can be done using ICA where each process broadcasts its percentage of workload to the group. At the end of ICA, all the processes have a consistent list of all proposed values. All processes can make uniform decision to elect the process with the lowest proposed value as the leader. These two algorithms can be a good fit for decentralized power applications. Paxos, Raft, 2-Phase and 3-Phase commit protocols are best applicable for distributed database applications or for state machine replica applications. They may not be well-suited for decentralized power applications.

2.4.2 Leader Election

A leader is a distinguished process in a group which makes decisions or calculations for the group. Rest of the processes follow the commands of the leader. It is better to elect a leader dynamically instead of a static fixed leader. A fixed leader is an easy target for security attack as the identity is always known. Instead a leader (static or rotating) can be elected dynamically by all processes in the group. All the other nodes (followers) in the group can periodically monitor the status of the leader using heartbeat mechanism to detect if a leader has failed. If a node detects that the leader has failed (unreachable or not sending data etc.), it can start a new leader election. Once a new leader is elected, the system can go back to normal operation. This kind of failure handling mechanism may not be possible in applications using fixed leaders. This is another feature of DC that can be useful for decentralized power applications where generally there is a fixed leader (for example, a regional control center) capable of doing computations and control actions for a group comprising of generation units, sub stations etc. The computation and control action software may be available only in these fixed leader sites. So, it is easily vulnerable to failure and attack. Instead, identical pieces of software can be installed in all communicating entities in the group and a leader election algorithm can be used to elect a distinguished process to make decisions for the group. Upon leader failure, a fresh election can be initiated or a secondary fall back leader can take over the leader activities.

Leader Election DC algorithms should adhere to following properties:

- Safety - A participant process P_i has $elected_i = \perp$ or $elected_i = P_l$ where P_l is the leader of the election.

- Integrity - All processes in the group participate and eventually elect a leader or crash.

In a large group, there are several ways to elect a leader. The methods differ in complexity, design and effectiveness. Some of the popular Leader Election algorithms are described below:

i. Ring-Based Election (G. Coulouris and Blair., 2011) - This is suitable for systems where collection of processes are arranged in a logical ring. Each process has communication channel only to its immediate neighbor and sends messages in clockwise direction. Each process in the election will be either a non participant, participant or coordinator. There are two types of messages exchanged: *election* and *elected*. Initially at the beginning of election, every processor is a non-participant. The election is executed in two rounds.

   Round 1: A processor sets itself as participant and adds its own ID to the *election* message and sends to the adjacent processor. Each processor checks if its own ID is greater than identifier in the election message. If yes, it replaces its identifier in the *election* message and forwards to the next processor. In either case, it marks itself as participant. This process continues, till all the nodes in group participate in the election. By end of this round, the *election* message will contain the highest identifier.

   Round 2: When the election message reaches the processor with the highest ID, it marks itself as the coordinator and adds its own ID in the *elected* variable of *elected* message and sends to next processor. When *elected* message reaches each of the processors, it marks itself as non-participant and saves the *elected* variable in the *elected* message as coordinator ID.

   This algorithm is not an efficient algorithm because it works on the assumption that there are no failures. The worst case scenario is when the processor with highest

identifier is anti-clockwise neighbor of the processor that initiates the election. In such a case, it has a very high turn-around time of $3N - 1$ messages, where N is total number of messages.

ii. Bully Algorithm (Garcia-Molina, 1982) - The algorithm works in presence of crash failures and appropriate for synchronous systems because it uses timeouts to detect if the leader has failed. It uses three types of messages. They are:

- *Election* To announce start of election.
- *Answer* − Sent in response to election message.
- *Coordinator* − Message containing ID of the coordinator.

When a process detects that the coordinator has failed, it starts the leader election by sending *Election* message to all processors with higher identifiers. It then waits for *Answer* messages from all the processors. If no *Answer* message is received, then it can appoint itself as coordinator and send *Coordinator* message to rest of the processors. When a higher ID processor receives *Election* message, it sends back *Answer* message and starts a new election (by sending election message to processors with higher IDs). The processor with the highest ID, directly elects itself as coordinator and sends *Coordinator* message to all the processors with lower IDs. When a process receives *Coordinator* message, it sets its elected variable to the coordinator ID. When a process receives an *Election* message, it sends *Answer* message and starts a new election. Here, multiple simultaneous elections will be on-going concurrently.

It is called "Bully" algorithm because if a crashed process (leader) is restarted, it can announce itself as the leader and overthrow an already existing coordinator. This algorithm does not guarantee safety property for the case where a crashed leader is

restarted with the same ID. The restarted process (having highest ID) and newly elected leader can announce themselves as coordinator simultaneously. If there is no guarantee on message order, the recipients can reach different conclusion on the coordinator ID.

iii. Dijkstra and Scholten LE Algorithm (Vasudevan et al., 2004) - The algorithm is for mobile *ad-hoc* networks where topology is changing continuously with nodes joining or leaving a network continuously. The goal of the algorithm is to elect a leader who is the most-valued node among all the nodes in the group. The value of a node is a performance related characteristic like battery life, computational capability etc. The mobile *ad-hoc* network is modeled as undirectional graph which is changing as nodes move. The vertices in the graph correspond to the nodes and edges correspond to the communication link between them (i.e., the nodes are close to each other and can communicate with each other).

The leader election algorithm operates as a growing and shrinking spanning tree with the node that initiates the leader election at the root node of the tree. The election algorithm consists of five messages. They are:

- *Election* − Election start message.
- *Ack* Message to acknowledge receipt of election message.
- *Probe* Probing message to check adjacent node is still connected.
- *Reply* Message sent in in response to *Probe* message.
- *Leader* Message containing the leader id.

When a node detects that the leader has disconnected from the network or has failed, it starts leader election by sending *Election* message to adjacent nodes. When a node

j receives *Election* message from node i it sets node i as the parent and forwards the message to all adjacent nodes (children) except to the parent node. This process continues until all nodes in the group are covered. When a node receives *Election* message, it has to send *Ack* message to its parent. But it holds off sending the *Ack* message, till it receives *Ack* message from all its children. The *Ack* message contains the id and value of the most valued node among all its down-stream nodes. Once all *Ack* messages are received by the root node, it sends *Leader* message containing the ID of the leader (most-valued node), which is eventually propagated to all the nodes in the spanning tree. It is necessary to detect nodes that get disconnected from the network because the topology is continuously changing in mobile *ad-hoc* networks. Otherwise, the nodes keep waiting for *Ack* messages from its children indefinitely. In order to detect network topology changes, each node periodically sends *Probe* messages to all its neighbors. When a node receives *Probe* message, it sends *Reply* message in response. If *Reply* message is not received from a node within specified timeout, it is removed from the neighbor list implying that the node has left the group.

It also handles cases of partition merges, where two groups/components each having unique leader are merging together. In such a case, a fresh election is started and new leader (most-value node in the merged group) is elected. The message overhead can be anywhere from $2-3$ broadcasts and $0-3$ unicasts.

iv. TORA based LE Algorithm (Malpani et al., 2000) - The algorithm is proposed for mobile *ad-hoc* mobile networks subjected to frequent topology changes. The algorithm is able to detect disconnection (due to partition or crash failure) from the current leader and start a new election. The nodes in a component are connected as a directed

acyclic graph (DAG) where destination is the only sink (no outgoing link). Each node maintains a height, a tuple consisting of (l_{id}, r_i, oid_i, r_i, d_i, i) where l_{id} is the leader ID, r_i is the time when partition was detected by a node, oid_i is the ID of the node (originator) that detected the partition, r_i is the reference level, d_i is the distance from the originator and i is the node identifier. Each link between two nodes is directed from node having higher height to node having lower height. When a node detects a partition, it declares itself as the new leader of the component/group and updates its height with new reference level and new leader ID. It then sends an *Update* message containing new height to its neighbor nodes which in turn update their own height with respect to sender height. The *Update* message is then forwarded to its neighbor nodes and the process continues till all nodes in the component realign their links and become aware of the new leader ID.

Table 2.2: Leader Election Algorithms Summary

Name	**Failure Model**	**Message Complexity**	**Time Complexity**
Ring based	None	$3N-1$	$2-3$
Bully	C	Best Case: $N-2$, Worst Case: $O(N^2)$	1 to $N-2$(elections)
Dijkstra and Scholten based	C, P	$4N$	4
TORA based	P	$2-3$ broadcast and $0-3$ unicast	3

A summary of Leader Election algorithms described above with different failure models supported and performance is provided in the Table 2.2. Handling Byzantine failures in leader election does not make sense because if elected leader is byzantine then the followers will not agree on the issued commands. The parameters of the table are:

- C = Crash Failure
- O = Omission Failure
- N = Number of processes

If the application requires leader election based on highest processor ID, then Bully algorithm seems like a good fit. The last two algorithms can be applied for mobile *ad-hoc* network applications such as e.g., vehicle *ad-hoc* network, where vehicles are joining and leaving a group frequently. However for a static network, where a leader needs to be elected based on an application specific value, the best option is Interactive Consistency Algorithm. This is because all the local values can be collected at each node and any application specific computation (e.g., cyber-physical criteria) can be applied on the list later to select the leader node.

2.4.3 Ordered Multicast

When building distributed system applications, it is important to have all coordinating entities in a group to have consistent view of all the messages received in the group, group member failure messages and group membership change messages. One of the key aspects to ensure that the processes work in tandem is to have a reliable multicast communication that delivers messages to the group in the same order and with atomicity (all processes get a message or none do) despite multiple concurrent senders. This simplifies higher level code and reduces the risk of distributed processes performing inconsistent actions.

Reliable Multicast is one that satisfies the following properties:

- *Integrity* − A correct process p delivers a message m at most once.

- *Validity* − If a correct process multicasts message m, then it will eventually deliver m.

- *Agreement* − If a correct process delivers message m, then all other correct processes in group(m) will eventually deliver m.

Reliable multicast can be broadly classified on the type of message ordering semantics that it provides. They are described below:

i. FIFO Ordering - If a correct process multicasts a message m and then message m to a group g, then every correct process in the group first delivers message m and then message m'. Here, the message ordering is with respect to a single sender. This is achieved by maintaining a sequence number for every message sent and received from a process. At the receiver end, a FIFO queue is maintained for each sender process. When a sender p wants to send a message, it first increments its sequence number and piggybacks it along with the message and multicasts the message to the group. When a receiver q receives a message from p, it checks the received sequence number against the sequence number it maintains for the sender. If it is same, it immediately delivers the message to higher level. Else, it is put in FIFO queue, until missing message arrive. Birman and Joseph (1987) defines this type of ordering semantics as *FBCAST*.

ii. Total Ordering - Here, if a correct process delivers message m before m' then all correct processes deliver the messages in same order i.e. m before m'. In total ordering, the message delivery can be ordered arbitrarily (i.e., message delivery is not based on happened before relationship) as long as all messages are delivered in same order. Birman and Joseph (1987), defines multicast operation with this type of ordering

semantics as *ABCAST*.

Total order delivery algorithm is similar to FIFO ordering algorithm. The difference compared to FIFO ordering is that the processes keep group specific sequence numbers instead of a process specific sequence numbers. Total ordering can be accomplished in two ways. One, there is a global sequencer which assigns global sequence numbers to each message in the group. Two, ordering method as implemented in ISIS for *ABCAST* protocol (Birman and Cooper, 1991). There is no global sequencer, but instead, all processes coordinate with each other to agree on a global sequence number for each message in the group. A message can have two states: *deliverable* or *undeliverable*. Initially, a sender multicasts a message to all processes in the group. Each recipient assigns its own sequence number to the incoming message and adds it to the end of its priority queue tagging it as *undeliverable*. Each recipient proposes a sequence number for each incoming message to the sender. The sender collects all the sequence number proposals for the message. It then finds the highest proposal and broadcasts the value back to all the recipients. The recipient reassigns the sequence number of the message as per agreed upon value, resorts the queue as per its sequence number (and priority) and tags the message deliverable.

iii. Causal Ordering - If a message m is sent before m' i.e., $m -> m'$ where $->$ represents happened-before relation, then all processes in the group deliver the messages in the same order i.e., m before m'. Birman and Joseph (1987) defines the multicast operation with this type of ordering semantics as *CBCAST*. *CBCAST* protocol implemented in ISIS (Birman et al., 1991) uses vector time stamps to established causality between messages. Each process maintains its own vector time stamp. Before send-

ing message, process P_i increments its vector time stamp as $VT(P_i)[i] + 1$. When a recipient P_j receives the message, it places message in the hold back queue and does not deliver until following conditions are satisfied. For all $k = 1$ to n, where n is total number of processes,

$$\begin{aligned} VT(m)[k] &= VT(P_j)[k] + 1, \text{ if } k = i. \\ VT(m)[k] &\leq VT(P_j)[k], \text{ otherwise.} \end{aligned} \tag{2.1}$$

CBCAST can be extended into causal, totally ordered ABCAST protocol with the help of a token holder process which sets the order for all causal order and total order messages. Birman et al. (1991) describes the details of the protocol.

The above ordering semantics are applicable for basic multicast (which does not ensure atomicity) as well. But, it is better to describe it in terms of reliable multicast primitive, assuming we have a reliable communication network and robust DS platform.

Reliable Multicast with same message ordering and atomicity is necessary for consistent operation of distributed processes. To build robust decentralized power applications, reliable multicast communication with appropriate message ordering is necessary. Total ordering semantics is best, if application is not concerned with happened-before relationship of messages.

2.4.4 Voting

In voting, the processes choose one value from a set of values. This can include average functions such as mode, median, mean, and ALL-NOT-FAILED, as well as other functions. This can be implemented using Simple Consensus or Interactive Consensus Algorithm. All

the processes in the group can get consistent view of list of proposed values and apply these functions on the list of proposed values.

2.4.5 Mutual Exclusion

Mutual Exclusion is a mechanism to ensure that one and only one process is performing certain steps at a time (e.g, accessing a shared resource). The mutual exclusion algorithms must adhere to two requirements.

- *Safety* - At most one process may execute in the critical section (CS) at a time.
- *Integrity* - Requests to enter and exit the critical section eventually succeed.

Some of the distributed mutual exclusion algorithms are described below:

i. Ring based (G. Coulouris and Blair., 2011)- Here, the processes are arranged in a logic ring. A process having access to a *resource access* token can have access to the shared resource. The token is continuously passed from node to node in clockwise direction. If a process wants to use a shared resource, it has to first gain access to a resource access token. So any process wanting access to the shared resource, waits till it gets the token. After getting the token, the process enters critical section and releases the token only after exiting from the critical section. Any other process not interested in accessing the resource, just forwards the token to next node in the ring. This algorithm is not very efficient because the network bandwidth is continuously consumed since the token message is passed around even if no process needs access to the shared resource. It takes anywhere from 0 to N message transmission delay to gain access to the resource.

ii. Ricart and Agrawala's Algorithm (Ricart and Agrawala, 1981) - The algorithm uses logical time stamps to synchronize access to shared resource. The algorithm uses two messages. They are:

- *REQUEST* - message requesting access to critical section.
- *REPLY* - message sent in response to *REQUEST* message.

If a process wants to gain access of a shared resource, it sends a *REQUEST* message and waits for *REPLY* messages from all processes. If a process is holding the shared resource, it does not send *REPLY* message immediately and puts the *REQUEST* message in its request queue. After releasing the resource, it sends *REQUEST* message to the process in the front of the queue. Concurrent requests are handled by sending the *REPLY* message to the process having the lowest logical time stamp. The number of messages exchanged is $2(N-1)$, where N is total number of processes. The performance details are given in the Table 2.3.

iii. Maekawas Voting Algorithm (Maekawa, 1985) This algorithm is similar to Ricart and Agrawala's Algorithm but it reduces high number of messages by reducing the number of processes from whom it needs to obtain permission to enter critical section. Here, it needs to get permission from only K set of processes where $K = [\sqrt{N}]$ (N is the total number of processes). Each node has its own voting set $V_i = \{P_1, P_2 \cdots P_n\}$ and it should satisfy following requirement:

- $P_i \in V_i$
- $V_i \cap V_j \neq \varnothing$
- $\mid V_i \mid = K$

The processes in the intersection of two set of votes ensures that only one process enters the critical section at a given time.

The algorithm uses three type of messages. They are:

- *REQUEST* - Message requesting access to critical section.
- *RELEASE* - Message sent after exiting from critical section.
- *REPLY* - Message sent in response to *REQUEST* message.

The algorithm is executed in following steps:

(a) When a process wants to access a resource, it sends *REQUEST* message to its voting set and waits for *REPLY* from all the processes in its voting set.

(b) When a process in the voting set receives *REQUEST* message, it sends *REPLY* message back immediately unless it is using the shared resource or it has already sent *REPLY* message to another process.

(c) When a process exits the critical section, it sends *RELEASE* message to its voting set.

(d) When a process in the voting set receives *RELEASE* message, it sends REPLY message to the first process in its pending queue.

The number of messages exchanged is $3N$, where N is total number of processes. This algorithm is deadlock prone. To avoid deadlock, the *REQUEST* messages needs to be queued in the pending queue in a happened-before order. The performance characteristics is shown in the table 2.3.

iv. Suzuki-Kasami's Broadcast Algorithm (Suzuki and Kasami, 1985) - Each process maintains a request array RN[i] where $i = 1..N$ consisting of highest sequence num-

bers received from each of the other processes in the group. When process j wants to enter the critical section, it sets RN[j] = RN[j] + 1 and broadcasts a request message of the form $R(j, RN)$ where j is process ID. When a process i receives *REQUEST* message, it updates its own RN array (element wise) as maximum of two array elements (i.e., max of received $RN[i]$ and local $RN[i]$). If the receiving process is executing the critical section, it sends *PRIVILEGE* message after exiting from critical section. The *PRIVILEGE* message is of the form PRIVILEGE(Q, LN) where Q is list of requesting nodes, $LN[i]$ where $i = 1N$ is array of highest sequence numbers that have been granted access. It sets $LN[i] = RN[i]$ to indicate that the node has just finished executing critical section. The Q queue is updated if $RN[i] = LN[i] + 1$ indicating that there are few new nodes requesting the critical section. Finally the *PRIVILEGE* message is sent to the node at the head of Q queue. The performance characteristics is shown in the table 2.3.

v. FaTLease Algorithm (Hupfeld et al., 2008) - The algorithm is proposed for a fault tolerant shared resource lease negotiation for distributed systems (e.g., distributed file systems). Here, lease is a token which grants exclusive access to a resource for a finite duration of time. The algorithm internally uses Paxos algorithm but improvises it a bit by removing the dependence on persistence storage. It uses timeouts for leases and thereby avoiding the need for stable storage. This algorithm is proposed for tightly synchronized system with very small clock skew between communicating processes. The algorithm uses MultiPaxos where separate Paxos instances are used for reaching consensus on a particular lease for a particular resource. Each resource access is managed by one distinguished process. A process accessing the resource will

have acquired lease $\lambda(h_k, t)$ where h_k is process id, t is duration of lease period. The lease is valid upto the duration of t. The algorithm is basically reaching consensus on the proposed lease request λ. The algorithm has the same participants and uses the same messages as Paxos as described earlier. An additional message *Outdated* message is sent by Acceptor if it sees an outdated instance number in the Prepare request. The Proposer starts a consensus process by sending λ as proposal to all Acceptors. Paxos algorithm will choose one proposal during the consensus rounds and that becomes the current lease.

Round 1: This is Initiate Consensus round. Proposer creates a unique ballot number (proposal value) for instance i and sends a *Prepare* request message (b, i) to all Acceptors.

Round 2: When the Acceptors receives *Prepare* message, it checks if local instance number j is greater than received instance number i. If yes, it sends *Outdated* message. If no, it discards old instance and starts new instance (implies that Acceptor has missed some instances), updates local instance and sends ACK message. If $i == j$, it sends ACK immediately.

Round 3: If Proposer receives *Outdated* message, it starts MultiPaxos to find out current instance number and starts new consensus by executing Round 1 again. If it receives $NACK$ message, wait for sometime and initiate consensus again. If ACK message received from majority, then start the Accept phase.

Round 4: Proposer sends *Accept* message (λ, i, b) to all Acceptors.

Round 5: The operation is similar to Round 2 except that Acceptor is accepting the proposal.

Round 6: If ACK received for Accept request from majority $(H/2+1)$, then send the decision to all Acceptors using learn message. If $NACK$ received, restart again with *Prepare* phase (Round 1).

The drawback with this algorithm is time synchronization. If the clocks are not accurate as assumed, then the algorithm will not work. This is because two seemingly concurrent requests may be granted access simultaneously.

Table 2.3: Distributed Mutual Exclusion Algorithms Summary

Name	**Failure Model**	**Message Complexity**	**Time Complexity**
Ring based	None	N	Best case: 1, Worst case: N
Ricart and Agrawala	None	$2(N-1)$	2
Maekawa's Voting	None	$3N$	$2T$
Leslie Lamports Algorithm	None	$2(N-1)$ to $3(N-1)$	T
Suzuki-Kasamis Broadcast Algorithm	None	Best case: 0, Worst case: N	0 or T
FaTLease	C	$(2F+1)(N-3)$	Best case: 5

A summary of Distributed Mutual Exclusion algorithms with different failure models supported and corresponding performance details is provided in the table 2.3. Lamport (1978) describes granting access of a shared resource to a process having the lowest logical time stamp.

The parameters of the table are:

- C = Crash Failure

- N = Number of processes
- T = Message delay

2.4.6 Group Membership Management

It is often desirable for a distributed system to be divided into multiple groups, each performing a dedicated task. The processes within each group can coordinate with each other to perform the task. Birman and Joseph (1987) describes virtual synchronous environment as that which allows processes to be structured into process groups and make events like broadcasts to a group, group membership changes, and migration of an activity from one place to another appear to occur instantaneously in other words synchronously. That means, virtual synchrony makes non synchronous steps to appear as synchronous. Some of the key elements of group management are listed below.

i. Group Address Expansion - Each group can be assigned with a unique group identifier. It should be possible to multicast to a group by only using the destination group id in the multicast interface. The protocol must expand the group identifier into current membership and deliver to all members in the group in the current view.

ii. Group Membership - The group membership management service should provide methods to create/destroy a group of processes (formed for specific purpose), add or remove members from the group etc. Also, processes should be able to join and leave groups dynamically.

iii. View Maintenance and Delivery - Group view refers to list of processes currently in the group. It is important that all the members in the group have a consistent view of the group (i.e., list of active members, list of failed members etc.) so that they make

consistent decisions based on current membership state. Whenever a process joins or leaves a group (either by removing itself or by crash failure), the view of the group changes. All the members of the group need to be communicated of this new view concurrently so that all processes become aware of membership change at the same time. Two consecutive views $view_i(g)$ and $view_{i+1}(g)$ of a group g, differ by addition or subtraction of exactly one process. The properties of view delivery is similar to multicast delivery. All the processes deliver views in the same order. If one member in the group delivers a view, it implies that all members have delivered the same view (atomicity is maintained). Processes can learn of the failure of other group members through this view mechanism, instead of probing each process continuously to check for aliveness.

iv. Reliable Message Delivery with Message Order Sensitivity - All messages to the group should be delivered with the same order and atomically. The details about multicast communication and ordering semantics are described in Section 2.4.3.

v. Failure Monitoring - The group membership service can also detect group member failures. The member can crash due to software or hardware exceptions or become unreachable due to communication network failure. These can be detected by the group membership service by using some sort of heartbeat timeout mechanism. Any failures detected should be reported to all members in the group as a new view event. Any message from the failed process should be delivered before the view is delivered.

vi. State Transfer - When a new process joins the group, the view delivery management should transfer the current working state of the group (number of active, failed members) to the new member joining the group.

vii. Failure Recovery - Some tools/platforms also support failure recovery of failed processes. Failure recovery mechanisms such as Check pointing and Rollback mechanism and replication can be used. In Check pointing and Rollback mechanism, check pointing of data and commands are done at regular intervals. If any failure is observed, it is rolled back to last correct checkpoint. ISIS^2 (Birman and Joseph, 1987) uses a recovery manager which is able to detect totally failed processes like crash failures (as opposed to partially failed) and start a recovery process. Recovery manager waits for failed process to restart and join the group. After that, it transfers the current state of the group (and also perhaps replicated state of the process) to the process.

viii. Replication - Replication can be accomplished with active or passive replication. In active replication, each client request is processed by each active replica but the result is sent back by only one primary replica. So when primary replica fails, other replicas can easily take over. In passive replication, primary replica executes the client request and updates the state of the operation to back up replicas. When primary replica fails, other replicas can easily take over.

2.4.7 Group Discovery

Group Discovery is generally used by mobile computing entities to discover other computing entities or nodes in the neighborhood and become part of an existing group or form a new group. In the context of decentralized power applications, it can be useful provide peer-to-peer mechanism for a decentralized agent (e.g. electric vehicle) that wishes to support the grid (e.g., voltage support by a plug-in DER) to form a group (e.g. to support voltage in a given fairly small area). These groups may need to discover what other peers are "nearby" to be able to coordinate with these peer groups or possibly a parent group. If

required, a leader of the group can be elected to make decision for the group.

There are several algorithms for neighbor discovery in Mobile *Ad-hoc* Network (MANET) in the literature. But, the basic principle remains the same: To determine if a node is within communication range with other nodes. Two nodes are considered within communication range if they are separated by at most k hops and is stable for at least t amount of time (stable means it does not leave the region). The value of k can be 2, 3 etc. depending on the requirements of the algorithm.

i. Neighbor Discovery Algorithm - Malpani et al. (2000) proposes reliable neighbor discovery algorithm for Vehicular *Ad-hoc* Networks (VANET). In addition to `broadcast()`, `ack()` and `recv()` actions, it defines two new actions `linkup()` and `linkdown()` actions. If two nodes i and j remain at most k hops apart for a duration t (based on message transmission delay and link set up time), then $link(i, j)$ is considered Up. Whenever a node enters a new region and knows that it is going to stay in the region for sufficient time, it broadcasts a *join* message. When a node receives *join* message, it replies with *join reply* message depending on the time it is going to remain the region. When a node wants to exit a region, it broadcasts a *leave* message some time before it actually exits. The link tear down time should be sufficient for *leave* message and all the pending messages from the exiting node to get delivered to all neighbor nodes. Once, a group is formed using this neighbor discovery scheme, any of the leader election algorithm applicable for mobile *ad-hoc* networks described as in Section 2.4.2 can be used to elect a group leader for the group.

ii. Landmark Ad-Hoc Routing (LANMAR) method of Dynamic Group Discovery - Hong and Gerla (2002) is describes group discovery using Landmark Ad Hoc Routing (LAN-

MAR) routing protocol. Groups are considered as logical subnets comprising of set of nodes. Each subnet has a leader node called Landmark node. Each node maintains a local routing table consisting of list of neighbor nodes and corresponding number of hops to reach them. The maximum number of hops is set to N. A nodes group is list of all the nodes that are reachable within maximum of N hops and are stable in the region for more than M minutes. Once a group is established, all the nodes in the group decide on a leader based on the "weight" of node. The weight of a node is a parameter comprising of number of neighbor nodes, stable time and its local ID. The node with higher weight wins the election.

