DETECTING FAILURES IN AN ASYNCHRONOUS SYSTEM THAT NEVER STOPS CHANGING

An Undergraduate Research Scholars Thesis

by

HIMANK YADAV

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Jennifer Welch

May 2018

Major: Computer Science

TABLE OF CONTENTS

Page

ABSTRACT

Detecting Failures in an Asynchronous System That Never Stops Changing

Himank Yadav
Department of Computer Science
Texas A&M University

Research Advisor: Dr. Jennifer Welch
Department of Computer Science
Texas A&M University

This thesis presents an algorithm for detecting failures in dynamic asynchronous distributed systems or environments in which new participants may continually join the system and old participants may continually leave the system (a phenomenon called churn), and active participants may fail.

Such behavior is exhibited by many dynamic modern networks, for example, peer-to-peer networks. Devices are continually joining and leaving, and peers often remain in the network only long enough to retrieve the data they require. Another example would be mobile networks. Devices are constantly on the move, resulting in a continual change in participants. In these types of networks, the set of participants is rarely stable for very long and is dynamically changing.

Many problems are not solvable if the fraction of participants that are crashed is too large. Yet the participants will continue to leave the system or crash. To avoid crossing the threshold where too many participants are crashed, it is of paramount importance to detect crashed participants and remove them from the system.

The problem of detecting failures has been solved in static and synchronous distributed systems. However, since processes in an asynchronous dynamic distributed systems possess no global clock or synchronized logical clocks or timing information, detecting failures is a hard problem to solve in such systems.

We propose a failure detector for an asynchronous system with churn by exploiting the dynamic nature of the system to estimate elapsed time. We design an algorithm to detect failed processes in such an asynchronous system and prove that if a process is identified as crashed by our failure detector, it has indeed failed. Additionally, we also prove that if the churn continues forever, then under certain circumstances every failed process is eventually identified as crashed.

DEDICATION

To those who keep pushing the boundaries of computing forward.

ACKNOWLEDGMENTS

I cannot thank enough my advisor, Dr. Jennifer Welch, for her unending support in making this thesis possible, and for introducing me to this amazing world of research and distributed computing. I would not have been here in the first place if it were not for Dr. Welch. She introduced me to interesting problems in the field of distributed algorithms and has taught me valuable lessons. She has provided insightful comments throughout the course of my research, important connections to past work, and unbounded encouragement to pursue new research directions. Her door has always been open for me. I am constantly amazed by her attention to detail and her commitment to research and mentoring. She has served as an immense source of support and inspiration, and I consider myself both extremely lucky and privileged to have had a chance to work with her.

I am also extremely grateful to Saptaparni Kumar for her advice, mentoring, friendship, and all the long discussions. The guidance and support I have received from Sapta throughout the course of my research have been absolutely paramount in completing this work. Her constant source of energy and enthusiasm towards new research ideas has played an important role in keeping me excited about exploring these ideas, and, therefore, towards establishing the direction of this thesis.

Lastly, a big thank you to Juliang, Victoria, Muin, Sahil, Tyler, Neal, Jay, and Prof. Tyagi for being my support pillars and the best friends anyone could ever ask for to shape their undergraduate experience. There are no other people I would rather spend so much time with, and I cannot even fathom having such an incredible time during my undergraduate years without them. Thanks also go out to all my other friends and the department faculty who have contributed heavily in making my time at Texas A&M University a wonderful journey.

NOMENCLATURE

α	Churn fraction
Δ	Failure fraction
D	Upper bound on message delay
$N(t)$	Number of nodes present in the system at time t
$CCREG$	Continuous Churn Register

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Shared memory is a popular mechanism to communicate in concurrent systems and a long-studied primitive for distributed algorithms, allowing each device to store and retrieve information. An advantage of shared memory algorithms is their simplicity and their more high-level nature than messaging passing algorithms. However, in modern large-scale systems, physical shared memory is not a viable option. Instead, shared memory objects are often simulated in a messaging passing distributed system.

