MY BIG FAT THESIS

by

STEPHEN CURTIS JACKSON

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2016

Approved

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

ABSTRACT

ACKNOWLEDGMENTS

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

1. RELATED WORK

1.1. FAILURE CLASSIFICATION

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

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

The fail-stop failure has three properties **?**:

1. When a failure occurs the program execution is stopped.
2. A process can detect when another fail-stop process has failed.
3. Volatile storage is lost when the fail-stop process is stopped.

1.1.2. Omission Failure. An omission failure occurs when a message is never received by the receiver. Omission failures can occur when the communication medium is unavailable or when the latency of message exceeds a timeout for its expected delivery. Protocols like TCP do not tolerate omission failures: a packet is

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

1.2. VIRTUAL SYNCHRONY

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

Virtual synchrony is a model of communication and execution that allows the system designer to emulate synchronous execution. Although the processes do not execute tasks simultaneously, the execution history of each process cannot be differentiated from a trace where tasks are executed simultaneously.

Consider a distributed system where events can be causally related based on their execution on the local processors and communication between processes, as defined in (**?**, p .101).

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

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

of histories H and H' are equivalent if, the history of a process p, $H|_p$, cannot be differentiated from another history, $H'|_p$, based on the casual relationships between the events in the history. (**?**, p .103) Additionally, a history is considered complete if all sent messages are delivered and there are no casual holes. A casual hole is a circumstance where an event e is casually related to e' by $e' \rightarrow e$ and e appears in a history, but e' does not. A virtually synchronous system is one where each history in the system is indistinguishable from all histories produced by a synchronous system. (**?**, p .104)

1.2.1. Process Groups. Virtual synchrony models also support process groups. Although each implementation of a virtually synchronous system applies a different structure to the way processes are grouped, implementations share common features. A process group in a virtual synchronous system is commonly referred to as a view. A view is a collection of processes that are virtually synchronous with each other. A view refers to a specific group: the set of processes in that view, and a leader if there is one. A group is a general descriptor for a collection of processes. Process groups in virtually synchronous systems place obligations on the delivery of messages to members of a view. A history H of a view is legal if (**?**, p .103):

1. Given a function $time(e)$ that returns a global time of when the event occurred e, then $e \rightarrow e' \Rightarrow time(e) < time(e')$.

2. $time(e) \neq time(e') \forall e, e' \in H|_p$ (where $e \neq e'$) for each process.

3. A change in view (group membership) occurs at the same logical time for all processes in the view.

4. All multicast message deliveries occur in the current view of a group. That is, if a message is sent in a view, it is delivered in that view, for every process in that view.

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

- The system employs a membership service. This service monitors for failures and reports them to the other processes in the view.
- When a process is identified as failing it is removed from the groups that it belongs to and the remaining processes determine a new view.
- After a process has been identified as failing, no message will be received from it.

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

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

1.3. EXTENDED VIRTUAL SYNCHRONY

One of the major shortcomings of the original virtual synchrony model lays in how it handles network partitions. In virtual synchrony, when the network is partitioned, only processes in the primary partition were allowed to continue. Processes that were not in the primary partition could not rejoin the primary partition without being restarted: the process joining the view must be a new process so that there are no message delivery obligations for that process in the view. **?** presented an improved version of virtual synchrony, dubbed extended virtual synchrony. Extended virtual synchrony is compatible with the original virtual synchrony, and can implement all the functionality and limitations of the original design. Because it supports virtual synchrony, extended virtual synchrony has become the basis for a number of related frameworks that have been designed since Isis including Horus, Totem, Transis, and Spread.

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

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

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

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

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

a message delivery which occurs after the creation of a view must occur after the view is installed by every process in the view.

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

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

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

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

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

It also provides additional restrictions on the sending of messages between two different views (Item 2) and an obligation on the delivery of messages (Item 4). The concept of a primary view still exists within the extended virtual synchrony model and is still described by the notion of being the largest view. Since views can now partition and rejoin the primary partition the rules presented above also allow the history of the views with respect to the regular messages in the primary partition to be totally ordered. Additionally, two consecutive primary views have at least one process that was a member of each view.

To join views, processes are first informed of a failure (or joinable partition) by a membership service. Processes maintain an obligation set of messages they have

acknowledged but not yet delivered. After being informed of the need to change views, the processes begin buffering received messages and transition into a transitional view. In the transitional view messages are sent and delivered by the processes to fulfill the causality and ordering requirements listed above. Once all messages have be transmitted, received and delivered as needed and the process' obligation set is empty, the view transitions from a transitional view to a regular view and execution continues as normal.

1.3.1. Comparison To DGI. The DGI places similar but distinct requirements on execution of processes in its system. The DGI enforces synchronization between processes that obligates each active process to enter each module's phase simultaneously. This is fulfilled by using a clock synchronization algorithm. The virtual synchrony model, at its most basic, does not require a clock to enforce its ordering.

However, this simplifies the DGI fulfilling its real time requirements. Processes must be able to react to changes in the power system within a maximum amount of time. These interactions do not require interaction between all processes within the group. Additionally, this allows for private transactions to occur between DGI processes.

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

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

1.4. ISIS (1989) AND HORUS (1996)

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

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

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

1.4.1. Group Model. In Horus and Isis, a group is a collection of processes that can communicate with one another to do work. Multicasts are directed to the view, and are guaranteed to be received by all members or no members. Over time, due to failure, the view will change. Since views are distributed concurrently and asynchronously, each process can have a different view. Horus is designed to have a modular communication structure composed of layers, which allows the communication channel to have different properties that will affect which messages will be delivered in the event of a view change. Horus' layers allow developers to go as far to not use the complete virtual synchrony model, if the programmer desires. For

example, Horus' modules can implement total order using a token passing layer, or casual ordering using vector clocks.

Isis has limited support for partitioning. In the event that a partition forms, dividing the large group into subgroups, only the primary group is allowed to continue operating. The primary group is selected by choosing the largest partition. Horus, on the other hand implements the extended virtual synchrony model and does not have that limitation.

1.5. TRANSIS (1992)

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

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

1.6. TOTEM (1996)

Totem[?] is designed to use local area networks, connected by gateways. The local area networks use a token passing ring and multicast the messages, much like Isis and Horus. The gateways join these rings into a multi-ring topology. Messages are first delivered on the ring which they are sent, then forwarded by the gateway which connects the local ring to the wide area ring for delivery to the other rings. The message protocol gives the message total ordering using a token passing protocol. Topology changes are handled in the local ring, then forwarded through the gateway where the system determines if the local change necessitates a change in the wide area ring. Failure detection is also implemented to detect failed gateways.

1.7. SPREAD

Spread[?] is a modern distributed framework. It is designed as a series of daemons which communicate over a wide area network. Processes on the same LAN as the daemons connect to the daemon directly. This is analogous to the gateways in the other distributed toolkits, however, the daemon is a special, dedicated message passing process, rather than a participating process with a special role.

Using a dedicated daemon allows a Spread view to reconfigure less frequently when a process stops responding (which normally correlates to a view change) since the daemons can insert a message informing other processes that a process has left while still maintaining the message ordering without disruption. Major reconfigurations only need to occur when a daemon is suspected of failing.

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

1.8. FAILURE DETECTORS

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

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

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

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

VITA