CHAPTER 3

OVERVIEW OF DECENTRALIZED POWER ALGORITHMS

3.1 Introduction

The power systems are becoming very complex due to the increasing penetration of DERs, and increasing application of IEDs into the field to get accurate phasor measurements. It is becoming increasingly difficult for traditional control centers to monitor and control the power system efficiently due to the intermittent nature of power generation of DERs, large set of system variables and increased data flow from the IEDs. The power system is operating very close to the power system margin and slight fluctuation is leading to tipping over the operating threshold. Traditional control centers may not able to provide control action fast enough in the future. As a result, there is growing need to move away from centralized architecture of power system applications to a more decentralized architecture so that some of the computations and control can be done locally instead of sending all the power data to the central control center. So far, the existing power applications are either centralized or completely local. The pitfalls of centralized applications are well known, namely single point of failure, single point of attack, bottleneck etc. Local control is not efficient either since it is based on local disturbances and with limited network visibility. Local control with limited system visibility may trigger a sequence of undesirable perturbations in the neighboring areas and may eventually lead to complete power system

collapse. San Diego blackout in 2011 is an example of local control with cascading effects.

Decentralized architecture offers viable solutions to alleviate the problems observed in local and centralized control. An optimal solution can be to have a decentralized architecture where a large power system is divided into multiple sub groups and each group performs monitoring and control individually with limited interactions with the control center or other adjacent groups performed within these groups. A subset of substations within a neighborhood form a group based on geographical distance and other power parameters. The nodes in the group exchange local measurement and coordinate with each other to provide fast control with increased topological visibility. Distributing the control helps in reducing the dependence on central control center and reducing the cascading of faults as observed in total local control.

Table 3.1: List of Decentralized Power Applications and Applicable DC Algorithms

Decentralized Power Applications	**Applicable DC Algorithms**
Distributed Voltage Stability	Group Membership, Leader Election, ABCAST
Distributed State Estimation	Group Membership, Leader Election, ABCAST
Distributed Remedial Action Schemes	Group Membership, Simple Consensus, ABCAST
Decentralized Wind Power Monitoring and Control	Group Membership, Leader Election, ABCAST
Distributed Frequency Control	Group Discovery, Group Membership, Leader Election, Supply Agreement, ABCAST
Decentralized Optimal Power Flow	Group Membership, ABCAST
Decentralized Reactive Power Control	Group Membership, Interactive Consistency Algorithm
Decentralized Inverter Control	Group Membership

Hitherto, there have been several decentralized power algorithms proposed for targeted power application scenarios. It is seldom possible to develop a single distributed control

algorithm that is suitable or applicable to all decentralized power applications. There has been no single generalized solution developed to mitigate the problems in power applications. However, many decentralized algorithms have been proposed to address the needs of specific power applications. Some examples of such algorithms are provided in this chapter. For each decentralized power application scenario, a possible solution using some of the DC algorithms described in Chapter 2 is also suggested. A summary of decentralized power applications and applicable DC algorithms for each scenario is shown in Table 3.1.

3.2 Decentralized Voltage Stability

The voltage stability monitoring and control is the ability of a power system to maintain steady acceptable voltages at all buses in the system at both, normal operating conditions and after being subjected to a disturbance. Voltage instability occurs when the receiving end voltage falls below normal voltage due to heavy load, sudden fluctuations in the load, faulty transmission lines or shortage of reactive power. It has to be restored back by control actions such as transformer tap change, injecting reactive power or finally load shedding if former two actions fail. The problem has be resolved quickly or it might lead to more drastic problem of complete voltage collapse resulting in blackouts. The voltage stability problem is being observed more often now since the power grids are being operated closer to the stability point due to growing economic demand. Increasing number of renewables being used in the power grids and limited reactive power are also additional factors for frequent voltage stability problems. Since centralized control has high response time and voltage stability problem is inherently local, it can be solved locally using available reactive power resources in the neighborhood. As a result, there has been significant research in recent years to come up with decentralized solutions to mitigate voltage stability problem.

Moradzadeh et al. (2013) proposes a distributed cooperative Model based Predictive Control paradigm (MPC) for long term voltage control of multi-area power systems. Here, the local control action is based on local model of its own area and reduced order quasi steady-state models of its immediate neighboring areas. The planned local control sequence is then sent to the neighboring control areas which is taken into account in the next control sequence iteration.

Based on this behavior, it seems distributed groups of neighboring control areas can be formed and local information observed at each substation can be communicated with each member of the group (or to a group leader). All the nodes within group can coordinate with each other and make uniform decision on the control action based on the group-wise network visibility. Only limited information (e.g, tie-line information connecting two substations in adjacent groups) needs to be exchanged with neighboring groups. DC algorithms like group membership to form distributed groups and leader election algorithm to elect one distinguished node to make decisions for the group can help provide robust solution for this problem. The applicable DC algorithms are summarized in Table 3.1.

3.3 Decentralized State Estimation

State estimation (SE) is a very critical application for power system. SE extracts information from huge amount of (often unreliable and redundant) raw measurement data from meters and sensors, and converts it into a reliable state estimate of the power system for the Energy Management System (EMS) applications. It acts as a pre-filter for all other EMS components and its accuracy, reliability and robustness will impact the whole system performance. The output of SE is an input for other power system operations like online load flow, congestion management, load forecast, stability analysis, security assess-

ment etc. Distributed voltage stability application described in previous section also uses SE. Currently, all the measurement data is being sent to the central control center for state estimation. The execution time of SE is very critical as many applications are dependent on it. With the increasing complexity of the power system and integration of more control devices in the power system, the traditional control state estimation is taking additional time to converge or may not even converge. To overcome this problem, Decentralized State Estimation (DSE) is being considered and a brief description of few of them follows.

In Van Cutsem et al. (1981), a hierarchical 2-level SE is proposed, where sensors at the edge of the power system network send their measurement data to Local State Estimators (LSE). LSEs perform local state estimation and send their estimates to Central State Estimator (CSE) for final estimation. Korres (2011) describes multi-area distributed state estimator where the whole power system is divided into multiple non overlapping regions/areas. Each area performs local estimation based on local measurements and send it to central coordinator. The central coordinator performs system wide estimation based on boundary measurements and state estimates received from individual control areas. It proposes splitting the system into multiple areas based on observability and controllabilty to perform decentralized state estimation within each area. But, it does not talk about how the communication of bus and line data is achieved within each area and across multiple areas (for tie-line data).

Distributed State Estimation using Message Passing algorithms was proposed in In Li (2012). It is assumed that in the future, devices like sensors, IEDs, meters will be equipped with enough computational power to run DSE algorithm and can communicate with neighbor nodes using message passing. Here, traditional mesh network used in power systems is converted into spanning tree type of network. Local state estimates are exchanged

between neighboring nodes and further state estimates are calculated based on local and neighbor information. Xie et al. (2012) proposes fully distributed state estimation using iterative method, where again the whole power system is split into multiple areas, and each control area performs local estimate based on local measurements and exchanges state estimates between neighboring areas. New state estimates are computed based on new measurements, past estimates and estimates from neighbor areas. This goes on till the solution converges into a complete system wide estimate after finite number of iterative steps.

All of these algorithms concentrate on describing the power algorithm for DSE. But, they have not addressed computational and communication requirements of implementing these DSE algorithms. For example, they do not explain about how the multiple control areas are formed or how the communication between different entities is achieved. Such gaps can be filled by using DC algorithms described in Chapter 2. A robust solution using DC algorithms is to form distributed sub groups (or multiple control areas) using group membership algorithm, where each group is capable of performing DSE separately with minimal communication with central control center or neighboring sub groups. A lead node for a group can be elected using leader election algorithm. The nodes within the group (PMUs, sub stations, etc.) send their local phasor measurements periodically to the lead node. The adjacent group lead nodes exchange only the tie-line information. The lead node of each group executes DSE algorithm using the local phasor measurements and tie-line data. The applicable DC algorithms for DSE application are summarized in Table 3.1.

3.4 Decentralized Remedial Action Schemes

Remedial Action Schemes (RAS) are Special Protection Schemes (SPS) that get triggered in the advent of extreme abnormal conditions to protect the affected region of the power grid from complete power outage. RAS provides this reliability by monitoring conditions of buses and transmission lines and taking predefined corrective actions when severe abnormal conditions or contingencies are detected. Some of the contingencies and corresponding control actions are listed in Table 3.2.

Table 3.2: RAS Contingency and Control Action

Abnormal Condition	**Action Taken**
Over frequency	Generator dropping
Equipment over load	Generator dropping, System separation, Load dropping
Angular instability	Generator Excitation Forcing, Capacitor reactor switching, Advanced reactive support, Load dropping
Under frequency	Load dropping
Under voltage	Generator Excitation Forcing, Capacitor reactor switching, Advanced reactive support
Overvoltage	Advanced reactive support, Open end line tripping

The critical control actions listed above should be taken within few milliseconds. Typically, RAS schemes are configured during installation, and are expected to operate after many years. The change in network topology and variation in power system margin may cause the pre-configured RASs to mal-function. This was evident from the San Diego September 8, 2011 blackout, where S-line RAS designed with old topology settings (i.e., designed to operate for one tie-line) and not reconfigured with latest topology settings (a new tie-line in the same area was added), was triggered. So the S line RAS performed control action ignorant of its effect on the other tie-line. This had a cascading effect and finally led to

complete blackout of San Diego region. This was one of the most important observations made in the post analysis FERC report (Report, 2012). A dynamic RAS scheme capable of detecting changing topology and updating the logic parameters dynamically as per changing topology has been studied and demonstrated. Another important observation in the report was that a better coordination between S-line RAS and other neighboring protection schemes could have prevented the cascading effect. So, apart from being dynamic, the RAS should be made distributed and local. The RASs in the neighborhood can form a group and communicate with each other to take an informed collective action based on the local and neighboring operating conditions to minimize the impact and to efficiently restore normal operation of the system.

For example, if a RAS detects a local contingency like under current in one of its transmission lines, it can initiate a simple consensus run with other RASs in the neighborhood to trip a specific part of the load with "ALL OR NOTHING" type of decision criteria. Other RASs (or other protection schemes) in the neighborhood can agree or disagree to trip the load based on operation condition of its region. The affected RAS can go ahead and perform the proposed control action, only if all RASs in the neighborhood agree with the proposed control action. This type of coordination among neighboring RAS can be easily implemented with the help of DC algorithms. DC group membership algorithm can be used to form a group of neighboring RASs (assuming substations connected to RASs have a computing device to run the DC software). The neighboring RASs within the group can coordinate with each other using simple consensus algorithm to decide on a control action for a particular contingency. The decision criteria will be to check if all RASs have proposed/voted "Yes" to a proposed control action. The applicable DC algorithms for Decentralized RAS application are summarized in Table 3.1.

3.5 Decentralized Wind Power Monitoring and Control

Wind power generation has become an important additional source of power generation in smart grids. The transmission grid usually works on standard line ratings for ampacity (maximum current carrying capacity) which are predefined based on offline studies and with conservative assumptions. But the intermittent nature of power generation from wind farms (based on wind speed), may cause standard ampacity of the small load transmission lines to get exceeded and trigger unnecessary control action. Instead, Dynamic Line Rating (DLR) which calculates the ampacity dynamically and apply line ratings in real time based on actual operating conditions are being used in the field now. DLR calculates the line rating based on factors such as current weather condition (e.g. wind speed, solar radiation, and ambient temperature), line loading and other conductor related properties (e.g. conductor size, sag temperature). This accurate assessment of ampacity of transmission lines helps in maximizing current carrying capability of the transmission lines and optimizing (operating closer to DLR rating) the utilization of transmission grid. Yip et al. (2009) describes how DLR is used by control center to monitor the transmission lines and control the wind power generation remotely. DLR line sensors are placed on the transmission lines to collect measurements like conductor temperature, sag temperature etc. and the measurements are continuously fed to the DLR server software in the control center. Weather stations close to these transmission lines also feed the weather data continuously to the load management system in the control center. The DLR server software calculates DLR based on the measurements from DLR line sensors and weather sensors. Subsequently, SCADA and load management system uses the calculated DLR to monitor and control the transmission lines and remote wind power generators. The load management system uses DLR in combina-

tion with the measured line parameters (voltage, current) to check if any of the lines are overloaded. If a particular line is over loaded, it sends a signal to the remote wind power generator connected to the transmission line to curtail the power generation.

This scenario is another candidate for decentralized power application. Small pockets of wind farms can be grouped together based on geographical distance into separate decentralized control zones. A fixed leader (master) or an elected leader for each zone can perform monitoring and control of the wind power generation within that zone. Again, DC algorithms like group membership and leader election can be used to manage group membership and to elect a leader. The applicable DC algorithms for Decentralized Wind power monitoring and control application are summarized in Table 3.1.

3.6 Distributed Frequency Control

Frequency control provide mechanisms to reduce the frequency deviations from scheduled frequency so that the balance between load and generation is always maintained. It is mainly carried out with three types of control: primary, secondary and tertiary control.

- *Primary frequency control* is an immediate local automatic control action of controlling the operation of generation side controllers in response to local disturbances (either sudden imbalance in load or generation). The control action consists of changing the power output of a certain generating unit (by reducing or increasing the generator speed) in response to a deviation in frequency from its set value. An adequate primary control reserve must be available at the respective generating unit at all times in order to provide this service. The steady state frequency should always be maintained at nominal value of 60Hz and hence frequency regulation needs to be done every 60 seconds.

- *Secondary frequency control* is used to always maintain the steady state frequency at standard nominal value of 60Hz. This is generally performed by a central control unit called Automatic Generation Control (AGC) system for a region. In smaller systems or micro grids, secondary control system is integrated into the generators itself. AGC calculates set points for active power outputs based on the frequency deviation and other economic related cost functions and sends it to the generation units for governor control. The set points are used to adjust the production levels and to correct the frequency offset. For example, generators will be asked to increase their output to provide regulation up and conversely as load decreases the generators will be asked to decrease their output to provide regulation down. As generators respond to changes in their reference set point, system frequency will begin to return to 60 Hz and in turn governor action will cease.

- *Tertiary frequency control* is activated when the frequency deviation in the control area lasts for longer than 15 minutes. Tertiary control is used to provide additional reserve if the secondary control reserve is not sufficient. It relieve secondary control so that it is once again free to support or restore primary control whenever necessary. Tertiary control can be manual or automatic.

The fluctuations in the voltage and frequency will become more frequent in future power grids due to increased usage of DERs, storage capable loads like EVs due to their intermittent characteristics. The unpredictable behavior of loads is also a major factor for constant frequency deviations. Frequency and voltage fluctuations need to be stabilized in a very short time in order to prevent catastrophic damages such as blackouts. It will become increasingly difficult for the control center, AGC and the generator side control alone to

manage the frequency deviation and restore stability of the system within such a short time. Recent research explores the idea of using demand side response from prosumers (producers and consumers) like EVs, building clusters to help in frequency control and mitigation of frequency deviation locally.

In Moghadam et al. (2013), frequency control algorithm using EVs via controlled charging or discharging of power from or to the grid is proposed. Here, each EV monitors the frequency deviation separately and responds by either charging, discharging or going idle. If the system frequency is higher than nominal value, then each EV can help in decreasing frequency by either stopping its on going discharging activity or start charging from the power grid. Similarly, if the system frequency is lower than nominal value, then each EV can help in increasing frequency by either stopping its on going charging activity or start discharging power to the grid. But, if all EVs respond simultaneously, it can lead to frequency oscillations. To avoid this, EVs response to frequency deviation is randomized using Poisson exponential distribution technique.

Present decentralized frequency regulation is completely local, based on local disturbances and local measurements without considering its effects on neighboring areas. This may lead to inter area oscillations and further system instability. In Nazari et al. (2014), decentralized frequency control using prosumers is proposed. It uses model predictive control method with interactions and information exchange limited to only neighboring prosumers. The optimization problem at prosumer i is solved iteratively by adding coupling constraint parameter: perception of prosumer i based on control action of prosumer j. This perception parameter is exchanged in every iteration step and used in the next calculation in the next iteration. In this way when a local disturbance is observed, all prosumers in the neighborhood coordinate to stabilize the frequency.

Decentralized Frequency control using DC algorithms can be formulated in the following way. Prosumers like EVs, building clusters and micro grids can form a group based on the availability, geographical distance, response time and impact time. The prosumers can use group discovery DC algorithms to "discover" nearest group when it gets connected to the power grid. A leader election can be used to elect one distinguished prosumer to make decision about the control action sequence for the group. Each EV can provide its vector of possible energy contribution over set of discrete time intervals to the leader of the group. The leader can decide on the energy contribution by each prosumer for each/some discrete time interval and inform the decision to all other prosumers in the group. The prosumers can monitor the frequency deviation independently and respond by either charging or discharging or going idle based on agreed upon transaction behavior. The applicable DC algorithms for Decentralized Frequency Control application are summarized in Table 3.1.

3.7 Decentralized Optimal Power Flow

The optimum power flow within the power system can be maintained by adjusting the settings of the Flexible Alternating Current Transmission System (FACT) control devices. FACT devices are devices which are able to enhance AC system controllability and stability and to increase power transfer capability. Some of them are Static VAR Compensator (SVC), Thyristor-Controlled Series Compensation (TCSC), etc. The optimal set points of these FACT devices are determined by an objective function whose objective is to enhance the steady state security of the system by improving the voltage profile and keeping the currents below their limits. The parameters involved in the calculation are

i. State variables like voltage magnitude and angle of the buses.

ii. Control variables like the settings of the FACT devices.

iii. Soft constraints.

iv. Transaction cost (i.e., electrical energy import price) between neighboring areas.

In Hug-Glanzmann and Andersson (2009), decentralized optimal power control for multi area power system is described. Here the power system is divided into multi-areas where each area is defined based on sensitivity analysis of the FACT devices, i.e., the influence of the FACT device over the region. This may lead to overlapping areas and coordination between neighboring (non-overlapping or overlapping) areas is required to avoid conflicting control settings. The optimal power setting of FACT devices in each area is determined by solving the Decentralized Optimal Power Flow (DC-OPF) objective function in iterative manner until it reaches convergence. The objective function is solved in each iteration by using the past and current state variables and constraints of the local area and neighboring areas. But the communication of these variables within the local area and neighboring areas is not considered.

In Biskas et al. (2005), DC-OPF is implemented using a network of work stations using parallel processing. Each work station has the Parallel virtual machine software which uses message passing model to communicate with other work stations in the network. The work stations are structured in hierarchical fashion with bottom layer consists of work stations assigned to regional Transmission Service Operator (TSO) and are responsible for managing the transmission of their own local region. These are connected to a master work station representing Super-TSO which checks for convergence of the whole system. Each work station exchanges tie-line flows, tie-line electricity energy export prices and boundary bus voltage parameters with its neighboring area. It solves the area wise DC-OPF objective

function in iterative manner by using local parameters and external parameters until local DC-OPF solution is reached. Local convergence is reached when the tie-line flows on both sides of the areas converge within tolerance level. This is communicated to Super-TSO which determines system wide convergence i.e., when all local regions or area have reached convergence.

This is one of the rare papers where the communication aspects of multi-area control is discussed. Similar implementation using DC group management features can be used to form multiple control areas. Supply agreement algorithm can be used where export prices for each time interval can be exchanged by neighboring areas to speed up the DC-OPF solution.

3.8 Decentralized Reactive Power Control

Reactive power control is used to minimize the power losses and maintain the acceptable voltage profile of the power system. The VAR control equipment like shunt capacitors are used to improve the voltage profile and thereby enhancing the power quality and overall system reliability. The increased penetration of DG at the distribution feeders has led to greater fluctuations in the voltage profile (due to variable power output) along the feeders and interrupting voltage sensing capabilities of the capacitor banks at the feeders. So there is a growing need for coordination between the capacitor banks and the DGs. Elkhatib et al. (2012) proposes a coordination algorithm for decentralized reactive power control by placing RTUs with digital computing power at each of the substation, DGs and capacitor banks. The RTUs are represented in a tree structure where the station is at the root of the tree. Each RTU determines its voltage profile using the voltage measurements of its busses, active and reactive power flow of the lines connected to these buses and the voltage

values of its neighbor buses. Capacitors RTU also calculates the losses index for different combinations of reactive power injections. Then each RTU sends the calculated maximum feeder voltage, minimum feeder voltage to the upstream RTU and finally to the station RTU. The station RTU checks if there is need for improving the voltage profile (by checking the overall minimum voltage), determines optimal reactive power injection which constitutes to minimum losses and sends to the downstream RTU and finally to the capacitor to take action. If there are multiple capacitors in the distribution feeders, similar coordination to determine the best combination among multiple capacitors for optimal reactive power injection to maintain minimum losses is also explained in the paper.

For this power application example, all the RTUs along the feeder can form a group and each RTU determines its local voltage profile and exchanges its calculated voltage (minimum and maximum) to its neighbor for next iteration. Interactive Consistency Algorithm (ICA) can be used to get all possible reactive power injections available in the group. At the start of the algorithm run, each Capacitor RTU publishes its proposal of possible reactive power injections to the group. At end of ICA, all RTUs will have entire list of possible reactive power injections. It can then use this to determine the optimal reactive power control with minimum losses to improve the voltage profile of its region.

3.9 Decentralized Inverter Control

The frequency and voltage parameters of the inverters in the distributed generation (DG) units need to be controlled efficiently for system stability. Droop control is a strategy to control the frequency and voltage of the generators by adjusting the active and reactive power. In traditional droop control, the droop characteristics of synchronous generators are maintained by local estimates of real and reactive power. In Liang et al. (2013), the

stability issues with traditional droop control strategy of decentralized inverters due to limited visibility is addressed. It proposes power sharing based control strategy where total active and reactive power information is shared among adjacent inverters via wireless communication. The difference between actual and desired sharing of real and reactive power is used to adjust reference frequency of the inverters. If there is huge difference, it is adjusted more else it is adjusted less, until desired sharing is achieved.

Here, group membership DC algorithms can be used to form a group of inverters. Each inverter publishes its active, reactive power information to adjacent inverters or to the whole group.

CHAPTER 4

DESIGN OF DCBLOCKS

4.1 Introduction

Distributed Coordination algorithms have been proposed as a viable solution for many applications that utilize multi-sensor network facilities. One such application can be the power system applications as described in Chapter 3. Many DC algorithms in the literature offer solutions with different pros and cons. However, application specific design is still an important aspect so that the framework can help in making the application more robust. This chapter describes the design of Distributed Coordination Building Blocks (DCBlocks) framework. The collection of DC algorithms to be designed and implemented in DCBlocks is picked based on the trend of the current and emerging decentralized power applications as seen in Chapter 3. As described in Chapter 2, there are so many factors to be considered when building decentralized computing applications. A typical power engineer who has little knowledge of distributed computing and associated sub problems, will almost certainly miss the boundary cases when trying to coordinate the decentralized pieces of his or her power algorithm. This will in turn make it less robust in general and incorrect under certain network conditions. The goal of our DCBlocks is to exploit the vast body of theoretical and pragmatic research and build a robust set of building blocks comprising of these DC Algorithms to make them more user friendly for the power engineers. The design

of DCBlocks is not limited to mere adaptation of various decentralized algorithms but also making it fault tolerant with minimal performance trade-offs and customizing the interfaces for power applications.

Figure 4.1 shows the decentralized architecture for distributed power applications comprising of DCBlocks, power application logic and distributed software (DS) platform. Although the final aim of DCBlocks is to have individual building block for each sub problem described in Chapter 2, in this thesis, only design and implementation of Group Management, Ordered Multicast, Leader Election, Consensus (a.k.a Agreement) and Mutual Exclusion building blocks are considered. The distributed power applications can use any of these building blocks as per their application needs. Each software layer shown in Figure 4.1 is described in the following sections.

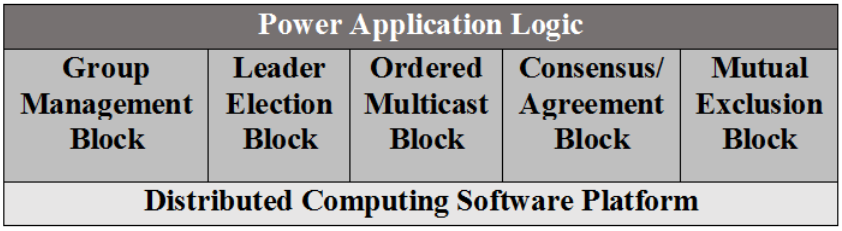


Figure 4.1: Distributed Coordination Software Stack using DCBlocks.

4.2 Distributed Software Platform

This layer handles the actual communication between communicating processes in a distributed network (along with lower layers like UDP or TCP with supported features like atomicity and message ordering. Some distributed software platforms like ISIS^2, JGroups (Also built with ISIS), Apache Zookeeper, Ensemble, AKKA and others. also provide other necessary features like group membership management and fault tolerance support. The DCBlocks library is built using one such open source software platform called Akka

Java (Akka, 2015). It is freely available, easy to use and popular among DC application developers.

The Akka Java toolkit uses actor model where in each computing entity is an actor and the actors communicate with each other by exchanging messages asynchronously. The actors can reside in the same system or in remote systems and they interact with each other in the same way irrespective of their exact physical location. Akka also provides group (called clusters) membership and member life cycle (Joining, Up, Down, Leaving, Unreachable) management services. It has a well-defined supervision hierarchy and failure handling strategy for some of the failures like crash failures, software exceptions etc. The failure handling strategy can be easily extended to other user-defined failure types as well. The DCBlocks is built using all of these features. The design of DCBlocks is kept as generic as possible so that it may require minimal changes even if the underlying DS platform is changed in the future (e.g., change to ISIS^2, etc.)

4.3 DCBlocks Design

This section provides design details of the five building blocks. Each building block has one or more public interfaces. Figure 4.2 shows the class diagram containing all the public interfaces, corresponding implementation classes and the interactions between them. Since Akka Java uses actors and message passing, the internal implementation of these interfaces is done using actors and messages.