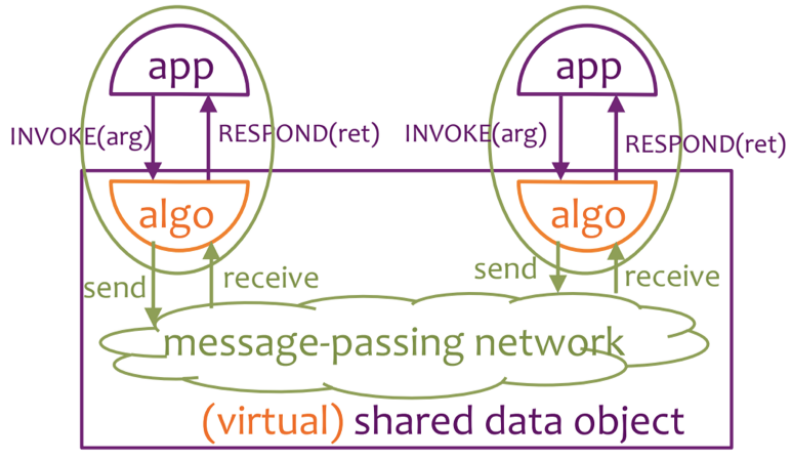

Figure 1.1: Overview of a virtual shared object [1]

Figure 1.1 illustrates how external applications invoke operations on a shared data object and receive responses. The shared data object is simulated by different processes in a distributed system. Individual processes keep independent copies of the shared object and use message-passing amongst processes to keep their copies consistent, thus providing an illusion of a shared object to external applications while hiding all the complexity under the hood.

Most existing work has focused on simulating atomic shared read/write registers. Many simulations duplicate the value of the register in various servers and require readers and writers to communicate with a majority of servers. One such example is the ABD simulation [2] which assumes a majority of the processes do not crash and replicates the value of the register in the server processes. For a single writer and reader, the writer sends the value along with a sequence number and waits for a confirmation from a majority of them. The reader contacts a majority of the server processes for the value and returns the one with the highest sequence number. This approach can be further extended to multiple readers and writers by having both a read phase and a write phase take place during reads and writes [3]. This approach is proven to work well for static systems where the number of readers, writers and servers are predetermined or fixed. And it has motivated research for dynamic systems where the processes may enter and leave anytime.

The original work on distributed systems with churn (in which processes may enter or leave dynamically) relies on the assumption that the system size is bounded [4] or the churn eventually stops for sufficiently long periods of time [5]. A recent algorithm [6] lets the churn continue forever while still ensuring that read and write operations continue and processes can join and leave the system anytime. The churn model in [6] assumes an upper bound on the number of processes that can enter or leave the system during a certain time interval which is derived from the size of the system. This allows the simulation of a read/write register in a crash-prone asynchronous system where processes enter and leave continuously as long as they satisfy the churn model constraints.

The system is crash-prone and therefore processes possibly fail as the system continues to exist. The original CCREG algorithm [6] does not detect failures thereby decreasing the robustness of the system.

1.1 Motivation

Our motivation to tackle this problem is threefold.

1. Since we are dealing with a dynamic system with churn, we would like our failures to be dynamic as well, and would like the system to detect failures instead of relying on some entity outside the system.

2. Not being able to detect failures means that the system can no longer afford to have processes that fail unless new processes are added in order to maintain an upper bound on the ratio of failed processes with respect to the system size.

3. Undetected failed processes would also reduce the dynamic nature of the system since existing processes in the system will not be allowed to leave as that could violate the upper bound on the ratio of failed processes with respect to the system size.

4. If we are able to implement a failure detector, we can explore other problems like solving consensus (a primitive in distributed computing that ensures all processes agree on a common value) which are unsolvable in the presence of undetected failures.

1.2 Failure Detectors

Failure detectors were first proposed as way to solve consensus in asynchronous distributed systems with crashes by differentiating between slow and crashed processes [7]. Each process can run the failure detector module and use the information concerning which processes the failure detector suspects have crashed to help it solve consensus.

Failure detectors are classified in [7] based on when their suspicions are correct. The main tenets of such classification include accuracy and completeness. The accuracy con-

dition states that if a process is suspected as crashed, then it has really crashed. The completeness condition states if a process is crashed, then it is suspected as crashed.

Certain failure detectors are strong enough (i.e., give sufficiently accurate and complete information) to allow consensus to be solved. Since consensus is unsolvable in our model, we cannot implement such failure detectors in our model in general.