4.3.1 Group Management

It might be necessary for decentralized power applications to form groups for a well defined purpose even if for a short duration of time. The processes might need to join or leave

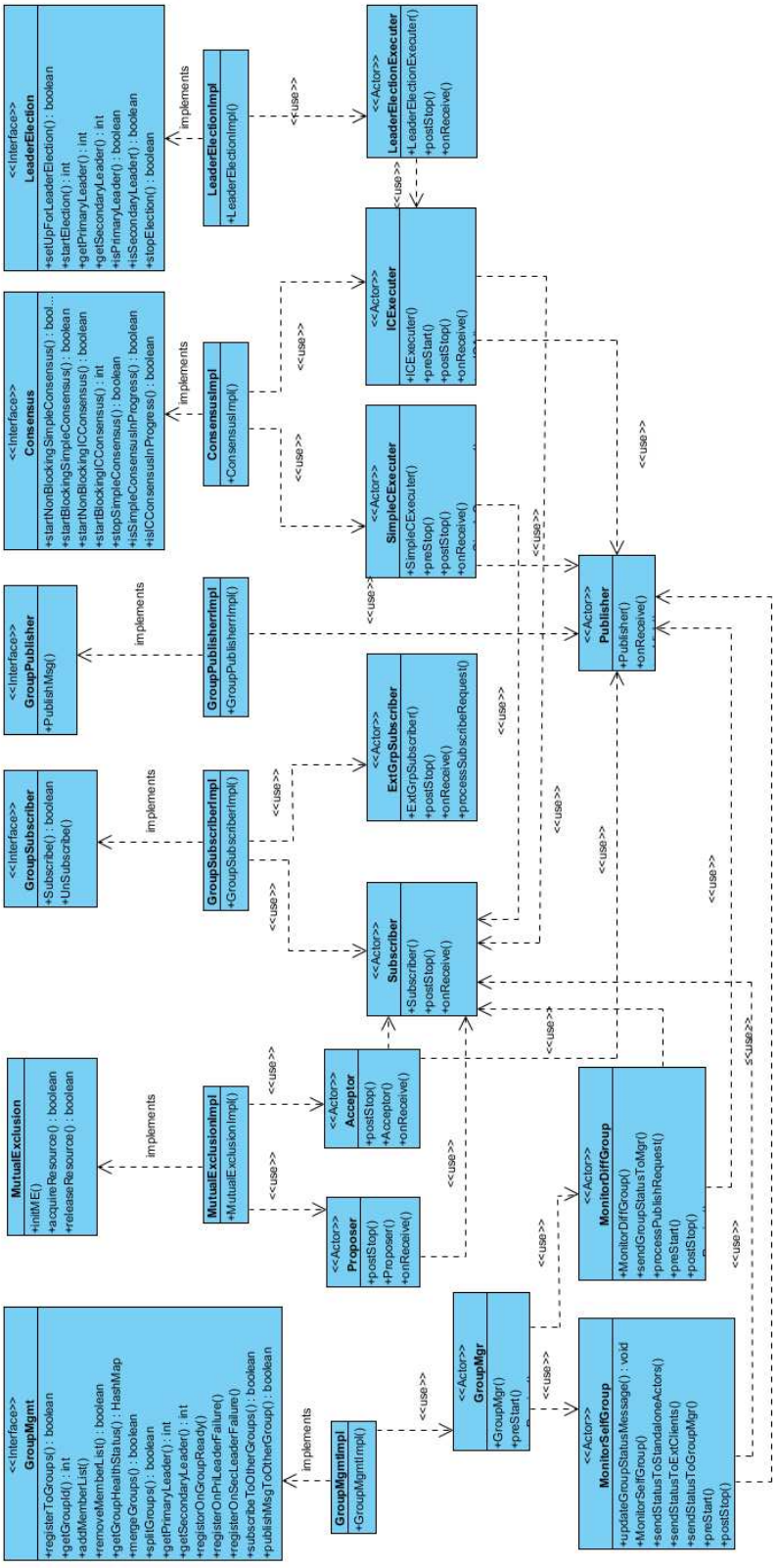


Figure 4.2: DCBlocks Class Diagram.

the group whenever required. Also, the application may need to dynamically reorganize groups whenever required. For example, consider an application that is monitoring and controlling the voltage dynamics of the power system in small distributed groups. If a voltage stability issue is observed within a particular group and if it cannot be resolved within the same group, then it becomes necessary to reorganize the group (perhaps merge two or more groups) to resolve the issue. After the problem is resolved, the merged group can again be split back into original groups. Group failure monitoring and notification is another feature that is necessary. Hence, the design of group management block is made keeping all these factors in mind.

The `GroupMgmt` interface (see Figure 4.2) provides group management services to the application logic. The `GroupMgmt` interface and corresponding methods is listed in appendix Section A.1. The interface is implemented by `GroupMgmtImpl` class. It uses three actor classes - `GroupMgr`, `MonitorSelfGroup` and `MonitorDiffGroup` to perform the group management activities. Below is short description of each class.

- `GroupMgmtImpl` - This class implements all the public interface methods of `GroupMgmt` interface. The interface methods can be broadly classified into two types: operation requests (requests that need some action) and status requests. The class handles operation requests like add member/s to specified group, remove member/s from specified groups, merge two or more groups, split groups by sending `OperationInfo` message to the `GroupMgr` class to perform the actual action. The operation related information like the operation type (e.g. `ADD_SINGLE_MEMBER`, `REMOVE_SINGLE_MEMBER` etc), group ID, node ID etc are filled into `OperationInfo` message before sending. It waits for result from the `GroupMgr` and updates the same to the caller. For the status requests such as: get list of members of a group, get health status of a group, the

GroupMgmtImpl class just returns the local latest information to the caller. GroupMgr, MonitorSelfGroup and MonitorDiffGroup together are responsible for monitoring the status of all the processes in the group. GroupMgmtImpl class is immediately notified, whenever any status change is detected. GroupMgmtImpl in turn updates its local status lists as per latest status information.

- GroupMgr - This class is the core class of the group management service. It is responsible for handling all the operations requests from GroupMgmtImpl class like add, remove, merge, split, get health status, register with external group to send/receive messages etc. It also responsible for collecting the status of all the processes and updating the GroupMgmtImpl class for bookkeeping. It creates two child actor classes, MonitorOwnGroup and MonitorDiffGroup to help with the activities.

- MonitorSelfGroup - The actor class monitors the status of the members of its own group and forwards this member status to parent actor GroupMgr. The class has to register with Akka Java to receive notification events whenever there is any change in group activity. For example, Akka Java sends memberUp event whenever any node joins the group, memberRemoved event whenever any node leaves the group, memberUnreachable event whenever any node is not reachable even after specified timeout and memberTerminated event whenever a node crashes. GroupMgr receives these notification events and updates the status list for each member as healthy or faulty and forwards to GroupMgr. It also sends the same status list (member list and corresponding status) to external groups. Only dedicated processes (seed processes) in each group send the status message to external groups.

- MonitorDiffGroup - The actor registers with external groups to get the group status

(group member list and corresponding status) from external groups. It then forwards the status to `GroupMgr` actor for bookkeeping.

Some of the main group management activities provided by the `GroupMgmt` interface are:

i. Register to Group Management Service - The application logic can call `registerToGroups` method to register for group management service. The method internally initializes the `ActorSystem` of Akka Java and starts the `GroupMgr` actor. The `GroupMgr` actor is responsible for handling all the operation requests from the `GroupMgmt` interface implementation class and to monitor status of all processes in the system.

ii. Add members to a group - The `addMember` and `addMemberList` interface methods can be used by a process node to join a particular group. The sequence diagram showing add member request operation is illustrated in Figure 4.3. An `OperationInfo` message with operation type parameter set as "`ADD_SINGLE_MEMBER`" or "`ADD_MEMBER_LIST`" is sent to `GroupMgr` actor. When the `GroupMgr` actor receives this message, calls the `join` method of Akka Java to join the specified group. After the node has joined the group, it is added to the member list maintained for group status monitoring.

iii. Remove members from the group - The `removeMember` and `removeMemberList` interface methods can be used by a process node to leave a particular group. An `OperationInfo` message with operation type parameter set to "`REMOVE_SINGLE_MEMBER`" or "`REMOVE_MEMBER_LIST`" is sent to `GroupMgr` actor. The `GroupMgr` actor calls the `leave` method of Akka Java to remove each node from the specified group. After the nodes have been removed from the group, they are also deleted from the member list that is maintained for group status monitoring.

iv. Merge two or more groups together - The `mergeGroups` interface method can be used to merge two or more groups together. The input parameters of this method are new group ID and list of group IDs that have to be merged to this new group. The merge operation involves the following steps.

   (a) `GroupMgmtImpl` sends the `OperationInfo` message containing information like operation type as "`MERGE_GROUPS`", list of group ids to be merged to new group and new group ID to `GroupMgr`.

   (b) `GroupMgr` calls the leave method of Akka Java to remove all the group members from each group in the merge group list. `GroupMgr` sends the status back to `GroupMgmtImpl` class.

   (c) `GroupMgmtImpl` class deletes all the information related to these groups and informs that these groups are dissolved.

   (d) `MonitorSelfGroup` informs other groups that these groups are dissolved.

   (e) After that, `GroupMgmtImpl` class sends the `OperationInfo` message to `GroupMgr` with parameters such as operation type set to "`ADD_MEMBER_LIST`", node ID list containing list of all the node IDs that have to be added to new group ID.

   (f) GroupMgr actor class calls the `join` method of Akka Java for all nodes in the node ID list. It then sends the status back to `GroupMgmtImpl` class.

   (g) `GroupMgmtImpl` class update the local member list for this new group.

   (h) If the application has registered for a callback method, then it is called with the status of the merge operation.

v. Split a group into two groups - The `splitGroups` interface method can be used to split a single group into two groups. The input parameters of this method are: group

id which needs to be split into two groups (i.e., old group ID), new group ID and list of nodes that have to be moved to this new group. The split operation involves the following steps.

(a) `GroupMgmtImpl` sends the `OperationInfo` message to `GroupMgr` class with information such as operation type set to "`SPLIT_GROUPS`", old group ID, new group ID and list of node IDs to be added to new group(s).

(b) `GroupMgr` calls the `leave` method of Akka Java to remove all the nodes in the node id list from the old group. `GroupMgr` sends the status back to `GroupMgmtImpl` class.

(c) `GroupMgmtImpl` class deletes all the bookkeeping information related to these nodes from the old group.

(d) After that, `GroupMgmtImpl` class sends the `OperationInfo` message to `GroupMgr` with information such as operation type set to "`ADD_MEMBER_LIST`", new group ID, and list of all the node IDs that have to be added to new group ID.

(e) GroupMgr actor class calls the `join` method of Akka Java for all nodes in the node ID list. Sends the status back to `GroupMgmtImpl` class.

(f) `GroupMgmtImpl` class updates the local member list for this new group.

(g) If the application has registered for a callback method, then it is called with the status of the split operation.

vi. Get group health status - The `getGroupHealthStatus` provides health status of all the group members for a specific group. The health status can be healthy or faulty. More details about failure monitoring and how the health status is updated is provided in Section 4.3.1.1.

vii. Register for operation status - The `GroupMgmt` interface provides methods for the application logic to register for callback upon completion of some critical (which are complex and time consuming) group activities or on any node failure. The registration methods are:

   (a) Group ready status - If the application is dependent on starting its normal operation after a set of nodes have joined the group, it can register to the `registorOnGroupReady` method with its callback method as input parameter. Once the group has reached the specified group size (for example, 10 nodes), the application call back method is called to inform that the group is ready.

   (b) Primary leader failure - The application can register to get notified when the primary leader of a group has failed. Whenever a group member is detected unreachable/terminated by `MonitorSelfGroup`, the corresponding member status is set as faulty and a status message is sent to `GroupMgr` which in turns sends it to `GroupMgmtImpl` class. `GroupMgmtImpl` class checks if the faulty node is the primary leader of the group. If yes, then the registered call back method is called to inform about the primary leader failure.

   (c) Secondary leader failure - The application can register to get notified when the secondary leader of a group has failed. Whenever a group member is detected unreachable/terminated by `MonitorSelfGroup`, the corresponding member status is set as faulty and a status message is sent to `GroupMgr` which in turns sends it to `GroupMgmtImpl` class. `GroupMgmtImpl` class checks if the faulty node is the secondary leader of the group. If yes, then the registered call back method is called to inform about the secondary leader failure.

(d) Merge operation completion status - The application can register to get notified when the merge operation is completed. After the merge operation is completed, the registered call back method is called to inform about the completion status with details like old group ID, new group ID, new member list, etc.

(e) Split operation completion status - The application can register to get notified when the split operation is completed. After the split operation is completed, the registered call back method is called to inform about the completion status with old group ID, new group ID, member list, etc.

(f) Member failure status - The application can register to get notified on any group member failure. Whenever a group member is detected faulty, the registered call back method is called.

4.3.1.1 Failure Detection and Tolerance

The common method for checking if a remote node is active/reachable or not is by sending heartbeat messages periodically to all remote nodes and waiting for replies from each of them. If reply is not received from any of them within the specified time interval, corresponding node is considered as unreachable/faulty. Our group management service uses Akka Java for failure monitoring. Akka Java internally uses failure detectors to detect failures (through heart beat mechanism) in the group. Figure 4.4 shows the sequence of operations involved in detecting a leader crash failure and reporting back to the application. In a group/cluster each node is monitored by a few (default maximum 5) other nodes, and when any of these detects the node as unreachable that information will spread to the rest of the cluster through the gossip protocol. In other words, only one node needs to mark a node unreachable to have the rest of the nodes in the group/cluster mark that node as un-

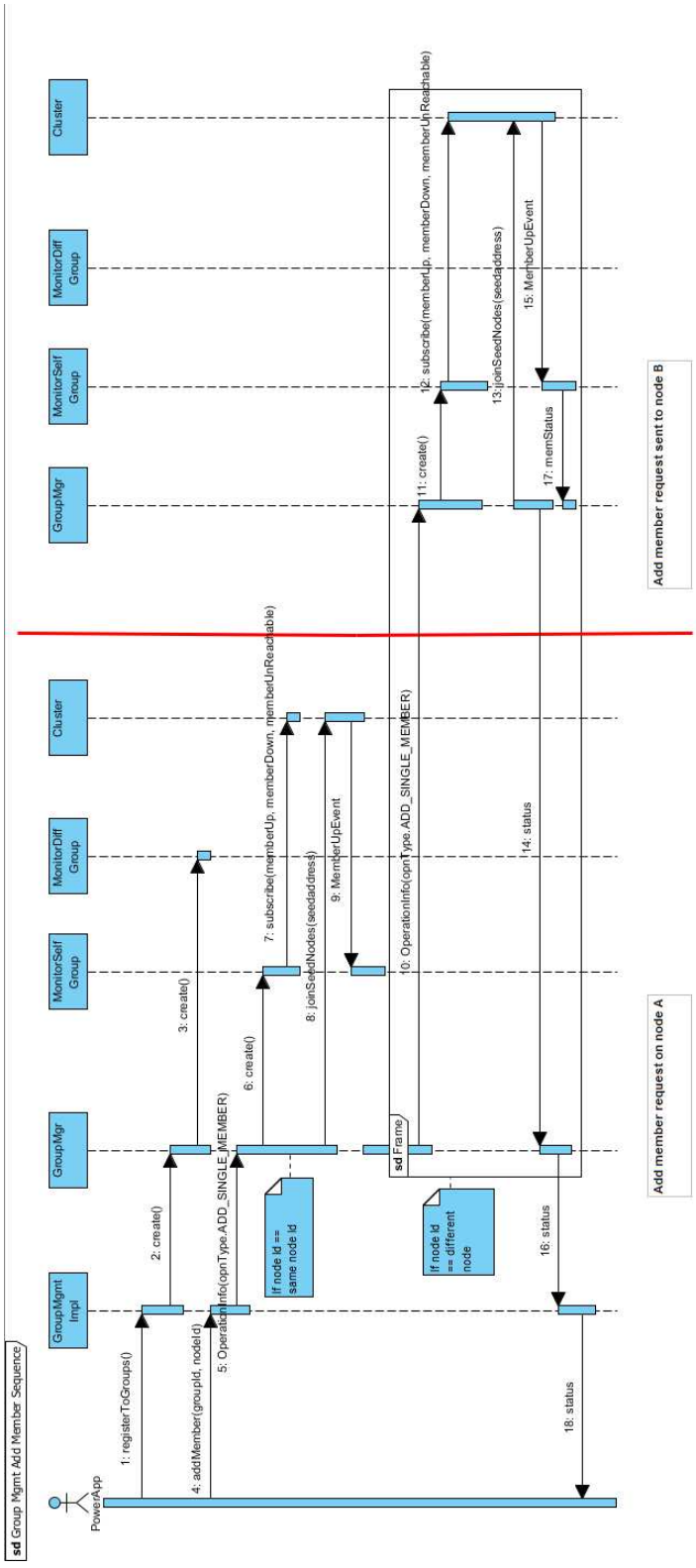


Figure 4.3: Sequence Diagram for Add Member Request

reachable. The `MonitorSelfClass` registers with Akka Java to receive `UnreachableMember` and `memberRemoved` events. So whenever it receives these events, it sets the corresponding member status to faulty and informs `GroupMgr` which in turn informs `GroupMgmtImpl`.

Similarly, the failure detector will also detect if the node becomes reachable again. When all nodes that monitored the unreachable node detects it as reachable, the information is spread to rest of the nodes in the group through gossip dissemination and finally all nodes in the group will consider it as reachable. The corresponding notification event from Akka Java is `ReachableMember`. The heartbeat timeout value is adjusted by Akka Java dynamically based on the current network conditions.

4.3.2 Ordered Multicast

This block provides mechanism to multicast messages to all members in the same group or to external group members. The `GroupPublisher` interface provides methods to multicast/publish messages of a particular topic type to a group. `GroupSubscriber` interface provides methods to subscribe/unsubscribe to receive multicast messages of a particular topic type from a group. The underlying Akka Java framework handles the actual publish-subscribe communication ensuring all the nodes receive FIFO ordered messages and atomically. `GroupSubscriber` interface internally uses `Subscriber` actor class to handle subscription requests for messages from own group and `ExtGroupSubscriber` actor class to handle subscription requests for messages from external group. The `GroupSubscriber` and `GroupPublisher` interfaces and corresponding methods is listed in appendix Section A.4 and A.5.

`GroupSubscriber` interface provides interface methods to handle the following activities:

i. Subscribe to receive messages of specific topic - If application wants to receive mes-

sages of specified topic type from a specific group, it needs to register by calling `Subscribe` method with the interested topic, group ID and call back function to get the received messages as input parameter. If the group ID is owns group, the operation request message is sent to `Subscriber` which in turn calls `subscribe` method of `DistributedPubSubMediator` class of Akka Java framework. The sequence diagram showing the subscription sequence is given in Figure 4.5. If the group ID is an external group ID, the operation request message is sent to `ExtGroupSubscriber` which in turn calls `registerSubscriber` method of `ClusterReceptionistExtension` class of Akka Java framework. Whenever the message with same topic is received, it is delivered to the application logic using the callback function.

ii. Unsubscribe to stop receiving messages of specific topic type - If an application wants to stop receiving messages of specified topic type from a specific group, it needs to unregister by calling `UnSubscribe` method with the topic and group ID as input parameter. If the group ID is own group, the operation request message is sent to `Subscriber` which in turn stops the `Subscriber` actor class. If the group ID is external group, the operation request message is sent to `ExtGroupSubscriber` which in turn stops the `ExtGroupSubscriber` class.

`GroupPublisher` interface provides `Publishmsg` method to send messages to same or external group. If an application wants to send a message to all members in the same group, it needs to use `Publishmsg` of `GroupPublisher` interface with the desired topic. Figure 4.6 shows the sequence of operations involved in publishing a message within the group.

4.3.2.1 Failure Detection and Tolerance

The default message delivery pattern used in Akka Java is at-most-once, FIFO message ordered delivery. When a message is sent by a process, it is delivered at most once. This means if a message is lost mid way in the network between the sender and receiver, it is not resent by the sender. At-most-once delivery helps in faster communication since there is no requirement for re-sending of lost messages. There are no duplicate message delivery. The application logic has to use timeout mechanism to detect lost messages and request the sender to resend message.

Akka Java uses FIFO message ordering i,e., messages are delivered in FIFO message order per sender. Akka Java does not support total-ordering semantics as explained in Section 2.4.3. This is a major drawback. FIFO message ordering will not be sufficient in applications where all the processes need to receive messages in the same order in order to perform same consistent action. The future work of DCBlocks should overcome this drawback and add Total-Ordering or ABCAST message delivery mechanism.

4.3.3 Consensus or Agreement

Based on decentralized power application survey discussed in Chapter 3, it seemed that Simple Consensus Algorithm and Interactive Consistency Algorithm would be most applicable for power scenarios. A brief description of these algorithms were presented in Section 2.4.1.

The interface for Consensus block is `Consensus`. The `Consensus` interface and corresponding methods is listed in appendix Section A.6. The implementation class for this interface is `ConsensusImpl` class. Simple consensus is implemented by `SimpleCExecutor` actor class and ICA is implemented by ICExecutor actor class. The `Consensus` provides

methods to run Simple Consensus and ICA algorithms both as blocking as well as non blocking methods. In blocking consensus, the application is blocked and waits for completion of the consensus algorithm. Whereas in non blocking consensus, the application is not blocked and continues with other unrelated operations. The application can register a callback function to get notified when the consensus algorithm gets completed. After completion, call back method is called with result (consensus decision) of the algorithm.

i. Simple Consensus - The `Consensus` provides `startNonBlockingSimpleConsensus` and `startBlockingSimpleConsensus` methods to run Simple Consensus algorithm. The methods pack the input parameters like local proposal value, decision function into a message and send to `SimpleCExecutor` to start the consensus algorithm. For non blocking method, it saves the call back function to be called later when the algorithm is completed. Figure 4.7 shows the sequence of operations for simple consensus algorithm.

   `SimpleCExecutor` gets the total number of nodes in the group and number of failed nodes (f) in the group from the group management service and checks if $f < (n-1)/2$, where n is the total number of processes in the group. If yes, it continues with the algorithm. In the first round, all the processes broadcast their local values to all other members in the group. In the subsequent rounds (upto $f+1$ rounds), each node transmits only those values which it has not received in the previous rounds. After $f+1$ rounds, the algorithm stops and calls the application specified decision function to compute a single common value from the list of values. The application specified decision function can be for example, to find the max, min, average or any cyber-physical power related function. The agreed up consensus value is returned to

`ConsensusImpl` class which is in turn returned to the caller.

ii. Interactive Consistency Algorithm - The `Consensus` interface provides methods `startNonBlockingICConsensus` and `startBlockingICConsensus` to run ICA. These methods pack the input parameters such as local proposal value into a message and send to `ICExecutor` to start the consensus algorithm. For the non blocking method, it saves the call back function to be called later after the algorithm is completed.

`ICExecutor` gets the total number of nodes in the group and number of failed nodes (f) in the group from the group management service and checks if number of failed nodes $f < (n-1)/2$, where n is the total number of processes in the group. If yes, it continues with the algorithm else it does not proceed. In the first round, all the processes broadcast their local values to all the other members in the group. In the subsequent rounds (upto $f+1$ rounds), each node prepares a vector (one slot for each process) to broadcast in that round of iteration. Each vector slot is filled based on the majority of values received for that slot in the previous round. If no values are received for that slot, it is filled with "-1" to indicate error or missing value. By the end of $f+1$ rounds, each process has the same decision vector. The decision vector is returned to `ConsensusImpl` class which is in turn returned to the caller.

The sequence of operations for ICA is shown as part of Leader Election sequence diagram in Figure 4.8. For this design, only integer types are considered as proposal values. But it can be extended to support generic data types in the future. The `Consensus` interface also provides methods to stop the execution of the algorithms and to check if algorithm is still in progress or not. For simple consensus, any application defined function (for example, minimum, maximum or any mathematical function on the power data measurements) can

be used to decide on a common value.

4.3.3.1 Failure Detection and Tolerance

The algorithms can tolerate failures upto $f = (n-1)/2$ faulty processes. The algorithm is started/continued only if $f < (n-1)/2$. This check is made at the beginning of each round. Crash failures are detected by group management block and notified to this block. Omission errors are detected using timeout mechanism. The timeout value is set based on the number of nodes in a group. If a message is not received from a particular node within the timeout, an error indicating value (-1) is filled in the corresponding slot in the local "rx" vector maintained for each process. In case of ICA, the vector for the next round is prepared by taking the majority of received values for each vector slot in the previous slot. If no message is received from any of the processes for that slot in the previous round, then that slot is filled with "-1" and broadcast in the next round of the algorithm. Final decision function is applied to only non error values.

4.3.4 Leader Election

This block provides methods to elect one process among them to make decisions or calculations for the group. Each participating node need to propose a vote to start the election. The leader election is executed through multiple rounds by collecting votes from the all nodes in each round and finally a leader for the group is elected. The decision criteria to elect the leader can be based on either the node having the highest processor ID or some application defined decision criteria. Secondary leader can also be elected which can be a fall back leader in case of failure of primary leader failure.

The available leader election algorithms in the literature like bully algorithm (G. Coulouris

and Blair., 2011), ring based algorithm (G. Coulouris and Blair., 2011) are based on electing the node having highest processor ID. But DCBlocks has to be designed to accept any generic application defined decision criteria (can be cyber-physical function too). For example, if the leader needs to be elected based on the node having the least computational (work) load in the group, then the decision function will be to compute the node having the lowest workload. For such cases, we need to get consistent view of all the proposed votes across all nodes in the group, and same decision function has to be applied by every member of the group. For this reason, Interactive Consistency Algorithm (ICA) seemed to be the best option and is internally used to get consistent list of votes. After the votes are collected, an application defined decision function is applied on the vote list to select the primary leader. Similarly, a secondary leader is also selected to be the fall back leader. The interface also provides methods to stop/abort a running election in case of failure and methods to get the identifier of current primary and secondary leader.

`LeaderElection` interface provides methods for leader election. The `LeaderElection` interface and corresponding methods is listed in appendix Section A.7. The implementation class for this interface is `LeaderElectionImpl` class. The leader election algorithm is executed by `LeaderElectionExecutor` actor class. Figure 4.8 shows the sequence of operations for Leader Election. `LeaderElection` interface provides methods for the following activities:

i. Set up for Leader Elections - During initialization, the application has to set up for future leader elections by registering its vote and decision function by calling `setUpForLeaderElection` method. After that, any node can initiate the election by calling `startElection` method.

ii. Start Leader Election - The `startElection` method can be used to start an election. It is done in following steps.

   (a) If there are no on-going leader elections currently, `LeaderElectionImpl` sends a message to the `LeaderElectionExecutor` to start the election.

   (b) `LeaderElectionImpl` in turn sends a `LEMsg` message with action type parameter set to `LE_START` to all nodes in the group to start the election process.

   (c) Once, the election start message is received by any node, `LeaderElectionExecutor` internally creates an instance of `ICExecutor` to run the ICA algorithm.

   (d) In the first round of the ICA, each node publishes its vote to the group.

   (e) In further rounds, all the votes are collected and finally (after $f + 1$ rounds) the decision vector is returned to `ICExecutor`.

   (f) `LeaderElectionExecutor` applies the registered decision function on the list of proposed votes and decides the leader of the group.

   (g) Similarly, secondary leader is elected (for example, the node with second highest votes). The node IDs of the primary and secondary leaders is informed to the caller.

   (h) The primary and secondary leader IDs are also sent to the group management block for bookkeeping and for failure monitoring.

iii. Stop Leader Election - The `stopElection` method can be used to stop a running election. If there is an on-going election, `LeaderElectionImpl` sends `LEMsg` message with action type parameter set to `LE_STOP` to `LeaderElectionExecutor` which in turn stops the `ICExecutor`.

iv. Get Primary and Secondary Leader IDs - The `getPrimaryLeader` and `getSecondary-Leader` can be used to get the primary and secondary leader IDs.

4.3.4.1 Failure Detection and Tolerance

The algorithms can tolerate failures upto $f = (n-1)/2$ faulty processes where n is total number of processes in the group. Since the design internally uses ICA to perform the election, this block also supports all the failure handling features provided by the `Consensus` block. The primary leader ID and secondary leader ID is sent to the group management block at the end of leader election. The group management block monitors the status of these nodes continuously. If any of the leaders are detected unreachable, it is immediately reported to the application. If a primary leader has failed, the application can switch to secondary leader. If secondary leader also fails, then the application can initiate a new leader election.

4.3.5 Distributed Mutual Exclusion

This block provides methods for exclusive access of a shared resource. Several algorithms as described in Section 2.4.5 were considered for the implementation. Algorithms like Ricart and Agrawala's algorithm, Maekawa's Voting algorithm are useful in understanding the logic and concept of distributed mutual exclusion. But, they are rarely used in the industry. Maekawa's Voting algorithm is deadlock prone and additional measures needs to be taken to prevent deadlock. Most of the distributed mutual exclusion algorithms for applications such as distributed file systems, distributed lock service etc. use some variation of Paxos algorithm. Hence, our design of this block is also based on Paxos. Hupfeld et al. (2008) proposes lease based exclusion algorithm where the persistence storage feature of original

Paxos is replaced with a definite finite lease period is introduced. When the lease period is over, any other process can bid to access the resource. When a process crashes when using a resource, the crashed process is easily detected and lease period is considered invalid and a new request is instantiated. When a crashed process restarts, the leader sends the highest instance number for all the resources. The algorithm is applicable for tightly synchronous systems with very small clock skews. This assumption is valid for power systems and hence can be applied to power system applications.

`MutualExclusion` interface provides methods for gaining and releasing access of shared resources. The `MutualExclusion` interface and corresponding methods is listed in appendix Section A.8. The implementation class for this interface is `MutualExclusionImpl` class. The mutual exclusion algorithm is executed by `Proposer` and `Acceptor` actor classes. Here, all the processes in the group can be Proposers and Acceptors.

`MutualExclusion` interface provides methods for the following activities:

i. Initialize Mutual Exclusion block - The `initME` method needs to be used to initialize the setup for distributed mutual exclusion. Initially, the `Proposer` and `Acceptor` actor classes are created.

ii. Request to access a shared resource - The `acquireResource` method can be used to request for shared resource. The input parameters such as resource ID and time duration for access are packed into `ResourceRequestMsg` and sent to `Proposer` actor. The following steps are executed to get access of the resource.

    The following steps are activities of `Proposer` for Prepare phase.

    (a) When Proposer receives `ResourceRequestMsg`, it starts the first round to initiate consensus.

(b) It checks if there is any process that is already using the resource. If yes, it waits till lease period is completed.

(c) If no, it generates a unique ballot number and prepares a `PrepareRequestMsg` with the ballot number and highest instance number seen so far for that resource. It then publishes the message to all the Acceptor processes in the group.

(d) It waits for `PrepareReplyMsg` message from all the Acceptor processes in the group.

(e) When `PrepareReplyMsg` message is received, it does the following based on the contents of the message.

   i. If message type is `Outdated`, then get latest instance number and restart from step 1.

   ii. If message type is `Ack` and if number of `Ack` messages $Ack > (H/2 + 1)$ where H is total number of Acceptor processes in the group, then it moves to Accept phase.

   iii. If message type is `Nack`, it updates the instance value with the highest value contained in all the reply messages. It then moves to Accept phase.

The following steps are activities of `Acceptor` for Prepare phase.

(a) When `Acceptor` receives `PrepareRequestMsg`, it checks the instance number i of the incoming message with local instance number j for the resource.

   i. If $i < j$, then `Proposer` missed some messages. Send `PrepareReplyMsg` with message type set as `Outdated`.

   ii. If $i > j$, then `Acceptor` has missed some prepare messages, so update local instance number. Send `PrepareReplyMsg` with message type set as `Ack`.

iii. If $i == j$, then `PrepareReplyMsg` with message type set as `Ack`.

iv. If ballot number is lesser, then its own ballot number in the `PrepareReplyMsg` with message type set as `Ack`.

The following steps are `Proposer` activities for Accept phase.

(a) `Proposer` publishes the `AcceptRequestMsg` message with ballot number and instance number parameters set to all the Acceptor processes in the group.

(b) It waits for `AcceptRequestMsg` message from all the Acceptor processes in the group.

(c) When `AcceptRequestMsg` message is received, it does the following based on the contents of the message.

i. If message type is `Ack` and if number of `Ack` messages $Ack > (H/2 + 1)$ where H is total number of Acceptor processes in the group, then it sends to `LearnMsg` phase.

ii. If message type is `Nack`, it waits for a time period and start with step 1 again.

The following steps are `Acceptor` activities for Accept phase.

(a) When `Acceptor` receives `AcceptRequestMsg`, it checks the instance number i of the incoming message with local instance number j for the resource.

i. If $i < j$, then `Proposer` missed some messages. Send `AcceptReplyMsg` with message type set as `Nack`.

ii. If $i > j$, then `Acceptor` has missed some prepare messages, so update local instance number. Send `PrepareReplyMsg` with message type set as `Ack`.

iii. If $i == j$, then `PrepareReplyMsg` with message type set as `Ack`.

(b) When `Acceptor` receives `LearnMsg`, it records the ballot number, instance number and lease duration for the resource.

iii. Release the access of a shared resource - The `releaseResource` method can be used to release the resource. The input parameters such as resource ID are packed into `releaseMsg` and sent to `Proposer` actor. The `Proposer` actor publishes it to all `Acceptor` in the group. The `Acceptor` records that the resource is released.

iv. Get Resource ID list - The `getResourceId` method can be used to get list of resources.

4.3.5.1 Failure Detection and Tolerance

The algorithms can tolerate failures upto $f = (H - 1)/2$ faulty processes where H is total number of Acceptor processes in the group. If a process crashes while using the resource, `Acceptor` of all processes in the group get notified through Group Management Block. It can update its local copies as invalid instance number and start a new prepare request.

4.4 Application Logic using DCBlocks

Distributed power applications can use the above blocks as per their application logic. Few use cases of power applications using DCBlocks is described in Chapter 5.

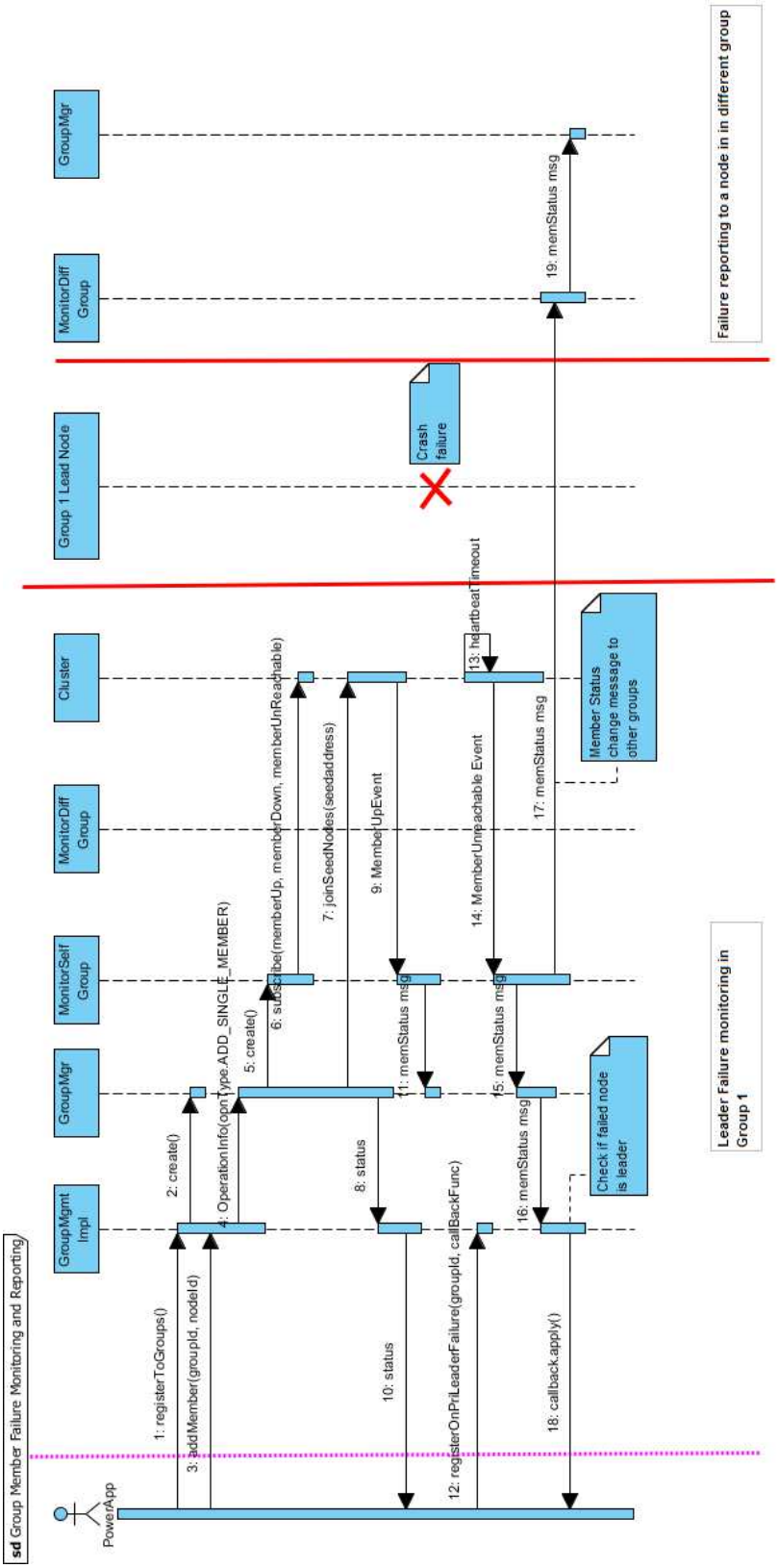


Figure 4.4: Sequence Diagram for Leader Failure Monitoring and Reporting

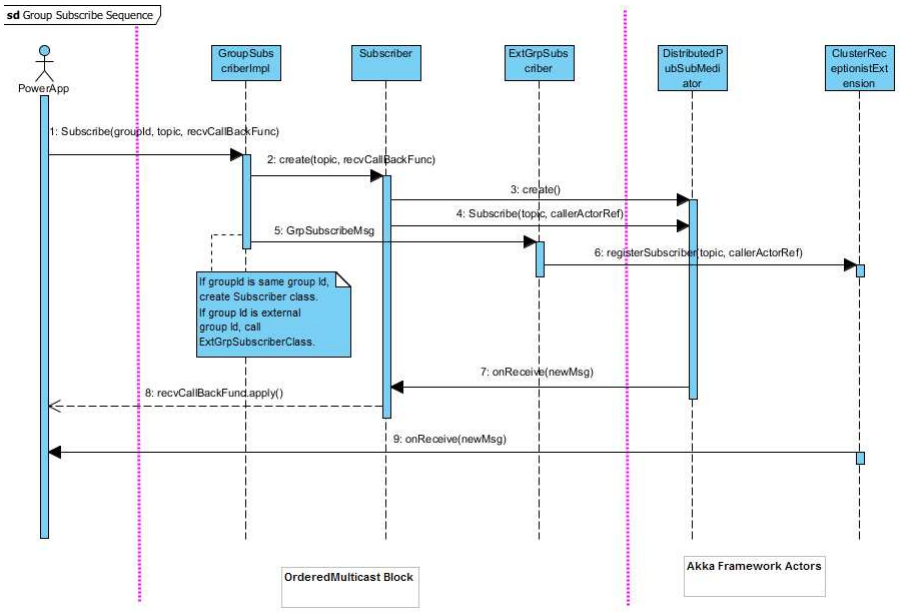


Figure 4.5: Sequence Diagram for subscribing to a topic

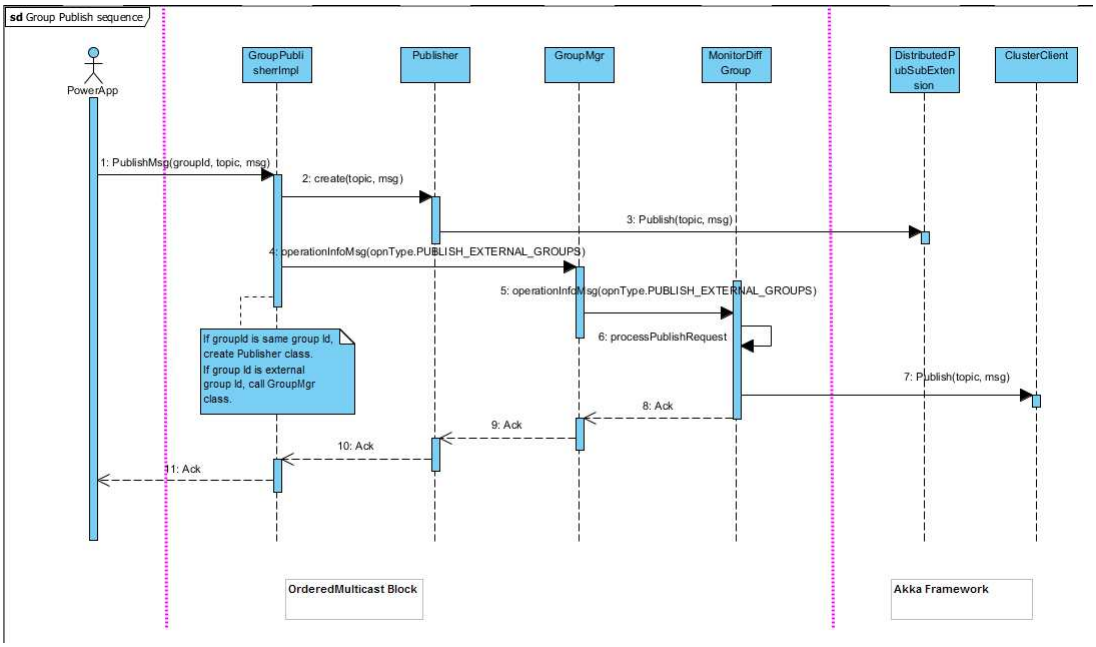


Figure 4.6: Sequence Diagram for publishing message with a topic

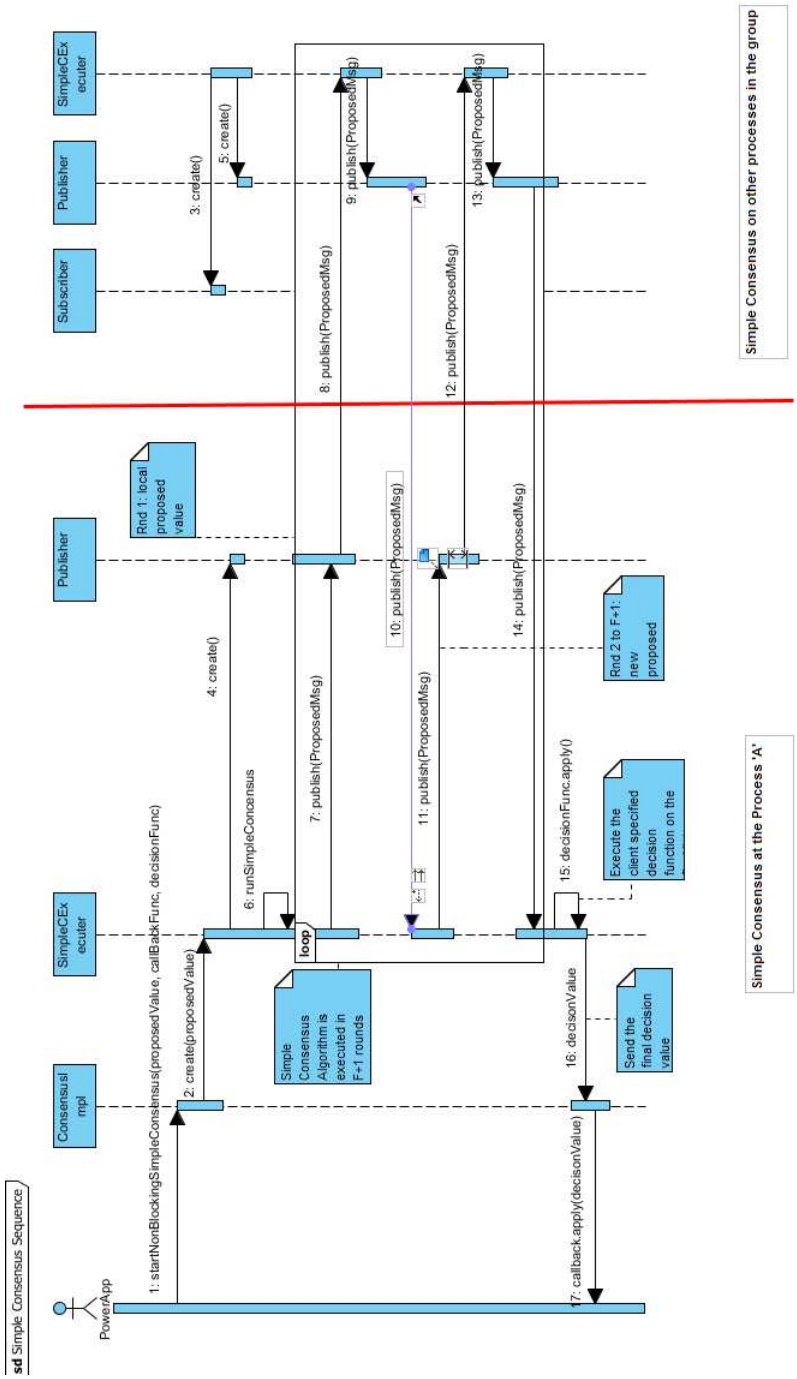


Figure 4.7: Sequence Diagram for Simple Consensus Algorithm

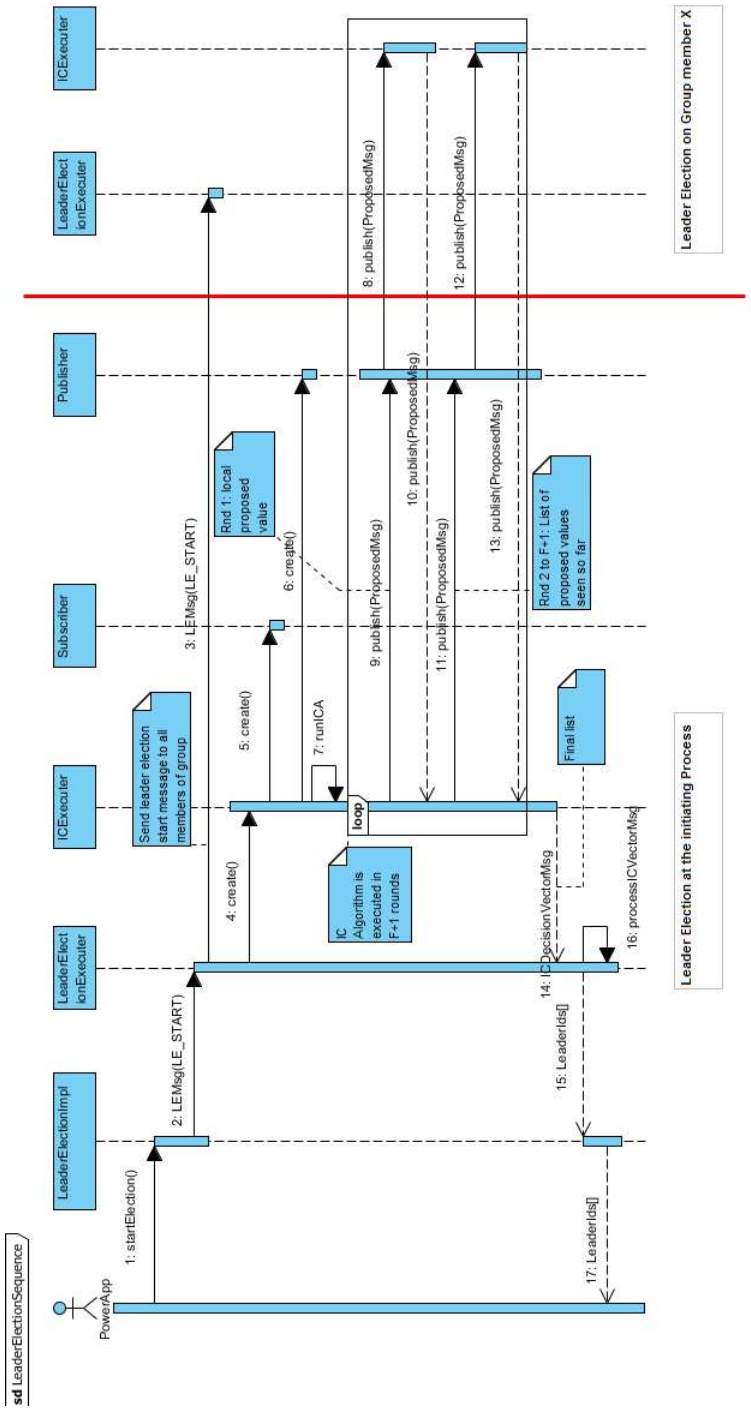


Figure 4.8: Sequence Diagram for Leader Election

CHAPTER 5

DECENTRALIZED POWER APPLICATION USE CASES USING DCBLOCKS

5.1 Introduction

Several examples of decentralized power applications using some of the DC building blocks described in Chapter 4 is given in this chapter. This will helps the reader to visualize the applicability of DCBlocks to present and emerging decentralized power applications. Each use case is using one or more of the building blocks described in previous chapter.

5.2 Decentralized Linear State Estimation

Decentralized Linear State Estimation (DLSE) is good candidate use case of decentralized power application developed using DCBlocks. State Estimation as described in Section 3.3 is a very critical application and its computational efficiency and robustness is critical for power system monitoring. The power system will have large set of system variables in the future, and the traditional centralized state estimation will take more time to compute the system status or sometimes may not be able to converge. Alternate approach is to divide the whole power system into smaller sub groups and perform Decentralized State Estimation (DSE) at group level as shown in Figure 5.1 to reduce the complexity of power system monitoring. The PMUs installed at the substations provide accurate phasor meas-

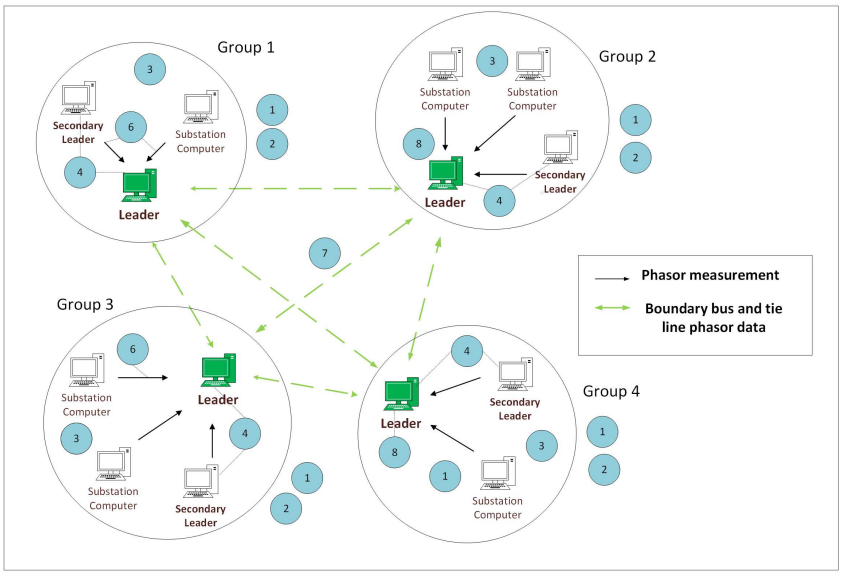


Figure 5.1: Distributed groups running DLSE algorithm

urements which can be used to compute DSE. All the substations in the group can send their local PMU data to a leader of the group. The leader can then run the DSE algorithm for the group. The output of DSE can be used by other dependent power applications like Decentralized Voltage Stability (DVS) described in Section 5.3.

Use case for Decentralized Linear State Estimation using computational blocks of DCBlocks is described below. Liu et al. (2016) (under review) developed in collaboration with research students of SGDRIL, WSU contains more details about Decentralized Linear State Estimation power algorithm using DCBlocks. Multiple group of sub stations can be formed using Group Management block of DCBlocks. A leader sub station to perform DLSE can be elected using Leader Election block of DCBlocks.

Table 5.1: Use Case for Decentralized Linear State Estimation

Use Case	**Decentralized Linear State Estimation**
Description	This use case describes Decentralized Linear State Estimation for decentralized groups. All the substations/nodes send local PMU measurements to the leader of the group. The leader runs the DLSE algorithm for the group. It uses DCBlocks for robust implementation of DLSE.
Input	The local bus and line power measurements from each substation in a group. Boundary bus and tie line power measurements from adjacent groups.
Output	State estimation result from the DLSE algorithm.
Actors	Leader substation, follower substations, PMUs.
Assumptions	Each substation has computing device capable of running DLSE software and collecting local PMU data and sending to the leader of the group.

continued ...

... continued

Use Case	Decentralized Linear State Estimation
Steps	i. On system start up, DSE application on each node initializes DCBlocks and checks the group it should belong to. Initial group is designed based on electrical distance, low connectivity between neighboring groups and additional application specific requirements. ii. Each node joins the group using `addMember` method of DCBlocks Group Management Block. iii. Each node starts leader election using `startElection` method of DCBlocks Leader Election Block. Each process broadcasts its local computational ability value to rest of the group. iv. The node with the highest computational ability in the group is elected as the leader. A secondary leader is also selected as back up leader to run the DSE algorithm if the primary leader fails. v. The lead node starts the leader activities (i.e., wait for incoming power measurements from all group members, run DSE monitoring algorithm). vi. Each node starts to send its local PMU measurement (current phasor values of the connected bus and current phasor value of connected line) to its lead node at every 100 milliseconds. vii. The boundary bus voltage phasor value and tie line current phasor value is obtained from the adjacent group leaders. viii. Once all the power measurements are received from the group members, the leader runs the DLSE monitoring algorithm.
Issues	The communication medium provided by DCBlocks and underlying DS platform software should be able to detect failures like crash, omission failures and report to the DLSE application software. DCBlocks should be able to handle coordination between different processes even in the presence of varying computational and network delays.

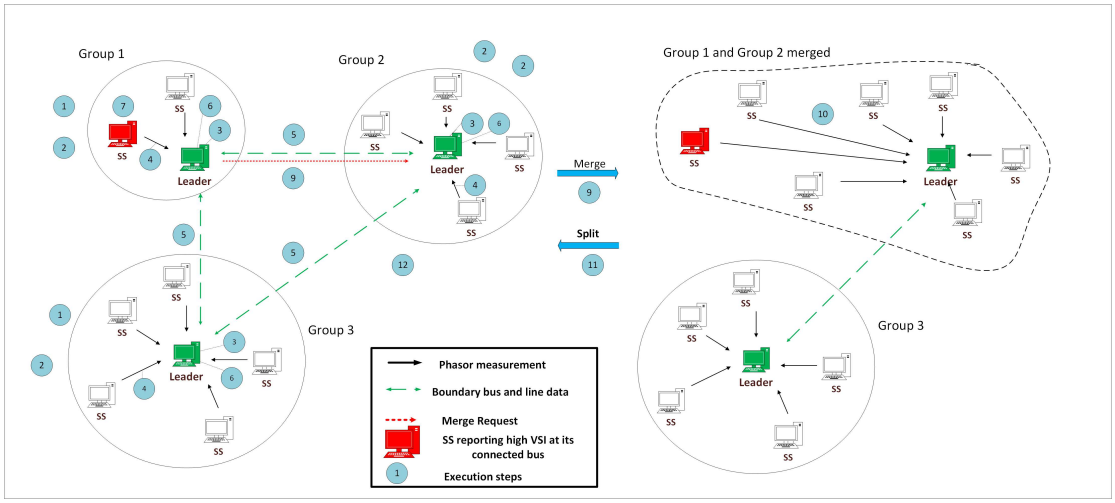


Figure 5.2: Distributed groups running DVS algorithm

5.3 Decentralized Voltage Stability

Voltage Stability problem described in Section 3.2 is inherently local, can be solved locally using available reactive power resources in the neighborhood dynamically. Hence, it becomes a good example for designing robust Decentralized Voltage Stability (DVS) application using DCBlocks. Consider a large power system organized into smaller sub groups of neighbor substations as shown in Figure 5.2. Each group performs the DVS monitoring and control independently with limited interaction with the central control center. The PMUs installed at the substations provide accurate phasor measurements which is used by the DVS monitoring and control algorithms. All the substations in the group send their local PMU data to a leader of the group. The leader runs the DVS monitor and control algorithms with this information. DCBlocks helps in providing reliable communication and efficient coordination among the communicating entities. This helps in making correct decisions locally. Thereby, enabling faster response, instead of sending all the sensor data to central control center and waiting for response. A prototype implementation of DVS using DCBlocks is described in the Chapter 6. Detailed description of the simulation of DVS using

DETERlab (DETERLab, 2015) and corresponding simulation results are also described in the same chapter. Further description of DVS power algorithm and implementation of the algorithm using DCBlocks can be found in our submitted paper Lee et al. (2015) developed in collaboration research students of SGDRIL, WSU.

Table 5.2 describes the use case of DVS application using DCBlocks.

Table 5.2: Use Case for Distributed Voltage Stability

Use Case	**Distributed Voltage Stability (DVS)**
Description	This use case describes DVS application where a large power system is organized into smaller sub groups based on electrical distance, voltage to reactive power sensitivity and reactive power availability. Each group performs the DVS monitoring and control actions independently. If reactive power is insufficient to resolve voltage stability issue within the group, it should be able to dynamically regroup to acquire reactive power from neighboring groups.
Input	The local bus (current, voltage phasor values, shunt capacitance) and line (impedance, capacitance) power measurements from each substation in a group. Boundary bus and tie line power measurements from adjacent groups.
Output	Application of reactive power using shunt capacitors if a voltage stability problem is observed at load buses in the group.
Actors	Leader substation, follower substations, PMUs.
Assumptions	Each substation has computing device capable of running DVS software. Each sub station collects local local PMU data and sends to the leader of the group.

continued ...

. . . continued

Use Case	Distributed Voltage Stability (DVS)
Steps	i. On start-up, each node checks the group it should belong to. Initial groups are designed based on electrical distance, reactive power availability and voltage to reactive power sensitivity parameters. Each node uses `addMember` method of Group Management block of DCBlocks to join the group. ii. Each node in the group takes part in leader election using Leader Election Block of DCBlocks to elect a leader for the group. Each process broadcasts its local computational ability value as its local vote. iii. The node having the highest computational ability is elected as the leader. iv. Each node sends its local power measurements (bus and line data from PMUs) to the leader. v. Lead nodes of adjacent groups exchange boundary bus and tie line phasor measurement data. vi. The leader collects all the data from the group and runs the DVS monitoring algorithm. The DVS monitoring algorithm computes maximum Voltage Stability Index (VSI) for the group vii. If VSI is greater than the threshold, then leader runs the DVS control action algorithm. viii. Reactive power is injected to affected load buses based on reactive power availability and priority determined by DVS control algorithm. ix. If reactive power is insufficient to resolve the problem within the group, then two or more adjacent groups are merged using `mergeGroups` method of DCBlocks group management block. x. Steps iv through viii are performed repeatedly until the VS problem is resolved. xi. The groups split back to original groups using `splitGroups` method of DCBlocks group management block. xii. The groups continue with normal operation after regrouping, repeating steps ii through xi continuously.

continued . . .

. . . continued

Use Case	Distributed Voltage Stability (DVS)
Issues	Dynamic regrouping need to be implemented carefully such that it does not lead to failures like crash failures, network partitions etc. The groups should be able to continue normal operation after re-grouping.

5.4 Decentralized Remedial Action Schemes

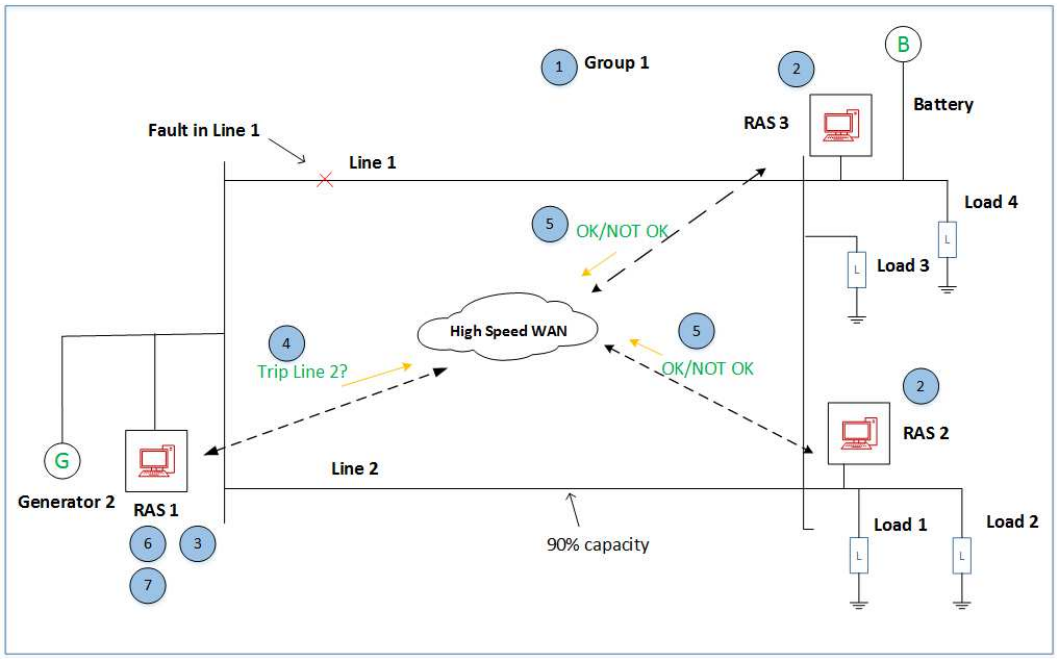


Figure 5.3: Group communication for DRAS

Another example of distributed power application using DCBlocks is Decentralized Remedial Action Schemes (DRAS). As described in Section 3.4, RASs protect the power grid by preventing local instability conditions to result in complete power outage. A dynamic RAS that can assess the system configuration and changing network topology continuously and perform the right type and magnitude of control action when a contingency is detected is required. With increasing RAS in power system it is very difficult to predict the effect of one RAS action on rest of the system or with respect to another RAS logic. Therefore,

all the RASs in the neighborhood need to coordinate with each other to make a uniform decision on the best possible control action. The timing of control action is very critical and it should be effective in few milliseconds. So, performing the coordination with neighboring RASs after the contingency is detected will lead to unnecessary delays. The coordination with neighboring RASs should be started in advance when an abnormal behavior is initially detected.

Consider a power system as shown in Figure 5.3. It consists of a group of three RASs (RAS 1, RAS 2, RAS 3) which are capable of communicating with each other. Generator 2 is connected to transmission line, Line-1 and Line-2. Line-1 and Line-2 each carry 50 percent of the current. Now, suppose a fault is observed at Line-1. As a result, Line-2 is over loaded and closely approaching maximum capacity (1.8 p.u). RAS 1 detects the over current contingency and wants to trip Line-2. A consensus algorithm can be started between all RASs in the group to make sure other regions/areas do not destabilize while trying to trip Line-2. If RAS 2 can shed some load (Load 1 or Load 2), it can reply with `NOT OK`. Similarly, other substations can reply with `OK`/`NOT OK` based on the operating condition of their region. If all reply with `OK`, RAS 1 can go ahead to trip Line-2. If one of them reply with `NOT OK`, RAS 1 aborts the trip action.

A general use case for DRAS using DCBlocks is described Table 5.3. A group of neighboring RASs can be formed using Group Management block of DCBlocks. Consensus on the required control action for the detected contingency can be done using Consensus block of DCBlocks. Coordination messages can consist of the following information (these messages can be exchanged with neighboring sub stations in the consensus rounds):

i. Type of contingency

ii. Faulty transmission line number

iii. Control action (e.g trip line-1)

iv. Consensus reply − `OK` or `NOT OK`.

Table 5.3: Use Case for Decentralized RAS

Use Case	**Decentralized Remedial Action Schemes**
Description	This use case describes DRAS where RAS logic in the neighborhood can communicate with each other and form a group. When local contingency is detected by any RAS in the group, it can start a consensus algorithm with other RASs in the group and reach consensus on the necessary control action to be taken resolve the contingency.
Input	Phasor measurements from local PMUs, breaker status of bus and lines connected at each bus
Output	Control actions such as line tripping, load shedding, generator tripping, etc., depending on the type of contingency.
Actors	RASs
Assumptions	Each RAS has computing device capable of running DRAS software. All the substations are connected by very high speed communication network.

continued ...

. . . continued

Use Case	Decentralized Remedial Action Schemes
Steps	i. On system start up, each RAS checks its group designed beforehand based on: (a) Electrical distance. (b) Geographical distance between adjacent RASs. Each RAS joins the group using `addMember` method of Group Management block of DCBlocks. ii. RAS logic continuously monitors the PMU data at each bus, transmission lines to check for any possible contingency. iii. If a RAS starts to detect a contingency in one of the transmission lines, it performs following steps. (a) Fill the parameters of coordination message with: i. Contingency type ii. Faulty transmission line number iii. Control action type (e.g. Trip Line 2) iv. REPLY - OK or NOT OK. iv. It starts `ALL OR NOTHING` type of consensus using Consensus block of DCBlocks. v. All substations check the coordination message and reply with `OK`/`NOT OK` in the next round of consensus based on their local operating condition. vi. If all nodes reply with `OK`, then consensus is reached. (a) The RAS logic in the affected area takes the agreed upon control action. vii. If any one of the substation reply `NOT OK`, then consensus is not reached. The affected RAS has to perform an independent alternate control action.
Issues	The consensus has to be performed well in advance. The consensus algorithm should terminate in few milliseconds since the RAS logic should be ready to apply the control action immediately. If the consensus is taking longer than few milliseconds, it should be immediately stopped.

5.5 Decentralized Wind Power Monitoring and Control

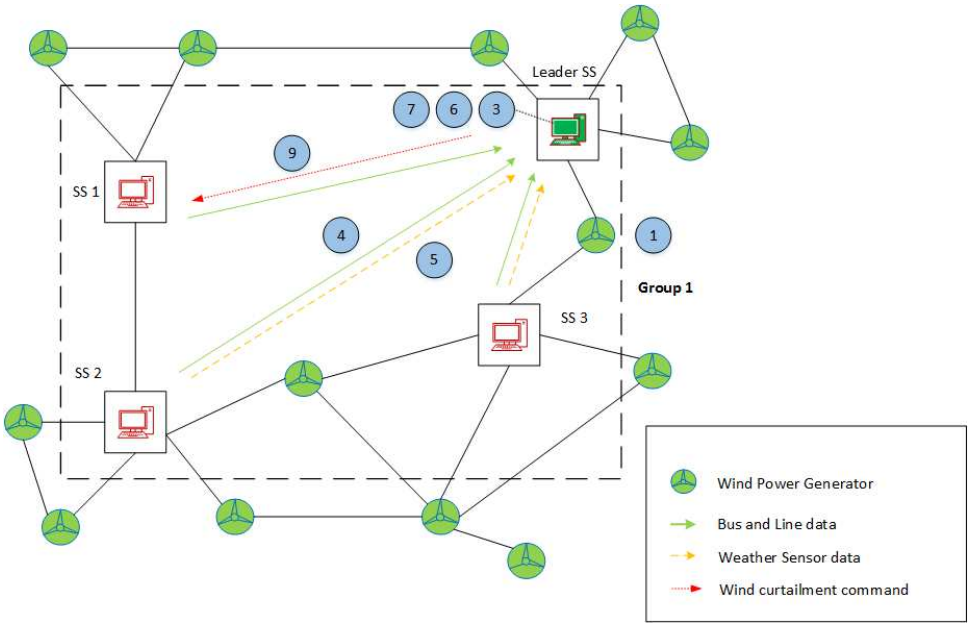


Figure 5.4: Decentralized Wind Power Monitoring and Control

Decentralized Wind Power Monitoring and Control application scenario described in Section 3.5 can also be developed using DCBlocks. Neighboring substations connected to wind farms can be grouped together as shown in Figure 5.4. One distinguished lead node in the group can perform wind power monitoring and control on behalf the group. The donut line sensors on transmission lines feed DLR data back to the lead node. The lead node calculates DLR for the transmission lines based on the DLR data from the donut line sensors and weather parameters. The DLR in combination with the measured line loads (voltage, current), is used by the leader to check if a line is overloaded. Then if required, take necessary control action steps like send load shedding commands or wind curtailment commands to the substations.

The use case for Decentralized Wind Power Monitoring and Control using DCBlocks is listed in the Table 5.4. A group of neighboring substations connected to wind farms can be

formed using Group Management block of DCBlocks. A leader sub station can be elected using Leader Election block of DCBlocks.

Table 5.4: Use Case for Decentralized Wind Power Monitoring and Control

Use Case	**Decentralized Wind Power Monitoring and Control**
Description	This use case describes decentralized wind power monitoring and control for small groups of substations connected to wind farms. The lead node performs the monitoring and control for the group.
Input	Weather data, bus and line information from each substation. DLR data from donut line sensors on transmission lines.
Output	Control actions like load shedding command or wind curtailment signal depending on the severity of over current issue.
Actors	Substations connected to wind farms, wind power generators, weather sensors, donut line sensors.
Assumptions	Each substation has computing device capable of running this software.

continued ...

... continued

Use Case	Decentralized Wind Power Monitoring and Control
Steps	i. Each substation joins a group using `addMember` method of DCBlocks Group Management Block based on the geographical distance between adjacent substations. ii. Each node starts leader election using `startElection` method of DCBlocks Leader Election Block. Each process sends its local computational ability value. iii. The node with the highest computational ability in the group is elected as the leader. A secondary leader is also selected as back up leader if the primary leader fails. iv. Each node starts to send its local PMU measurement to its lead node at regular intervals. v. All nodes connected with weather sensors send weather data to the lead node. vi. Donut line sensors send DLR data such as conductor temperature, sag temperature etc to DLR server software running on lead node. vii. The DLR output is fed to the DSE and load management applications to perform system analysis. viii. If the line current of some transmission line exceeds the threshold, leader sends load shedding command to the corresponding substation. ix. Steps iv to viii are repeated. If problem is not resolved, then leader sends wind curtailment command to corresponding substations to reduce the power generation. x. Steps iv through ix are repeated continuously.
Issues	The communication medium provided by DCBlocks and underlying DS platform should be able to detect failures like crash, omission failures and report to the application software. The substations can return to local mode in case of loss of communication link to lead node.

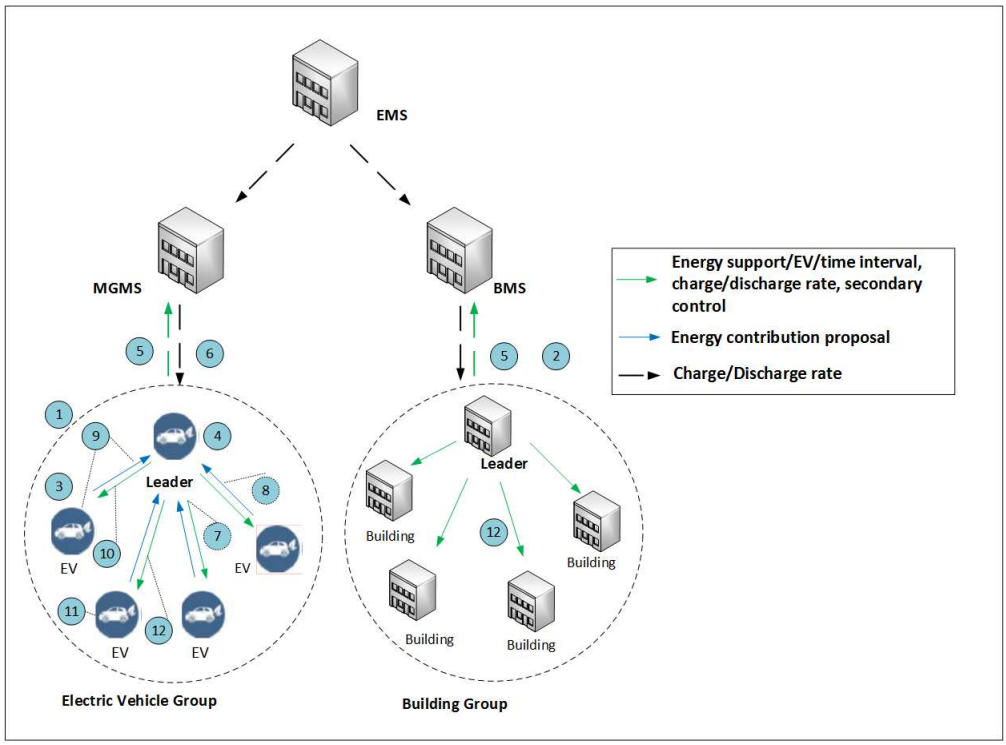


Figure 5.5: Decentralized Frequency Control With Demand Response

5.6 Decentralized Frequency Control with Demand Response

The reader is referred to Section 3.6 for a brief description of Decentralized Frequency Control. Short term frequency regulation and control can be performed by demand side prosumers for faster control and to reduce dependence on the AGCs and generation side controllers. The prosumers like building clusters, EVs and microgrids can be used for faster automatic frequency response since they are closer to loads when compared to generators and AGCs. They can respond more quickly to changes in load reference set point than large generators. There will be some spot price for both charging and discharging to provide incentive to EVs to support the grid. When the load is less than the generation output, the vehicle charging rate is less. So, EVs can start charging. Similarly, when the load is greater than the generation output, the vehicle charging is very high (and discharging rate increases), so EVs can discharge to support and also benefit from the transaction. For this

a two way transactive signal communication between the grid and prosumers such as EVs, building clusters is assumed.

Consider a power system scenario as shown in Figure 5.5. The figure shows a hierarchical structure consisting of peer groups comprising of EVs, building clusters at the bottom. EV groups are connected to Micro Grid Management System (MGMS) and building clusters are connected to Building Management System (BMS). These are connected to EMS on the top. All the EVs in a parking lot can be grouped together using group discovery DC algorithms discussed in Section 2.4.7. A leader can be elected for the EV group which can act as a coordinator between the EVs and the power grid. Similarly, a cluster of buildings can be formed using group Management Block based on the geographical distance. A leader for building cluster is elected using Leader Election Block. The leader of EV group communicate with MGMS and leader of building clusters communicate with BMS to get information such as current charge or discharge rate based on predicted load and system characteristics. The EVs send their possible energy contribution estimates per time interval (e.g., 2 sec, 5 sec etc.) to the leader. The leader collects all the proposals and calculates the energy contribution/EV/time interval based on the current load, charging rate etc. The EVs (with sensors) monitor the bus and line status using PMUs and IEDs connected to the buses and lines. When frequency deviation situation is observed, they respond by either charging or discharging based on agreed upon transaction behavior.

When an EV connects to a charging station, it tries to "discover" its nearest group (in the parking lot) and joins the group. After joining the group, the new node is notified of the leader ID. A new leader can be elected by running leader election. The new leader ID is communicated to MGMS. MGMS sends the charging rate to the leader based on predicted load and system characteristics. The EVs send their energy contribution estimates per time

interval (2 sec, 5 sec etc.) to the leader. The leader collects all the proposals and calculates the energy contribution/EV/time interval based on the current load, charging rate etc. All EVs monitor the bus and line status using PMUs and IEDs connected to the buses and lines.

When an frequency deviation from nominal value is observed and primary control is not sufficient, EVs and building clusters can contribute to secondary frequency control.

i. The leader sends UP or DOWN signal to charge/discharge all EVs in the group.

ii. EVs perform the charging/discharging operation as per their agreement.

After the system stabilizes, all the EVs can send their new energy supply proposals to the leader. New transaction steps get repeated. Whenever a node leaves the group (moves out of the parking lot or removes the charging plug), it should send "leave" message to the leader to indicate that it leaving.

Table 5.5: Use Case for Frequency Control Using Demand Response

Use Case	**Decentralized Frequency Control (DFC)**
Description	This use case describes DFC using demand side prosumers like EVs, building clusters. The EVs, building clusters form separate groups and help in secondary frequency control when required.
Input	Charging and discharging rate from MGMS, BMS, bus and line information from each nearby buses and lines.
Output	Energy consumed or discharged as per frequency overshoot or undershoot with respect to nominal value.
Actors	EV farms with IEMs, buildings with IEMs, MGMS, BMS, EMS, PMUs.
Assumptions	Electric Vehicles, Buildings have enough computing device for running this software.

continued ...

. . . continued

Use Case	Decentralized Frequency Control (DFC)
Steps	i. Each EV joins a nearby group based on group discovery mechanism. ii. Each building joins a nearby group using `addMember` method of Group Management Block based on the geographical distance between adjacent buildings. iii. Each node starts leader election using `startElection` method of Leader Election Block. Each process sends its response time, impact time and duration of stable period. iv. The node with best combination of all the proposed values is elected as the leader. v. Leader of EV group sends its leader ID to MGMS. Leader of building group sends its leader ID to BMS. vi. MGMS/BMS sends the charging/discharging rate to the leader based on predicted load and system characteristics. vii. Leader forwards the charging/discharging rate to its group members. viii. The EVs/buildings send their possible energy contribution estimates per time interval (e.g. 2 sec, 5 sec etc.) to the leader. ix. The leader calculates the contribution/EV/time interval based on the current load, charging rate etc. x. Leader sends the result (energy contribution by each EV) to each EV in the group. xi. Some EVs/buildings monitor the bus and line status using PMUs and IEDs connected to the buses and lines. xii. If a frequency deviation from nominal value is seen, the leader determines the secondary control action by: (a) Send UP or DOWN signal with new energy contribution values to all EVs, buildings in the group. xiii. The EVs and buildings contribute to frequency control by charging/discharging from/to the grid. xiv. When a node leaves the group, it should send "leave" message to the leader to indicate that it leaving.

continued . . .

. . . continued

Use Case	Decentralized Frequency Control (DFC)
Issues	The communication medium provided by DCBlocks and underlying DS platform should be able to detect failures like crash, omission failures and report to the application software.

Use case for Decentralized Frequency Control using computational blocks of DCBlocks is described below. A group of neighboring EVs (in a parking lot) can be formed using Group Discovery mechanism. A group of neighboring buildings can form a group using Group Management block of DCBlocks. A group leader for each group can be elected using Leader Election block of DCBlocks.

CHAPTER 6

DECENTRALIZED POWER APPLICATION DEMONSTRATION AND RESULTS

6.1 Introduction

Hitherto, the main principle of DC algorithms and power applications were described in detail. The DCBlocks design was presented in Chapter 4 to make it a viable solution for decentralized power scenarios. The applicability of DCBlocks to decentralized power scenarios have been described in Chapter 5. However, it is important to translate the design to a deployable implementation. The implementation of all the power scenarios is beyond the scope of this thesis and only one power scenario is considered. A prototype implementation of Distributed Voltage Stability (DVS) using DCBlocks is considered in this chapter. The power system used to implement DVS application is IEEE 30 bus power system. The power system is modeled in DETERlab (DETERLab, 2015) using 30 deter nodes where each deter node represents a substation connected to a power system bus. It is also assumed that each substation has a computing device capable of running the DVS software. DETERlab is a cyber-security testbed consisting of hundreds of nodes and can be used for testing experiments in which the nodes may be configured in a variety of ways with any of several existing operating system and with different network topology. It allows users to inject faults, introduce statistical delays, simulate network link failures and

security attacks like DDoS attacks, etc, which makes it a great platform to perform useful and realistic experiments (Goodfellow et al., 2013).

Consider a power system network as shown in Figure 6.1. It consists of a network of substations, which can be grouped together based on the electrical distance (electrical distance is the impedance path between two points), high voltage to reactive power sensitivity and reactive power availability. These substations (hereafter referred to as nodes) have the computing devices capable of computing Voltage Stability Indices (VSI) using reduced network model equivalent and synchrophasor measurements from Phasor Measurement Units (PMU). It is also assumed that they are capable of computing distributed state estimation before performing voltage stability assessment. When a voltage stability problem at a particular bus is encountered in the group, all the substations within that group communicate with each other to provide the reactive power needed to improve the voltage stability at that bus. The DLSE implementation is beyond the scope of this thesis.

So, on system start up, each node waits for the initial grouping from the control center. Once initial grouping information is received, all the nodes join their respective groups. After joining the group, all the nodes in the group coordinate with each other to select one distinguished node (leader) having the maximum computational ability to perform distributed state estimation and voltage stability assessment for the group. After the lead node is elected, all the other nodes start sending their local phasor measurements periodically to the lead node. The lead node runs the DVS monitor algorithm to check if there is a voltage stability issue in any of the buses in the group. If yes, it takes necessary control actions to improve the voltage stability. If the voltage stability issue still remains and the reactive power within the group is all used up, the lead node contacts other lead nodes for additional reactive power and dynamically reorganizes the group if necessary to acquire the additional

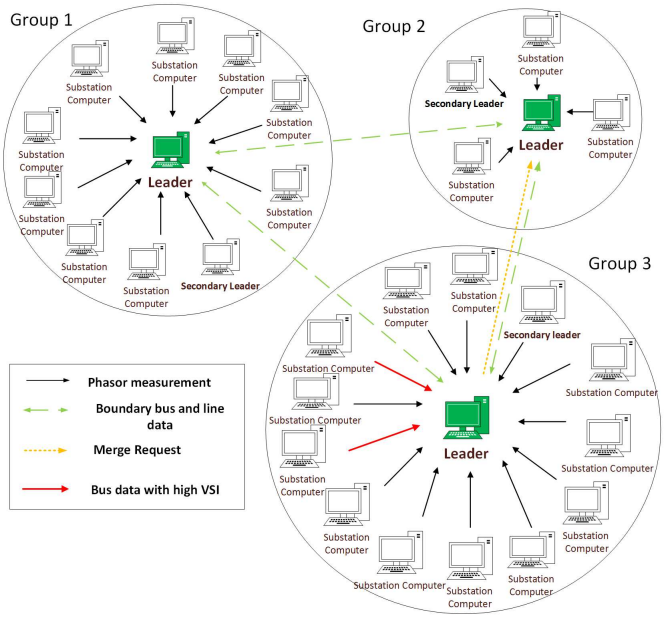


Figure 6.1: Initial Groups

reactive power. The groups split back to original groups after the voltage stability issue is resolved

Following sections describe each part of the implementation in detail. The first three sections describe the DVS power algorithms. The implementation and source code of these power algorithms were provided by research students of SGDRIL, WSU. These algorithms were integrated into the DVS application implementation.

Sections 6.6.1, 6.6.2, 6.6.3 describe the evaluation of DC building blocks described in Chapter 4 for various test scenarios such as increasing group size, varying failure conditions, and different network configurations. All the experiments were conducted in DeterLab test set up. Also, various test scenarios for DVS application were conducted and corresponding test setup and results are described in Section 6.6.4.

6.2 Initial Grouping

The large power system is divided into multiple smaller power system groups based on the electrical distance, voltage to reactive power sensitivity and reactive power availability factors. In order for these small distributed power system groups to provide short term and long term voltage stability, each initial group should contain at least

i. One generator above certain size.

ii. One transmission line.

iii. One load above certain size.

iv. One reactive power source above certain size.

v. All components are relatively close by electrical and geographical distance.

If there is a tie line (or boundary line) between two sub groups, then for power flow calculation, its impedance is considered as half of original line impedance. A virtual bus is connected to the boundary bus with half of the tie line impedance. The virtual bus can be a generator bus or a load bus depending on the line flow. If the line flow between two buses i and j is positive (power flow is from i to j), then it is replaced by a generator bus at boundary bus i and load bus at boundary bus j.

6.3 Distributed Voltage Stability Monitoring Algorithm

The DVS monitoring algorithm is run by the lead node of each group. The algorithm estimates Voltage Stability Index (VSI) for each load bus using Thevenins equivalent approach. For this, Thevinin's voltage V_{th} and Thevenin's impedance Z_{th} values for each bus in the group are calculated. The VSI values are then calculated using these values.

If the VSI value at a bus is close to 1, it indicates that the load bus is operating close to the edge. As the VSI is calculated for all the load buses in the group, the bus with the highest (maximum) VSI (i.e. the weakest bus) dictates the VSI of the entire group. So if the maximum VSI is greater than threshold (e.g, 0.7), control action is required. The maximum VSI will be used by Distributed Voltage Stability Control (DVSC) algorithm to control the use of reactive sources based on Priority Index (PI). The Distributed Voltage Stability Control algorithm is discussed in Section 6.4.

6.4 Distributed Voltage Stability Control Algorithm

Since the control action needs to be applied within few milliseconds, the reactive power resource should be close to the target load bus. Further, the initial groups need to be formed considering the electrical distance between buses and voltage to reactive power sensitivity factors. The lead node runs Distributed Voltage Stability Control Algorithm if it observes voltage stability issue in a particular target load bus.

The first part of the algorithm is to find Priority Index (PI) based on which the reactive power resources will be used. The Priority Index is calculated on the proximity of the reactive source to the target load bus, electrical distance and network sensitivity of the reactive power source to the target load bus. The first priority is given to reactive power source closest to the load bus having the voltage stability issue. Second priority is given to reactive power resources from those lines which are directly connected to the target bus based on the ascending ranking of electrical distance to the target load bus. The last set of priorities is given based on the ascending ranking of cumulative electrical distance to the target load bus (not directly connected to target bus).

Once the PI is found, the required reactive power (Q_{req}) for compensating the voltage

problem on target bus is calculated (using part of Jacobian matrix). Finally, the reactive power is applied based on the calculated Q_{req} and PI. The control action need to be performed multiple times if the voltage problem is not solved in one step control action. The DVSC algorithm uses all reactive power reserve until the problem is solved within the group. If the voltage problem cannot be not solved within the group due to insufficient reactive power, the lead node communicates with other group lead nodes for routing more reactive power. Two or more groups are merged together based pn best reactive power availability.

6.5 Distributed Voltage Stability integrated with DCBlocks

This section described the integration of the DVS power algorithms with DCBlocks. The combined implementation software is referred to as DVS power application here. The DVS power application is responsible for coordinating all activities like form initial grouping, start leader election, start leader activities, send power measurements periodically to the leader, start merge/split operations if necessary, etc. The communication between the processes is accomplished by DCBlocks and the underlying Akka Java framework. A brief description of each of the classes is given below.

i. `DVSAppMain` - This class is the main entry point of the DVS application. The class performs the following activites before handing over the control to `DVSAppMgr` class.

   (a) Initialize Group Management Block of DCBlocks.

   (b) Join a group based on the initial grouping technique described in section 6.2 using DCBlocks Group Management Block.

   (c) Instantiate `DVAppMgr` to manage the DVS application activities.

ii. `DVSAppMgr` - This class is the core class of the DVS application. It is responsible for

coordinating all the activities between the DCBlocks and DVS application specific classes. The class performs the following activities.

(a) Wait for notification from DCBlocks Group Management Block informing that its group is ready (i.e. initial group size has been reached) for normal operation.

(b) Start Leader Election using DCBlocks Leader Election Block to elect a lead node for the group.

(c) If the node is the lead node, it instantiates the Leader class to manage leader activities. It also publishes the leader ID and associated reference using DCBlocks Ordered Multicast Block to the rest of the group members.

(d) If the node is not the lead node, it waits for leader reference from the lead node.

(e) It instantiates `MeasurementSender` in order to send local phasor measurements to the leader periodically.

(f) If Group Management Block notifies that the primary leader has failed, `DVSAppMgr` informs `MeasurementSender` to stop sending messages to primary leader and switch to sending the data to the secondary back up leader.

(g) If the secondary leader also fails, it starts a new leader election using Leader Election Block.

(h) Coordinates activities for merge and split operation.

iii. `MeasurementSender` - This class is responsible for sending local phasor measurement message to the lead node. Each node periodically reads its local bus and line info using `PowerDataTextParser` class and sends a message containing this info to the lead node.

iv. `PowerDataTextParser` - This class provides methods to read bus and line information from different case files. For this implementation, the power system behavior is simulated offline using Real Time Digital Simulator (RTDS) and the required bus (voltage, current, shunt capacitance value) and line (impedance, capacitance etc.) information for various case scenarios is captured and stored in separate case files.

v. `Leader` - This class is responsible for collecting all the phasor measurement messages from all the nodes in the group. It sends the collected data to `LeaderWorkExecuter` class to work on the collected data. It also initiates merge and split operation based on the request from `LeaderWorkExecuter`.

vi. `LeaderWorkExecuter` - This is a worker class for Leader. It performs the following activities:

   (a) Collect all the measurement data from the `Leader`.

   (b) Call the DVS power algorithm method of `DVSProcessor` class to perform monitor and control action using the collected data.

   (c) It checks if monitoring and control action is successful. If the reactive power reserve (shunt capacitance) is empty, it sends a merge request message to the `Leader` to start merge operation. Similarly, also sends split operation request.

vii. `DVSProcessor` - This class performs the monitor and control action as described in Section 6.3 and Section 6.4.

The implementation steps of DVS application using DCBlocks is given below. Each node and group is assigned a unique node id and a unique group id.

i. On system start up, the `DVSAppMain` determines the initial grouping as explained in Section 6.2.

ii. The DVS application running on each node initializes the group management block by calling the `registerToGroups` method of Group Management Block.

iii. Based on the initial grouping, each node join its group using the `addMember` method of `GroupMgmt` interface. The initial grouping is shown in figure 6.1.

iv. It hands over the activities to `DVSAppMgr`. `DVSAppMgr` also registers with Group Management Block using `registorOnGroupReady` to get notified when the group is ready (i.e. group has reached a certain size).

v. Once the group is ready, each node starts leader election using `startElection` method of `LeaderElection` interface. The Leader Election Block publishes its local vote (in this case, local computational ability value) to the group. All the nodes collect the proposed votes in subsequent rounds and finally the node having the highest vote (i.e. best possible computational power) in the group is elected as the leader. A secondary leader is also selected as back up leader if the primary leader fails.

vi. `DVSAppMgr` checks if leader ID is its own node ID. If yes, it instantiates `Leader` class to start leader activities. The `Leader` starts the leader activities (i.e, wait for incoming power measurements from all group members, run DVS monitoring and control algorithms as required).

vii. The lead node publishes its ID to other groups for future communication using `publishMsgToOtherGroup` of `GroupMgmt` interface.

viii. `DVSAppMgr` instantiates `MeasurementSender` class to send local power measurement

data. So, each node to send its local power measurement (bus and line information) to its group leader at regular intervals (every 30 seconds).

ix. The boundary bus and tie line information (line connecting 2 substations in adjacent groups) is obtained from the adjacent group leaders.

x. Once all the power measurements are received, `Leader` submits to `LeaderElectionExecuter` to run the DVS monitoring algorithm as per Section 6.3.

xi. If the leader detects that the maximum VSI is greater than threshold (0.7), it performs the control action using the DVS control algorithm as described in Section 6.4. It might take several control action attempts to resolve the problem.

xii. If the problem cannot be resolved within same group, i.e. reactive power is insufficient then merging of one or more adjacent groups becomes necessary.

xiii. The lead node requests adjacent group leaders to send an estimate of reactive power reserve in their group by calling `publishMsgToOtherGroup` of `GroupMgmt` interface. The best candidate group is selected and the lead node sends a merge request message to that group. This is shown as orange arrow in figure 6.1.

xiv. The `DVSAppMgr` of lead node initiates the merge activity with adjacent group by calling `mergeGroups` method of `GroupMgmt` interface. This is shown in figure 6.2. The group management block calls the call back function to indicate the merge operation is completed. Each node begins to associate itself with the new merged group.

xv. The existing leader of the new group publishes its ID to merged group members using `PublishMsg` method of `GroupPublisher` interface.

xvi. Steps vi to xi are repeated till voltage stability problem is resolved.

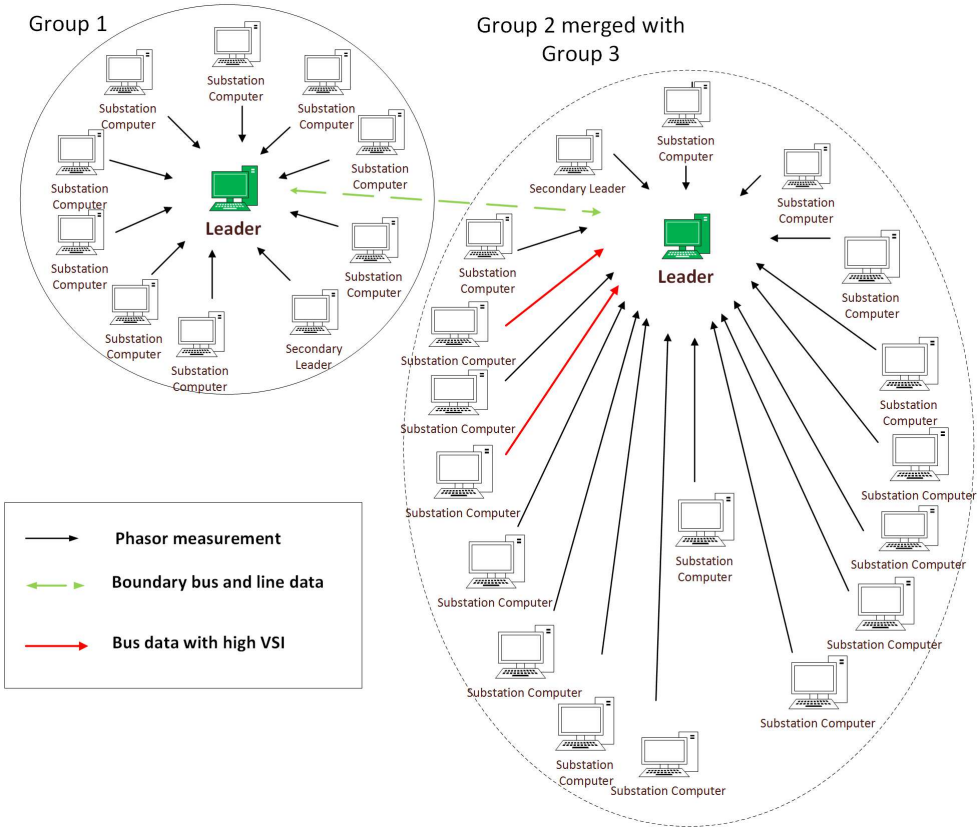


Figure 6.2: Merged Groups

xvii. When the voltage stability problem is resolved, the group is regrouped (split) into original two groups using `splitGroups` method of `GroupMgmt` interface. An indication (call back function) is given to the higher level application code that the split operation is completed.

xviii. Steps iii to xviii is repeated continuously.

6.5.1 Fault Tolerance Support

DCBlocks can detect and handle failure types like crash, omission, and timeout failures. It can tolerate upto $f = (n-1)/2$ faulty processes in a group where n is the total number of processes in the group. `DVSAppMgr` registers with DCBlocks to get notified on primary and secondary leader failure using methods `registerOnPriLeaderFailure` and `registerOnSecLeaderFailure`. If the leader is detected as faulty, DCBlocks group management block reports it to the `DVSAppMgr`, and then a secondary back up leader is made to take over the primary leader activities. If secondary leader also fails, then a new leader election is initiated.

6.6 Simulation Results

This section lists the various simulation results for each implementation block of DCBlocks and DVS application. All the experiments were conducted in DeterLab. To test the behavior of implementation blocks of DCBlocks, various test scenarios were simulated for changing group topology, different type of failures and different network configurations. A test application was used to run all the test scenarios for each implementation block. Similarly tests were conducted for DVS application to test the behavior of the application for various power scenarios and failure conditions. The corresponding simulation results for

DVS application is described in Section 6.6.4.

6.6.1 Results for Group Management Block

Experiments were conducted in DeterLab to profile the time taken for all the nodes in a group to join the group and for the group to reach group ready state to start normal operation. Each node joins the group during initialization phase. After the required number of nodes (group size) have joined the group, the Group Management Block triggers an indication to the application logic via a callback function that the group is ready. For this experiment, a network bandwidth of 500Mb/s and message latency of 0ms was chosen.

- Group Ready Status - Figure 6.3 shows the time taken to reach group ready state for varying group size of 10, 20, 30, 40 and 50 nodes. For a small group size of 10 and 20, the 50 nodes were divided into multiple groups and all the groups were started simultaneously in parallel. During initialization, each node joins its group (grouping is provided in XML file). After all the nodes have joined the group, a group ready indication is given to the application. As can be seen in the Figure 6.3 as the number of nodes increase, the time taken to reach group ready state is longer. This implies that with smaller group size, the nodes can start normal operation faster.

6.6.2 Results for Consensus Block

6.6.2.1 Simple Consensus

Here, experiments were conducted in DeterLab to profile the time taken for all the nodes in a group to reach simple consensus for different group size. At the end of each round, an appropriate sleep period is added to wait for incoming messages from all the members of the group. The sleep period (timeout) is increased as a percentage of group size. If a

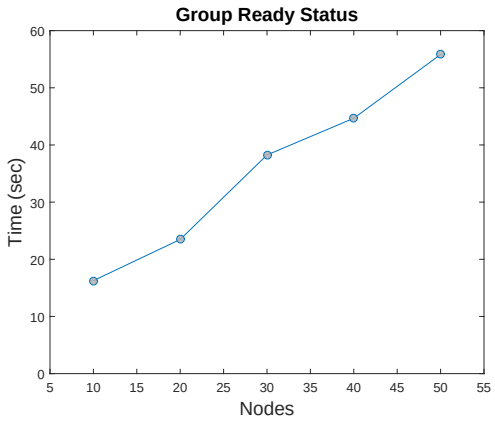
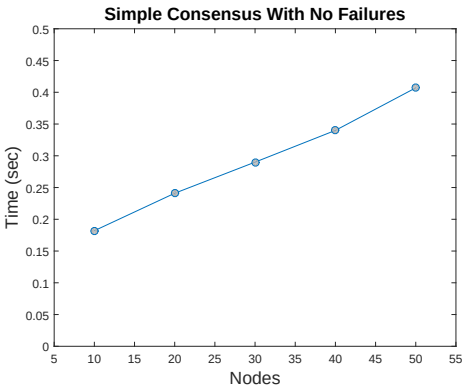


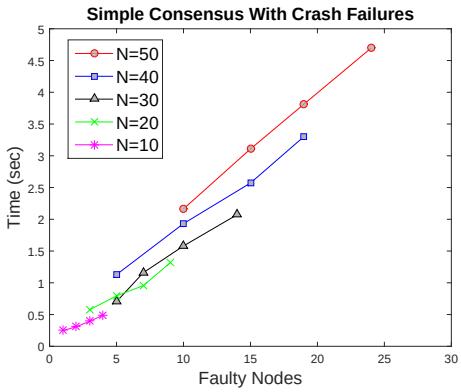
Figure 6.3: Group Ready Status For Different Group Size

message is not received within this timeout, an error value of "-1" is filled in the slot. For this experiment, a network bandwidth of 500Mb/s and message latency of 0ms was chosen. Here, the number of nodes in a group is represented by N and number of failures/faulty nodes in a group is represented by F. Figure 6.4 shows the behavior of Simple Consensus block for the no failure and failure conditions.

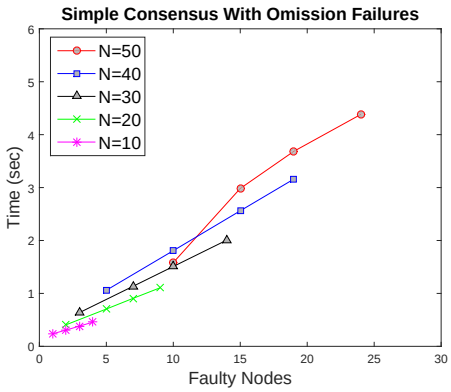
- Simple Consensus With No Failures - This case represents a scenario to reach simple consensus to find the average of all the proposed values. There are no failures at the start of the simple consensus algorithm. In the first round, all nodes in the group publish their local values to all the other members in the group. In the next round, each node has received all the proposed values from other nodes in the group. Here, since there are no failures (F = 0), algorithm stops in the second round. Each node has received all the values and so calculates the average of all the received values. The decision value (average) is same for all the nodes in the group. As shown in the Figure 6.4a, as the number of nodes in a group increases, the time taken to converge all increases clearly implying that the consensus is reached faster with smaller number of nodes in a group.



(a) Increasing Group Size And No Failures



(b) Increasing Number of Crash Failures



(c) Increasing Number of Omission Failures

Figure 6.4: Simple Consensus For Normal, Crash and Omission Failure Cases

- Simple Consensus With Crash Failures - Figure 6.4b shows the behavior of simple consensus implementation in the presence of crash failures. The experiments were conducted for varying group size, i.e, $N = 10, 20...50$ nodes. For each group, the number of crash failures were increased continuously upto $F = (N-1)/2$. As seen in the Figure 6.4b, as the number of failures increase, the time taken to converge also increase. This is because the number of rounds $(F+1)$ is directly proportional to number of failures observed during the run of the algorithm. For a large group size of 50 nodes, as the number of crash failures in the group increases, the time taken to converge is very high. For example, for a case of 20 crash failures, the algorithm is executed for 19 rounds $(F+1)$ leading to a delay of 1.3 sec. And an early decision can be made instead of executing all the $F+1$ rounds endlessly.

- Simple Consensus With Omission Failures - Figure 6.4c shows the behavior of simple consensus implementation in the presence of omission failures. The experiments were conducted for varying group size, i.e, $N = 10, 20...50$ nodes. For each group, the number of omission failures were increased continuously up to $F = (N-1)/2$. Some nodes omit to send messages to the group. These processes do not send messages for all the rounds of algorithm.

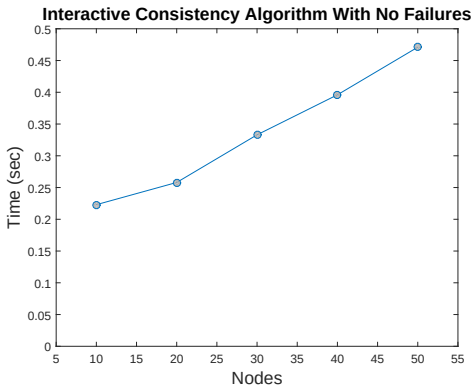
The implementation is non optimized. Further reduction in the timeout delay between consequent rounds is necessary. A practical upper bound for the number of execution rounds in presence of high number of failures need to be estimated. An early decision (to reach consensus or abort) should be made instead of executing all the $F+1$ rounds endlessly.

6.6.2.2 Interactive Consistency Algorithm

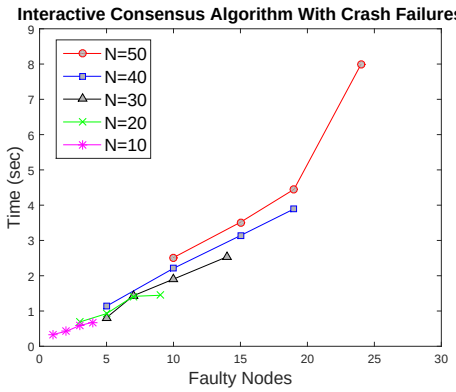
Here, experiments were conducted to profile the time taken for all the nodes in a group to decide on the same decision vector (one slot for each node in the group). At the beginning of first round, each node proposes its local value. In the subsequent rounds, rx vectors from each node is received. A new tx vector is calculated based on the majority of received values for each vector slot. This new vector is then transmitted in the next round. At the end of each round, an appropriate sleep period is added to wait for incoming messages from all the members of the group. The sleep period (timeout) is increased as a percentage of increasing group size. If a message is not received from a node within this timeout, an error value of "-1" is filled in respective slot. For this experiment, a network bandwidth of 500Mb/s and message latency of 0ms was chosen. Here, the number of nodes in a group is represented by N and number of failures/faulty nodes in a group is represented by F. Figure 6.5 shows the behavior of ICA block for the no failure and failure conditions.

- ICA With No Failures - Figure 6.5a shows the behavior of ICA implementation in the presence of no failures. This represents the ideal case of zero failures, high network bandwidth and zero message latency. Simulations were conducted for increasing group size ($N = 10, 20...50$) and the corresponding convergence delay were noted. As seen in the figure, with the increasing group size the time taken to converge also increases. This is because the sleep period (or timeout) between consecutive rounds is set based on the number of nodes in the group.

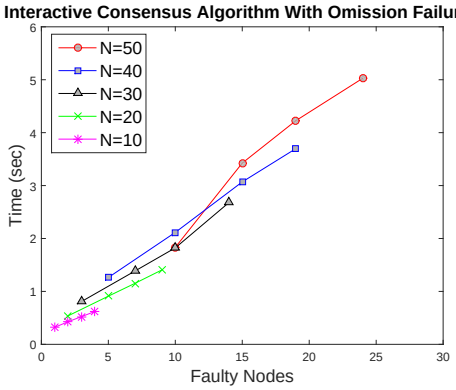
- ICA With Crash Failures - Figure 6.5b the behavior of ICA implementation in the presence of crash failures. The experiments were conducted for different group size ($N = 10, 20...50$). For each group size, the number of crashed nodes were constantly



(a) Increasing Group Size And No Failures



(b) Increasing Number of Crash Failures



(c) Increasing Number of Omission Failures

Figure 6.5: ICA For Normal, Crash and Omission Failure Cases

increased upto a maximum acceptable limit of $F = (N-1)/2$. As seen in the Figure 6.5b, the delay shoots up considerably for as the number of crashed nodes reaches close to maximum limit. This is because, as explained earlier, the number of execution rounds is directly proportional to number of faulty nodes (rounds $= F + 1$).

- ICA With Omission Failures - Figure 6.5c shows the behavior of ICA implementation in the presence of omission failures. The experiments were conducted for varying group size, i.e, $N = 10, 20...50$ nodes. For each group, the number of omission failures were increased continuously upto $F = (N-1)/2$. Some nodes omit to send messages to the group. These processes do not send messages for all the rounds of algorithm.

The implementation is non optimized. Further reduction in the timeout delay between consequent rounds is necessary. Also, an early decision to reach consensus or to abort the algorithm needs to be adopted to avoid executing the algorithm continuously if the number of failures observed in the system is above a predetermined threshold (optimum number below $f < (n-1)/2$).

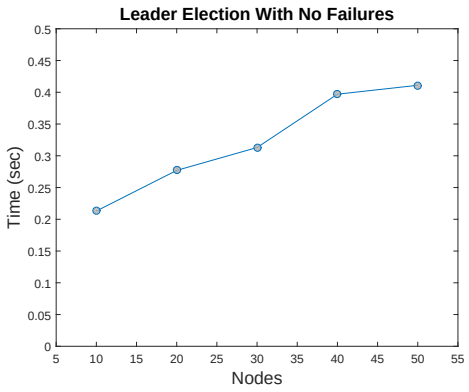
6.6.3 Results for Leader Election Block

This section describes the behavior of Leader Election Block for different group size, different failure conditions and network configurations. Experiments were conducted in DeterLab to profile the time taken to elect a leader for the group. The node having the smallest proposed value among all the proposed values is selected as the leader for the group. Initially, each node sets up parameters for future leader election by registering its vote (proposal value) and decision function (find the smallest vote) with the Leader Election Block. Next, the test application starts the leader election. The leader election is executed

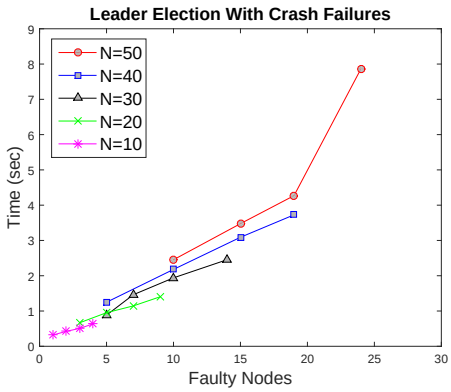
using ICA. In the first round, each node proposes its local vote (random number) to the group. In subsequent rounds, each node try to collect all the proposed votes from all the nodes in the group. After the final round, each node contains the same set of proposed vote list. So, it applies the decision function to find the smallest vote. The corresponding node is elected as the primary leader. The node with second smallest proposed vote is selected as the secondary leader. At the end of each round, an appropriate sleep period is added to wait for incoming messages from all the members of the group. The sleep period (timeout) is increased as a percentage of group size. If a message is not received within this timeout, an error value of " -1" is filled in the slot. For this experiment, a network bandwidth of 500Mb/s and message latency of 0ms was chosen. Here, the number of nodes in a group is represented by N and number of failures/faulty nodes in a group is represented by F. Figure 6.6 shows the behavior of Leader Election block for no failure and failure conditions.

- Leader Election With No Failures - Figure 6.6a shows the behavior of leader election implementation in the presence of no failures. This represents the ideal case of zero failures, high network bandwidth and zero message latency. Simulations were conducted for increasing group size ($N = 10, 20...50$) and the time taken to elect primary and secondary leader were noted. As seen in the figure, with the increasing group size the time taken to elect a leader also increases. This is because the sleep period (or timeout) between consecutive rounds is set based on the number of nodes in the group. The leader election block internally uses ICA to get consistent view of all the proposed votes. Hence, the behavior is similar to ICA explained in previous section 6.6.2.2 for no failure case.

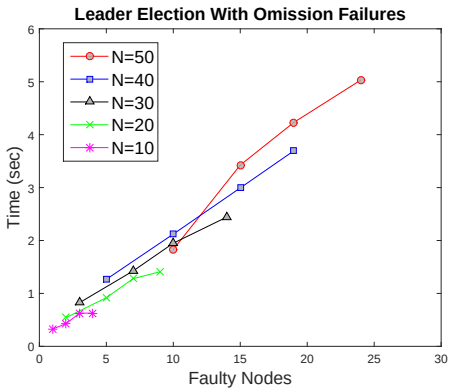
- Leader Election With Crash Failures - Figure 6.6b the behavior of leader election



(a) Increasing Group Size And No Failures



(b) Increasing Number of Crash Failures



(c) Increasing Number of Omission Failures

Figure 6.6: Leader Election For Normal, Crash and Omission Failure Cases

implementation in the presence of crash failures. The experiments were conducted for different group size ($N = 10, 20...50$). For each group size, the number of crashed nodes were constantly increased upto a maximum acceptable limit of $F = (N-1)/2$. The leader election block internally uses ICA to get consistent view of all the proposed votes. Hence, the behavior is similar to ICA explained in previous section 6.6.2.2 for failure case. With the increasing group size and increasing faulty nodes, the time taken to elect a leader is very high. This high delay needs to be reduced by making a early decision in presence of high number of faulty nodes. Figure 6.6b shows the behavior of leader election block in presence of increasing faulty nodes.

- Leader Election With Omission Failures - Figure 6.6c shows the behavior of LE implementation in the presence of omission failures. The experiments were conducted for varying group size, i.e, $N = 10, 20...50$ nodes. For each group, the number of omission failures were increased continuously upto $F = (N-1)/2$. Some nodes omit to send messages to the group. These processes do not send messages for all the rounds of algorithm.

As mentioned in 6.6.2.2, the algorithm needs to be further optimized to reduce the delay.

6.6.4 Simulation Results of Decentralized Voltage Stability Application

This section describes the simulation setup used for testing DVS application along with test scenarios and corresponding results. To test DVS with DCBlocks, a IEEE 30 Bus power system is considered (of Washington, 1993). It is assumed that extra controllable shunt capacitors and Static VAR compensator (SVC) are installed in load buses (bus 3, 7, 10, 14-22, 24, and 30) for control action. The reactive source is provided by changing

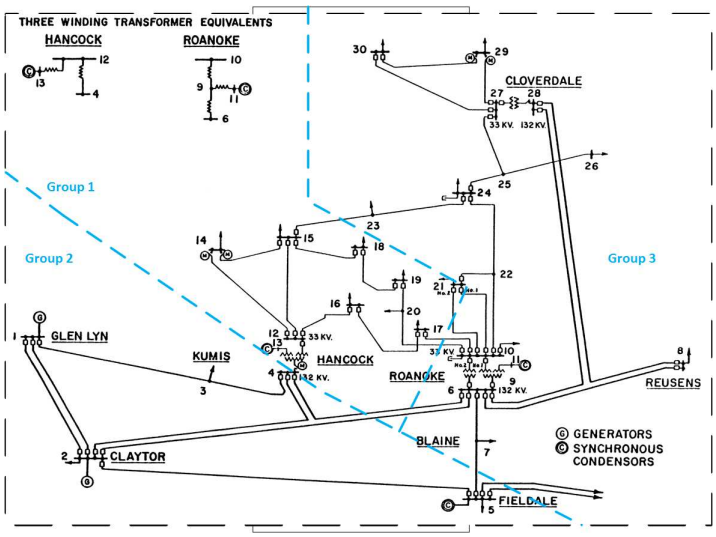


Figure 6.7: IEEE 30 bus power system divided into 3 groups

the transformer tap, re-scheduling generator, and load shedding. The figure 6.7 shows the IEEE 30 bus system with initial grouping.

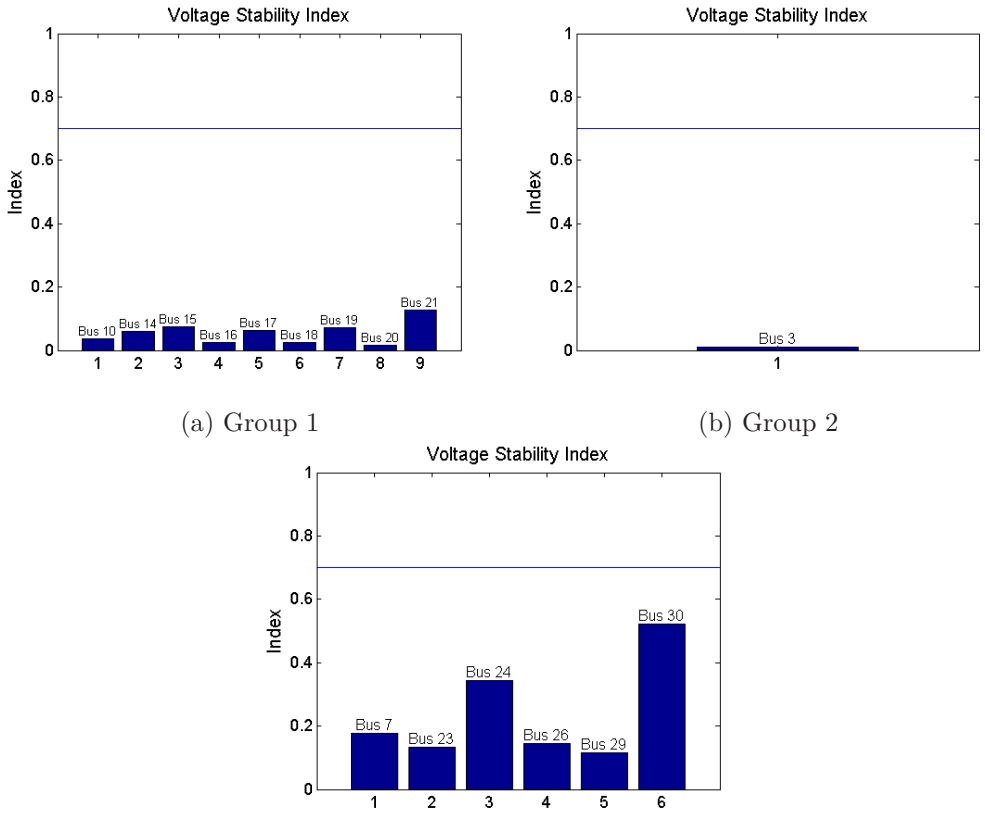
Each group comprises of following buses.

i. Group 1 = [10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]

ii. Group 2 = [1, 2, 3, 4, 5]

iii. Group 3 = [6, 7, 8, 9, 11, 22, 23, 24, 25, 26, 27, 28, 29, 30]

DVS application with DCBlocks were simulated in DETERLab testbed. The assumption made here is that each substation has a computational device capable of running this integrated software. The Deter nodes are connected via LAN with a network bandwidth of 500Mbps. With this test setup, a case study of three voltage stability scenarios were conducted and simulation results were captured. The description of each scenario and corresponding results are described in following sections.



(a) Group 1

(b) Group 2

(c) Group 3

Figure 6.8: Voltage Stabiltiy Indices for each group at normal case

6.6.4.1 Normal condition

In this scenario, the VSI values at each bus is well below the threshold and the system is working in normal operating condition. Each substation sends the bus and line information to its group leader every 30 seconds. The leader collects the bus and line data from all the group members, boundary bus and line data from adjacent group leaders and runs the DVS monitoring algorithm. The test results for the three groups for this case is shown in figure 6.8. It can be seen that all the load buses in Group 1, Group 2 and Group 3 have VSI values below the threshold.

6.6.4.2 Voltage stability problem and resolved within the group

The simulation results for all groups for this scenario is shown in figure 6.9. Here, VSI values of the some buses in one of the group is above threshold and a control action is required.

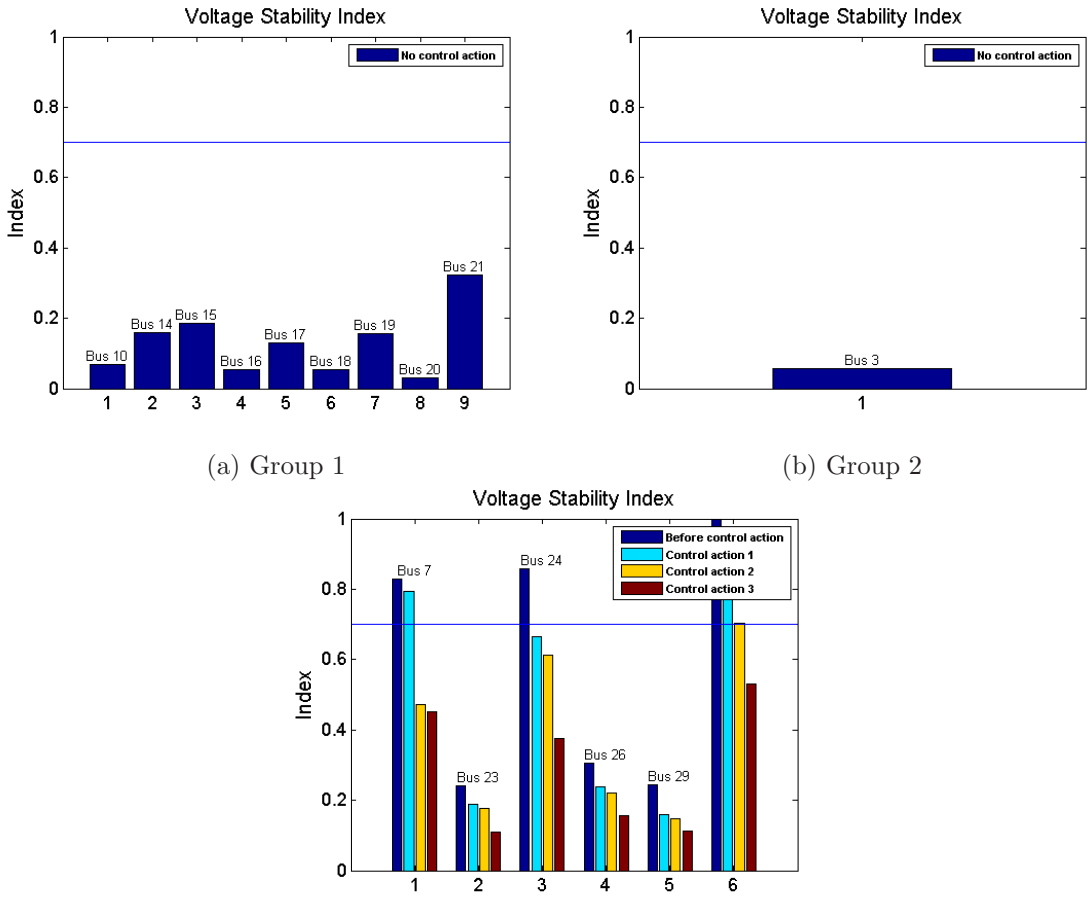
As shown in figure 6.9a and 6.9b, the VSI values of the load buses are below threshold. Whereas in figure 6.9c, the values for Bus number 7, 24 and 30 in Group 3 are above the threshold and leader has to perform control action to resolve the problem. In each control action attempt (marked as Control action 1, 2 and 3), reactive power is applied to the affected buses as per priority index. The reactive power reserve in Group 3 is sufficient to resolve the issue. The voltage stability problem in bus 7, 24 and 30 is resolved in three control action attempts as seen in figure 6.9c,.

6.6.4.3 Voltage stability problem and regrouping is needed

The simulation results for all groups for this scenario is shown in figure 6.10. Here, VSI values of the some buses in one of the group is above threshold and a control action is required. The reactive power reserve is insufficient in the group, and merging of two groups is needed to route more reactive powers.

As shown in figure 6.10a, 6.10b, the VSI values of load buses of Group 1 and 2 are below threshold. Whereas in 6.10c, VSI values for bus number 7, 24 and 30 in Group 3 are above the threshold. The leader of Group 3 has to perform control action to resolve the problem. The reactive power reserve in Group 3 is not sufficient to resolve the issue, Group 3 merges with Group 2 to resolve the issue. After couple of control action attempts, the problem is resolved and the merged group is split into original Group 2 and Group 3.

6.10c shows that even after three control action attempts, VSI of bus 30 is above



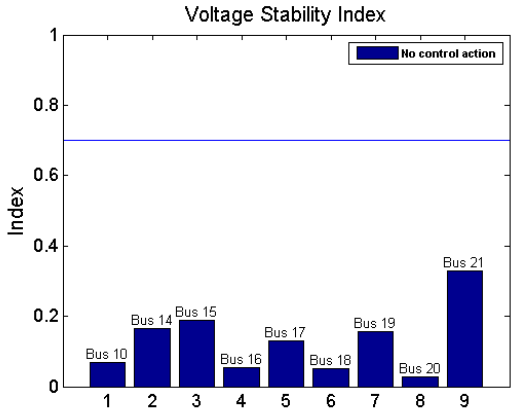
(a) Group 1

(b) Group 2

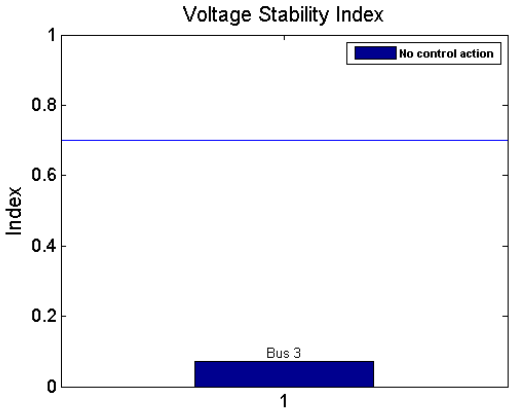
(c) Group 3

Figure 6.9: Voltage Stabiltiy Indices for each group at increased load: Case 1

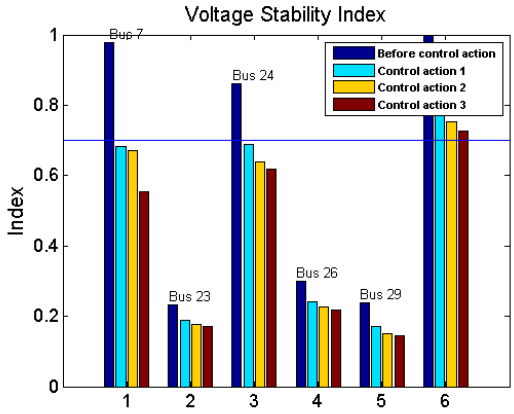
threshold. So, the leader of Group 3 starts merge operation with Group 2 to resolve the problem. The old leader of Group 2 is reelected as leader of the merged group and all the substations send the bus and line information to it. The leader of Group 2 finds that the maximum VSI is above the accepted threshold and so runs the control action algorithm. 6.10d shows that after two control action attempts, the issue is resolved. The groups split back into original Group 2 (figure 6.10e) and Group 3 (figure 6.10f) and continue with normal operation. The merged group is split back into original groups after problem is resolved because the group size may continue to grow and finally become one large group comprising of entire power system (similar to centralized design).



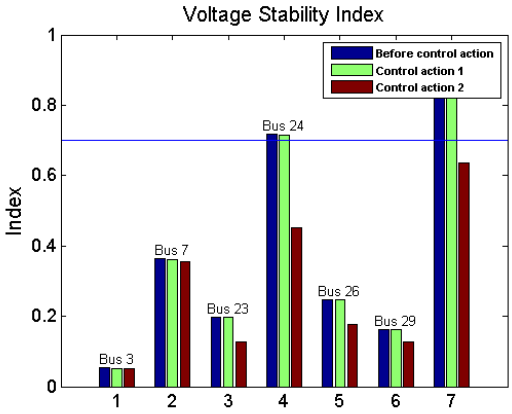
(a) Group 1



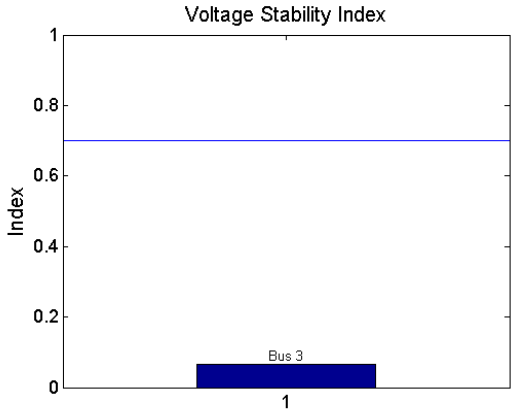
(b) Group 2



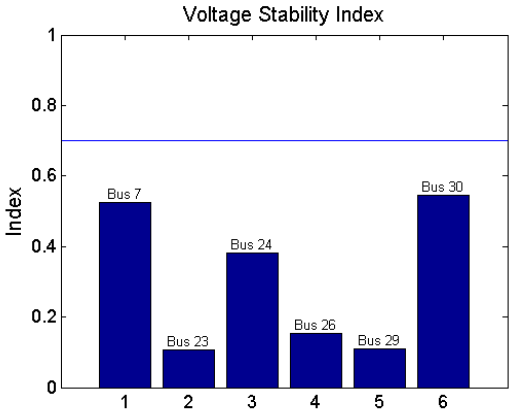
(c) Group 3 - voltage index problem in Bus 30



(d) Group 3 merged with Group 2 with control actions on Group 2



(e) Split into original groups : Group 2



(f) Split into original groups : Group 3

Figure 6.10: Voltage Stabiltiy Indices for each group at increased load: Case 2

CHAPTER 7

SUMMARY AND CONCLUSIONS

7.1 Summary and Conclusions

Decentralized Coordination (DC) algorithms for distributed systems are widely used in applications such as online banking, online shopping, distributed database management, the internet and many more. However, the application of DC algorithms in large scale engineering infrastructure is not very common. This thesis attempts to provide DC solutions to one such engineering systems infrastructure: the power grip infrastructure. The traditional CC based monitoring and control used in power system applications is no longer feasible with increasing in flux of IEDS, DERs and battery chargeable loads. It is becoming increasingly becoming difficult to provide centralized monitoring and control due to large set of system variables, intermittent nature of DERs and unpredictable behavior of increasing number of loads. On the other hand, a complete local control with only local system visibility maybe fast in solving the local problem, but it may have a cascading effect on the neighboring areas and ultimately hinder the overall system performance. Hence, the need for supplementing the conventional CC based power system control to a more decentralized power system control is being advocated in our submitted paper (Banerjee et al., 2015). There are several decentralized power algorithms being proposed for decentralized power applications. But, the change from a traditional central or local architecture to a more decentralized archi-

tecture need to be supported with a robust decentralized computing and communication infrastructure. This thesis attempts to leverage the vast body of theoretical and pragmatic research in DC and build a collection of decentralized coordination algorithms that is most suitable for decentralized application scenarios and to make them more robust and resilient to failures.

Overview of few important sub problems of DC such as consensus (a.k.a agreement), leader election, multicast message ordering, mutual exclusion, voting, group membership, group discovery, supply-agreement were described. Some of the applicable DC algorithms (both applied and theoretical) for each sub problem with corresponding pros and cons were discussed. The focus was on finding optimal DC algorithms for synchronous systems such as the power systems. Out of the set of DC algorithms presented for Consensus sub problem, Simple Consensus and ICA seemed like a good fit for power system applications. Among the set of leader election algorithms discussed, an adaptation of ICA with a generic power application specific decision criteria (e.g., cyber-physical criteria) to elect a leader seemed like the best option. Reliable mulitcast with same ordering semantics and atomicity is necessary to build robust decentralized power applications. Total-Ordered multicast or ABCAST delivery of messages is required so that it is gauranteed that all the communicating processes in a group receive messages in the same order. Among the mutual exclusion algorithms, Paxos based ME algorithm is picked as most suitable. The importance of group membership management is also discussed. Group discovery DC algorithms are applicable to mobile *ad-hoc* networks such as for electric vehicular communication. The implementation of DC algorithms for voting, group discovery and supply-agreement sub problems is out of scope for this thesis.

A few present and emerging decentralized power algorithms available in the literature

were discussed. The survey demonstrated the lack of concentration on the distributed computing and communication aspect of developing decentralized power applications. So, a possible applicability of DC algorithms to each decentralized power application scenario were also presented.

A design of group management, consensus, leader election and mutual exclusion DC building blocks were discussed in detail. The implementation of DCBlocks is done using Akka Java toolkit. One of the major drawback of Akka Java is that it only supports FIFO message ordering. If one needs ABCAST or total-ordered multicast, an additonal support feature needs to be developed on top of default Akka Java implementation. So, the implementation of DCBlocks does not support total ordering and this short coming is highlighted. The design handles crash, omission and timeout type of failures.

A handful of decentralized power application scenarios such as Decentralized State Estimation, Decentralized Voltage Stability, Decentralized Wind Power Monitoring and Control and Decentralized Frequency Response were picked as possible use cases for developing decentralized power applications using DCBlocks. An overview of the application behavior using various building blocks of DCBlocks such as group management, leader election, simple consensus etc. were described. This helped in visualizing the applicability of DCBlocks to a broad spectrum of decentralized power scenarios.

A prototype implementation of Decentralized Voltage Stability application using DCBlocks in collaboration with reserachers in SGDRIL, WSU were presented. The features of DCBlocks were demonstrated through this implementation. This implementation helped in highlighting that, some of the novel features such as dynamic regrouping of groups based on application specific condition can be easily accomplished using DCBlocks. Also, DCblocks provides efficient failure monitoring of all the nodes in the system. Hence, any node failure

can be easily detected. If the DCBlocks running on each node detects a primary group leader failure, it immediately notifies the higher application layer (DVS application logic) of the failure. After that, all the monitoring and control activities for the group is handled by the secondary leader. The behavior of DVS application for different power scenarios were demonstrated.

Experimental analysis of the implementation blocks of DCBlocks in DeterLab were conducted. The experiments were conducted to profile the behavior of DCBlocks under different test conditions such as changing group size and different failure types (crash and omission). The experiments were conducted for varying group size, i.e, $N = 10, 20...50$ nodes for three cases. In first case, there were no failures in the test setup. In the second case, crash failures were introduced and the behavior was tested. In the third case, the behavior in presence of omission failures was tested. For each group, the number of failures were increased continuously upto $F = (N-1)/2$. A synchronization message at the start of the algorithm execution was added to reduce the unwanted high wait periods between rounds. The implementation is non optimized. Further reduction in the timeout delay between consequent rounds is necessary. Also, an early decision to reach consensus or to abort the algorithm needs to be adopted to avoid executing the algorithm continuously if the number of failures observed in the system is above a predetermined threshold (optimum number below $F < (N-1)/2$).

7.2 Future Work

The goal of this thesis was to demonstrate the application of DC algorithms as a viable solution to build robust decentralized power applications. In this thesis, the concentration was in development of group management, leader election, consensus, ordered multicast and mutual exclusion blocks. Some of the future work activities are:

i. The focus should be on development of rest of the blocks for group discovery, voting etc.

ii. A new DC algorithm primitive named *Supply Agreement* algorithm can be developed where in each process in the group proposes a vector of values. Subsequently, all the processes collect the vectors proposed by other processes till all process agree on same set of vectors. For example, if there are N Distributed Energy Resources (DER) in a group, and each DER can provide how much it can support the grid in M time intervals, then this will lead to $N \times M$ matrix to be agreed upon by the communicating DERs.

iii. This thesis concentrated on the development of DCBlocks to demonstrate that DC concepts can be applied to decentralized power applications with realistic test setup. The design of DCBlocks can be optimized in several ways. Consensus and Leader Election blocks can be optimized to reduce the delay. A practical upper bound for the number of execution rounds in presence of high number of failures need to be estimated. An early decision (to reach consensus or abort) should be made instead of executing all the $F + 1$ rounds endlessly.

iv. For this thesis, the underlying distributed platform software selected was Akka Java.

While implementing DCBlocks, some of the drawbacks of Akka Java became evident. Some of them are Akka Java does not support ABCAST multicast delivery. It does not allow an actor to leave a cluster (group) and rejoin the cluster. To rejoin the cluster (group), the actor needs to be restarted. This may be OK if persistence feature is used. Else, it can become a costly operation. Hence, care should be taken when doing dynamic regrouping of nodes such that it does not lead to unpredictable or abnormal behavior.

An alternate distributed platform software/toolkit such as ISIS^2 having more robust features can be selected to implement DCBlocks. Some of the best design features discussed in this thesis can be used in the new implementation.

v. More robust testing of mutual exclusion block needs to be done.

vi. This design does not support Byzantine type of failures. Future work can focus on implementing the DC algorithms to handle Byzantine type of failures with the use of signed messages to authenticate the identity of senders.

vii. This design does not handle security attacks such as DDoS attacks, hijack of computers etc. Future work can focus on handling these type of security attacks.

viii. More decentralized power applications can be identified and developed using DCBlocks.

APPENDIX A

DCBLOCKS INTERFACES

DCBlocks consists of five building blocks namely, Group Management block, Leader Election block, Ordered Multicast block, Consensus/Agreement block and Mutual Exclusion block. Each block is designed with a generic public interface that the application logic can use. The interface and corresponding methods for each block is given below.

A.1 GroupMgmt Interface

```
package serviceInterfaces;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;
import java.util.function.Function;

/**
 * Interface for handling group management activities
 *
 * @author shwetha
 *
 */
public interface GroupMgmt {
  /**
   * Register with group management service to start group activities
   *
   * @param isStartUp
   *            Flag to indicate if it is called on start up or restart
   *
   * @return true if successful else false
   */
  Boolean registerToGroups(Boolean isStartUp);

  /**
   * Returns the group ID that the node belongs to.
   *
   * @param nodeId
   *            node ID
   * @return group ID
   */
  int getGroupId(int nodeId);
```

```
/**
 * Add node to the specified group
 *
 * @param groupId
 *            Group ID
 * @param nodeId
 *            node ID
 * @return true if successful else false
 */
Boolean addMember(int groupId, int nodeId);

/**
 * Remove node from the specified group
 *
 * @param groupId
 *            Group ID
 * @param nodeId
 *            node ID
 * @return true if successful else false
 */
Boolean removeMember(int groupId, int nodeId);

/**
 * Checks if node Id is member of specified group
 *
 * @param groupId
 *            Group ID
 * @param nodeId
 *            node ID
 * @return true if successful else false
 */
Boolean isMember(int groupId, int nodeId);

/**
 * Get list of current members in the group
 *
 * @param groupId
 * @return list of node Ids
 */
ArrayList<Integer> getMemberList(int groupId);

/**
 * Get list of node Ids and corresponding status for the specified group
 *
 * @param groupId
 *            group ID
 * @return Hashmap containing node IDs and corresponding status (true if
 *         healthy, false if faulty)
 */
HashMap<Integer, Boolean> getGroupHealthStatus(int groupId);

/**
 * Returns the status of specified node ID
 *
 * @param groupId
 *            group ID
 * @param nodeId
 *            node ID
```

```
     * @return status (true if healthy, false if faulty)
     */
    Boolean getMemberHealthStatus(int groupId, int nodeId);

    /**
     * Merge two or more groups together. The list of group members of specified
     * groups are moved to new group.
     *
     * @param groupIdToMerge
     *            Group Id to which other groups are merged.
     * @param groupIdList
     *            List of groups to merge to new group
     * @return true if merge request is successful else false
     */
    Boolean mergeGroups(Integer groupIdToMerge, ArrayList<Integer> groupIdList);

    /**
     * Split a group into 2 smaller groups. The list of members are moved to new
     * group
     *
     * @param oldGroupId
     *            Group ID that will be split into two.
     * @param newGroupId
     *            New Group ID
     * @param memberList
     *            List of node IDs that have to be moved to new group
     * @return true if split request is successful else false
     */
    Boolean splitGroups(Integer oldGroupId, Integer newGroupId,
        ArrayList<Integer> memberList);

    /**
     * Add list of Node IDs to the specified group
     *
     * @param groupId
     *            group ID
     * @param nodeIdList
     *            List of node IDs
     * @return true if successful, false if unsuccessful
     */
    Boolean addMemberList(int groupId, ArrayList<Integer> nodeIdList);

    /**
     * Remove list of Node IDs to the specified group
     *
     * @param groupId
     * @param nodeIdList
     * @return
     */
    Boolean removeMemberList(int groupId, ArrayList<Integer> nodeIdList);

    /**
     * Unregister from group management service
     *
     * @return
     */
    Boolean unregisterFromGroups();
```

```
/**
 * Returns number of groups in the system
 *
 * @return number of groups
 */
Integer getGroupCount();

/**
 * Get number of faulty node IDs for a specific group
 *
 * @param groupId
 *            group IDs
 * @return list of faulty node IDs
 */
ArrayList<Integer> getFailedList(int groupId);

/**
 * Get the primary leader for specified group
 *
 * @param groupId
 *            group ID
 * @return primary leader ID
 */
int getPrimaryLeader(int groupId);

/**
 * Set the primary leader for the specified group
 *
 * @param groupId
 *            group ID
 * @param leaderId
 *            primary leader ID
 */
void setPrimaryLeader(int groupId, int leaderId);

/**
 * Get the secondary leader for the specified group
 *
 * @param groupId
 *            group ID
 * @return secondary leader ID
 */
int getSecondaryLeader(int groupId);

/**
 * Set the secondary leader for the specified group
 *
 * @param groupId
 *            group ID
 * @param leaderId
 *            secondary leader ID
 */
void setSecondaryLeader(int groupId, int leaderId);

/**
 * Register to receive notification whenever the primary leader of specified
 * group fails
 *
```

```
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registerOnPriLeaderFailure(int groupId,
        Function<String, Boolean> callBackFunc);

    /**
     * Register to receive notification whenever the secondary leader of
     * specified group fails
     *
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registerOnSecLeaderFailure(int groupId,
        Function<String, Boolean> callBackFunc);

    /**
     * Register to receive notification whenever a group is ready to start
     * normal operation (i.e, group size has reached).
     *
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registorOnGroupReady(int groupId,
        Function<String, Boolean> callBackFunc);

    /**
     * Register to receive notification whenever a group is ready to start
     * normal operation (i.e, group size has reached).
     *
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registerOnMergeComplete(Function<MergeGroup, Boolean> callBackFunc);

    /**
     * Register to receive notification whenever a group is ready to start
     * normal operation (i.e, group size has reached).
     *
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registerOnSplitComplete(Function<SplitGroup, Boolean> callBackFunc);

    /**
     * Register to receive notification whenever a group is ready to start
     * normal operation (i.e, group size has reached).
     *
```

```
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registerOnAddComplete(Function<Integer, Boolean> callBackFunc);

    /**
     * Register to receive notification whenever all groups is ready to start
     * normal operation (i.e, group size has reached).
     *
     * @param groupId
     *            group ID
     * @param callBackFunc
     *            call back function to notify the application logic
     */
    void registorOnAllGroupsReady(int groupId,
        Function<String, Boolean> callBackFunc);

    /**
     * Subscribe to start receiving messages of specific topic type
     *
     * @param topic
     *            Subscription topic
     * @param ref
     *            Application reference to be notified when a message is
     *            received.
     * @return true if subscription is successful, else false
     */
    Boolean subscribeToOtherGroups(String topic, Object ref);

    /**
     * Unsubscribe to stop receiving messages of specific topic type
     *
     * @param topic
     *            Subscription topic
     * @param ref
     *            Application reference
     * @return true is un subscription is successful, else false
     */
    Boolean unsubscribeToOtherGroups(String topic, Object ref);

    /**
     * Publish messages to specific topic to particular group
     *
     * @param groupId
     *            destination group ID
     * @param topic
     *            message topic
     * @param msg
     *            message to publish
     * @return true if publish is successful, else false
     */
    Boolean publishMsgToOtherGroup(int groupId, String topic, Object msg);

    /**
     * Get the group Id list
     *
```

```
    * @return list of available group IDs.
    */
   Set<Integer> getGroupIds();
}
```

./appendixA/GroupMgmt.java

A.2 MergeGroup Interface

```
package serviceInterfaces;

/**
 * Interface containing information about ongoing merge activity
 * @author Shwetha
 *
 */

public interface MergeGroup {
   /**
    * Get Old group ID
    * @return group ID
    */
   public int getOldGroup();

   /**
    * Get ID of the group that other groups merged into.
    * @return new group ID
    */
   public int getNewGroup();

   /**
    * Get the number of nodes in the new merged group
    * @return Number of nodes
    */
   public int getNewGroupSize();

   /**
    * Get the node ID that initiated the merge activity
    * @return node ID
    */
   public int getStarterId();
}
```

./appendixA/MergeGroup.java

A.3 SplitGroup Interface

```
package serviceInterfaces;

import java.util.ArrayList;

/**
 * Interface containing information about ongoing split activity
 *
 * @author Shwetha
```

```
 *
 */
public interface SplitGroup {

  /**
   * Get Old group ID.
   * @return group ID
   */
  public int getOldGroup();

  /**
   * Get ID of the new group.
   * @return new group ID
   */
  public int getNewGroup();

  /**
   * Get the number of nodes in the old group after splitting.
   * @return Number of nodes
   */
  public int getOldGroupNewSize();

  /**
   * Get the number of nodes in the new group after split.
   * @return Number of nodes
   */
  public int getNewGroupSize();

  /**
   * Get the node ID that initiated the split activity
   * @return node ID
   */
  public int getStarterId();

  /**
   * Set node ID list to be moved to new group
   */
  public void setIdList(ArrayList<Integer> idList);

  /**
   * Get the node ID list to be moved to new group.
   *
   */
  public ArrayList<Integer> getIdList();
}
```

./appendixA/SplitGroup.java

A.4 GroupSubscriber Interface

```
package serviceInterfaces;

import java.util.function.Function;

/**
 * Interface to handle Subscribe/Unsubscribe methods to start/stop receiving
```

```
 * group messages.
 *
 * @author Shwetha
 *
 */
public interface GroupSubscriber {
  /**
   * This method subscribes to receive message for specified topic
   *
   * @param groupId
   *            subscribe to specified group
   * @param topic
   *            topic
   * @param recvCallBackFunc
   *            Call back function for received message
   * @return true if success else false
   */
  public Boolean Subscribe(Integer groupId, String topic,
      Function<Object, Boolean> recvCallBackFunc);

  /**
   * This method unsubscribes to receive message for specified topic
   *
   * @param groupId
   *            unsubscribe from specified group
   * @param topic
   *            topic
   * @return true if success else false
   */
  public Boolean UnSubscribe(Integer groupId, String topic);
}
```

./appendixA/GroupSubscriber.java

A.5 GroupPublisher Interface

```
package serviceInterfaces;

/**
 * Interface to publish messages to a group
 *
 * @author Shwetha
 *
 */
public interface GroupPublisher {

  /**
   * Publish the message to specified group
   *
   * @param groupId
   *            specified group
   * @param topic
   *            topic
   * @param msg
   *            message corresponding to topic
   * @return True if success, else false
```

```
    */
   public Boolean PublishMsg(Integer groupId, String topic, Object msg);
}
```

./appendixA/GroupPublisher.java

A.6 Consensus Interface

```
package serviceInterfaces;

import java.util.function.Function;

/**
 * Consensus Interface to handle Simple Consensus and Interactive Consistency
 * Agreement (ICA) algorithms.
 *
 * @author Shwetha
 *
 */
public interface Consensus {

   /**
    * Starts Simple Consensus (Non blocking)
    *
    * @param proposedValue
    *            local value
    * @param callBackFunc
    *            function to call after consensus/agreement has reached
    * @return True if start success
    */
   public Boolean startNonBlockingSimpleConsensus(Integer proposedValue,
       Function<Double, Boolean> callBackFunc,
       Function<Integer[], Double> decisionFunc);

   /**
    * Starts Simple Consensus (blocking)
    *
    * @param proposedValue
    *            local value
    * @return Consensus/agreement result
    */
   public Double startBlockingSimpleConsensus(Integer proposedValue,
       Function<Integer[], Double> decisionFunc);

   /**
    * Starts ICA (Non blocking)
    *
    * @param proposedValue
    *            local value
    * @param callBackFunc
    *            function to call after consensus/agreement vector has reached
    * @return True if start success
    */
   public Boolean startNonBlockingICConsensus(Integer proposedValue,
       Function<Integer[], Boolean> callBackFunc);
```

```
  /**
   * Starts ICA (Blocking)
   *
   * @param proposedValue
   *            local value
   * @return Consensus/agreement vector
   */
  public Integer[] startBlockingICConsensus(Integer proposedValue);

  /**
   * Stops Simple Consensus if running
   *
   * @return True if success Else false
   */
  Boolean stopSimpleConsensus();// For future: Can be supported for security
                  // aspects

  /**
   * Stops ICA if running
   *
   * @return True if success Else false
   */
  Boolean stopICConsensus();// For future: Can be supported for security
                // aspects

  /**
   * Checks if Simple Consensus is in progress
   *
   * @return True if running Else false
   */
  Boolean isSimpleConsensusInProgress();

  /**
   * Checks if ICA is in progress
   *
   * @return True if running Else false
   */
  Boolean isICConsensusInProgress();
}
```

./appendixA/Consensus.java

A.7 LeaderElection Interface

```
package serviceInterfaces;

import java.util.function.Function;

/**
 * Public interface for Leader Election (Based on IC Consensus)
 *
 * @author Shwetha
 *
 */
public interface LeaderElection {
  /**
```

```
     * NOTE: First function to be called by each member participating in Leader
     * election. It stores the election vote and call back decsion function to
     * be used when the actual election is triggered.
     *
     * @param myVote
     *            my vote
     * @param decisionFunc
     *            App defined decision function
     * @return True if success else failure
     */
    Boolean setUpForLeaderElection(Integer myVote,
        Function<Integer[], Integer[]> decisionFunc);

    /**
     * Starts the leader election by start request to Leader Election Executor
     * actor. Waits for the election result.
     *
     * @return Elected leader Id
     * @throws Exception
     */
    Integer startElection() throws Exception;

    /**
     * Gets the primary leader Id
     *
     * @return Primary leader Id
     */
    int getPrimaryLeader();

    /**
     * Gets the secondary leader Id
     *
     * @return Secondary leader Id
     */
    int getSecondaryLeader();

    /**
     * Checks if specified nodeId is primary leader
     *
     * @param nodeId
     * @return True if primary leader, else false
     */
    Boolean isPrimaryLeader(int nodeId);

    /**
     * Checks if specified nodeId is secondary leader
     *
     * @param nodeId
     * @return True if primary leader, else false
     */
    Boolean isSecondaryLeader(int nodeId);

    /**
     * Stops the leader election if running
     *
     * @return True if success else failure
     */
    Boolean stopElection();
```

```
  /**
   * Removes all the dependencies for leader election
   *
   * @return True if success else failure
   */
  Boolean removeSetUpForLeaderElection();
}
```

./appendixA/LeaderElection.java

A.8 MutualExclusion Interface

```
package serviceInterfaces;

import java.util.function.Function;

/**
 * Interface to handle mutual exclusion activity
 *
 * @author Shwetha
 *
 */
public interface MutualExclusion {
  /**
   * Request to acquire exclusive access to a shared resource.
   *
   * @param resourceId
   *            shared resource ID
   * @param durationInSec
   *            duration of access in seconds
   * @param callBackFunc
   *            call back function to notify when shared resource is available
   *            message corresponding to topic
   * @return True if request is successful, else false
   */
  Boolean acquireResource(int resourceId, long durationInSec,
      Function<Integer, Boolean> callBackFunc);

  /**
   * Release access of shared resource.
   *
   * @param resourceId
   *            shared resource ID
   * @return True if release is successful, else false
   */
  Boolean releaseResource(int resourceId);
}
```

./appendixA/MutualExclusion.java

APPENDIX B

LIST OF PUBLICATIONS

Banerjee, P., S. Niddodi, H. Lee, A. Srivastava, and D. Bakken, 2015: On the need for robust decentralized coordination to support emerging decentralized monitoring and control applications in electric power grid. *Proceedings of the Fourth Grid of the Future Symposium, CIGRE*, Chicago, USA, 1–9.

Lee, H., S. Niddodi, A. Srivastava, and D. Bakken, 2015: Distributed computing architecture for a decentralized voltage stability application. *Smart Grid, IEEE Transactions on*, To Appear.

Liu, R., S. Niddodi, A. Srivastava, and D. Bakken, 2016: Distributed computing based decentralized state estimation. *Power System Computation Conference*, In Review.

Niddodi, S. and D. Bakken, 2016: Dcblocks: A coordination platform for decentralized cyber-physical applications. *Software: Practice and Experience*, Wiley Subscription Services, Inc., In Preparation.

BIBLIOGRAPHY

Akka, 2015: Akka documentation. [Akka 2.3.14 (current stable release) for Scala 2.10 / 2.11 and Java 6+].
URL `http://akka.io/docs/`

Banerjee, P., S. Niddodi, H. Lee, A. Srivastava, and D. Bakken, 2015: On the need for robust decentralized coordination to support emerging decentralized monitoring and control applications in electric power grid. *Proceedings of the Fourth Grid of the Future Symposium, CIGRE*, Chicago, USA, 1–9.

Birman, K. and R. Cooper, 1991: The isis project: Real experience with a fault tolerant programming system. *SIGOPS Oper. Syst. Rev.*, **25**, 103–107, doi:10.1145/122120.122133.
URL `http://doi.acm.org/10.1145/122120.122133`

Birman, K. and T. Joseph, 1987: Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, **21**, 123–138, doi:10.1145/37499.37515.
URL `http://doi.acm.org/10.1145/37499.37515`

Birman, K., A. Schiper, and P. Stephenson, 1991: Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, **9**, 272–314, doi:10.1145/128738.128742.
URL `http://doi.acm.org/10.1145/128738.128742`

Biskas, P., A. Bakirtzis, N. Macheras, and N. Pasialis, 2005: A decentralized implementation of dc optimal power flow on a network of computers. *Power Systems, IEEE Transactions on*, **20**, 25–33, doi:10.1109/TPWRS.2004.831283.

Chaudhuri, S., M. Herlihy, N. A. Lynch, and M. R. Tuttle, 1993: A tight lower bound for k-set agreement. *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, 206–215.

DETERLab, 2015: Deter project website.
URL `http://deter-project.org/`

Elkhatib, M., R. Shatshat, and M. Salama, 2012: Decentralized reactive power control for advanced distribution automation systems. *Smart Grid, IEEE Transactions on*, **3**, 1482–1490, doi:10.1109/TSG.2012.2197833.

G. Coulouris, T. K., J. Dollimore and G. Blair., 2011: Distributed systems: Concepts and design, 5ed. *Boston: Addison-Wesley*.

Garcia-Molina, H., 1982: Elections in a distributed computing system. *Computers, IEEE Transactions on*, **C-31**, 48–59, doi:10.1109/TC.1982.1675885.

Gascn, A. and A. Tiwari, 2014: A synthesized algorithm for interactive consistency. *NASA Formal Methods*, J. Badger and K. Rozier, eds., Springer International Publishing, volume 8430 of *Lecture Notes in Computer Science*, 270–284.
URL http://dx.doi.org/10.1007/978-3-319-06200-6_23

Goodfellow, R., R. Braden, T. Benzel, and D. E. Bakken, 2013: First steps toward scientific cyber-security experimentation in wide-area cyber-physical systems. *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ACM, New York, NY, USA, CSIIRW '13, 39:1–39:4.
URL http://doi.acm.org/10.1145/2459976.2460021

Hong, X. and M. Gerla, 2002: Dynamic group discovery and routing in ad hoc networks. *Proceedings of the First Annual Mediterranean Ad Hoc Networking Workshop*.

Hug-Glanzmann, G. and G. Andersson, 2009: Decentralized optimal power flow control for overlapping areas in power systems. *Power Systems, IEEE Transactions on*, **24**, 327–336, doi:10.1109/TPWRS.2008.2006998.

Hupfeld, F., B. Kolbeck, J. Stender, M. Högqvist, T. Cortes, J. Marti, and J. Malo, 2008: Fatlease: Scalable fault-tolerant lease negotiation with paxos. *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, ACM, New York, NY, USA, HPDC '08, 1–10.
URL http://doi.acm.org/10.1145/1383422.1383424

Isis2, 2015: Isis2 cloud computing library. [V 2.2.2013 - latest general release].
URL http://isis2.codeplex.com/

JBoss, 2015: Jbossdeveloper.
URL http://www.jboss.org/

Korres, G., 2011: A distributed multiarea state estimation. *Power Systems, IEEE Transactions on*, **26**, 73–84, doi:10.1109/TPWRS.2010.2047030.

Lamport, L., 1978: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, **21**, 558–565, doi:10.1145/359545.359563.
URL http://doi.acm.org/10.1145/359545.359563

— 2005: Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=64631

Lee, H., S. Niddodi, A. Srivastava, and D. Bakken, 2015: Distributed computing architecture for a decentralized voltage stability application. *Smart Grid, IEEE Transactions on*.

Li, Y., 2012: Fully distributed state estimation of smart grids. *Communications (ICC), 2012 IEEE International Conference on*, 6580–6585.

Liang, H., B. J. Choi, W. Zhuang, and X. Shen, 2013: Stability enhancement of decentralized inverter control through wireless communications in microgrids. *Smart Grid, IEEE Transactions on*, **4**, 321–331, doi:10.1109/TSG.2012.2226064.

Liu, R., S. Niddodi, A. Srivastava, and D. Bakken, 2016: Distributed computing based decentralized state estimation. *Power System Computation Conference*.

Lynch, N., 1996: *Distributed Algorithms*. Morgan Kaufmann, 1 edition.

Maekawa, M., 1985: A n algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, **3**, 145–159, doi:10.1145/214438.214445.
URL `http://doi.acm.org/10.1145/214438.214445`

Malpani, N., J. L. Welch, and N. Vaidya, 2000: Leader election algorithms for mobile ad hoc networks. *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, ACM, New York, NY, USA, DIALM '00, 96–103.
URL `http://doi.acm.org/10.1145/345848.345871`

Menascé, D. A. and T. Nakanishi, 1982: Performance evaluation of a two-phase commit based protocol for ddbs. *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ACM, New York, NY, USA, PODS '82, 247–255.
URL `http://doi.acm.org/10.1145/588111.588152`

Moghadam, M., R. Zhang, and R. Ma, 2013: Randomized response electric vehicles for distributed frequency control in smart grid. *Smart Grid Communications (SmartGridComm), 2013 IEEE International Conference on*, 139–144.

Moradzadeh, M., R. Boel, and L. Vandevelde, 2013: Voltage coordination in multi-area power systems via distributed model predictive control. *Power Systems, IEEE Transactions on*, **28**, 513–521, doi:10.1109/TPWRS.2012.2197028.

Nazari, M., Z. Costello, M. Feizollahi, S. Grijalva, and M. Egerstedt, 2014: Distributed frequency control of prosumer-based electric energy systems. *Power Systems, IEEE Transactions on*, **29**, 2934–2942, doi:10.1109/TPWRS.2014.2310176.

of Washington, U., 1993: Power svstems test case archive.
URL `http://www.ee.washington.edulresearch/pstca`

Ongaro, D. and J. Ousterhout, 2014: In search of an understandable consensus algorithm. *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, USENIX ATC'14, 305–320.
URL `http://dl.acm.org/citation.cfm?id=2643634.2643666`

Pease, M., R. Shostak, and L. Lamport, 1980: Reaching agreement in the presence of faults. *J. ACM*, **27**, 228–234, doi:10.1145/322186.322188.
URL `http://doi.acm.org/10.1145/322186.322188`

Rajsbaum, S., 2001: Acm sigact news distributed computing column 5. *SIGACT News*, **32**, 34–58, doi:10.1145/568425.568433.
URL `http://doi.acm.org/10.1145/568425.568433`

Raynal, M., 2010: *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Morgan and Claypool Publishers, USA, 1st edition.

Report, F. E. R. C., 2012: Arizona-southern california outages on september 8, 2011. Technical report, FERC.

Ricart, G. and A. K. Agrawala, 1981: An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, **24**, 9–17, doi:10.1145/358527.358537.
URL `http://doi.acm.org/10.1145/358527.358537`

Skeen, D. and M. Stonebraker, 1983: A formal model of crash recovery in a distributed system. *Software Engineering, IEEE Transactions on*, **SE-9**, 219–228, doi:10.1109/TSE.1983.236608.

Suzuki, I. and T. Kasami, 1985: A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, **3**, 344–349, doi:10.1145/6110.214406.
URL `http://doi.acm.org/10.1145/6110.214406`

Van Cutsem, T., J. Horward, and M. Ribbens-Pavella, 1981: A two-level static state estimator for electric power systems. *Power Apparatus and Systems, IEEE Transactions on*, **PAS-100**, 3722–3732, doi:10.1109/TPAS.1981.317015.

Vasudevan, S., J. Kurose, and D. Towsley, 2004: Design and analysis of a leader election algorithm for mobile ad hoc networks. *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, 350–360.

Wu, F., K. Moslehi, and A. Bose, 2005: Power system control centers: Past, present, and future. *Proceedings of the IEEE*, **93**, 1890–1908, doi:10.1109/JPROC.2005.857499.

Xie, L., D.-H. Choi, S. Kar, and H. Poor, 2012: Fully distributed state estimation for wide-area monitoring systems. *Smart Grid, IEEE Transactions on*, **3**, 1154–1169, doi:10.1109/TSG.2012.2197764.

Yip, T., C. An, G. Lloyd, M. Aten, and B. Ferri, 2009: Dynamic line rating protection for wind farm connections. *Integration of Wide-Scale Renewable Resources Into the Power Delivery System, 2009 CIGRE/IEEE PES Joint Symposium*, 1–5.