However, under certain assumptions on the churn, we are able to implement a fairly strong failure detector in our model. We prioritize accuracy over completeness and aim to identify failed processes with perfect accuracy (all processes identified by our failure detector have actually crashed).

1.3 Contribution

This thesis augments the CCREG algorithm [6] for dynamic asynchronous message passing systems subject to crash failures by adding the ability to detect failures. Detecting failures enables us to increase the fault tolerance of the algorithm given the churn model. Once failed processes are detected, an external entity could take steps to remove such processes. We prove the correctness of our failure detector, i.e., every process that has been identified as crashed as indeed crashed, and show that under some circumstances, all failures can be detected if the churn continues forever.

2. MODEL

The model is the same as that adopted by the CCREG algorithm [6]. The system is an asynchronous message-passing system where processes have no idea of real or elapsed time due to the lack of clocks.

Processes use a broadcasting service to communicate with each other within the system. The broadcast service sends the same message to all processes in the system where all broadcasted messages have an upper bound on message delay, D, which is unknown to processes in the system. This implies that any message sent by process *p* at time *t* is guaranteed to be received by process *q* within D units of time provided process q is active throughout $[t, t + D]$.

The system has a churn fraction, α, known to all nodes, i.e., an upper bound on the churn that occurs in the time interval $[t, t + D]$. For any time t, the number of "*ENTER*(p)" or "*LEAVE*(p)" signals occurring in the time interval $[t, t + D]$ is at most $\alpha \cdot N(t)$ where $N(t)$ is the number of processes present at time t.

There is also a failure fraction, $\Delta < 1$, known to all processes, such that at any given time t, at most $\Delta \cdot N(t)$ processes have crashed. Note that no active processes can leave the system if $\Delta \cdot N(t)$ processes are crashed at time t.

An "*ENTER*(p)" signal experienced by process *p* causes *p* to enter the system and similarly a "*LEAVE*(p)" signal causes p to leave the system. These signals for any given process *p* can be generated at most once implying that processes cannot re-enter the system after leaving.

As seen in Figure 2.1 below, each process runs *failure_detector* threads along with the regular *client* and *server* threads. Processes can enter and leave the system as long as they satisfy the churn constraints.

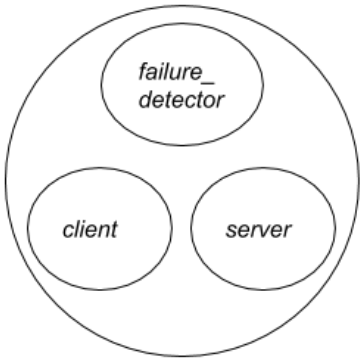

Figure 2.1: A sample process p running the three threads

We assume the joining protocol from [6] is executed. Accordingly, after entering the system, the process announces its entry to all processes by broadcasting an $enter$ message. After the process announces its entry to all processes, it waits to receive sufficient acknowledgement messages before announcing (by broadcasting a message to all processes) that it has joined the system. Joining the system is separate from entering the system. A joined process has a good estimate of the system composition and is able to perform read, write and failure detection operations.

A process is $present$ at time *t* if it has entered the system but not left by time *t*. The *Present* set for a process *p* is a local variable at p containing *p*'s estimate of all processes that are present in the system at the given time. The *Present* set is maintained according to the protocol presented in [6].

Processes that have crashed do not participate in the system in any way, i.e., crashed processes do not send or receive any messages. The terms crashes and failures are analogous and refer to the same thing.

Similar to the entering protocol, a process announces its leave to all processes by broadcasting a $leave$ message when it leaves the system [6]. The main difference between

a leave and a crash is that leaves are announced while crashes are not.

An $active$ process at time t is present in the system at time t and has not crashed.

3. METHOD

Our approach towards building a failure detector for the system is dependent on each active process running the failure detection module continuously on the failure detector thread. The failure detection module consists of a series of phases, defined below.

Since crashed processes do not interact with the system in any manner, i.e., they do not send or receive messages, we take advantage of this property of our model. Processes send failure-check messages to all other processes and wait for the acknowledgements from processes to come back. If a sender process does not receive acknowledgements back from a receiver process within a specified time period and the process has not left the system, the receiver process is marked as failed. However, the hard problem is that since this is an asynchronous system, processes do have any measure of time. Therefore, it is hard to estimate elapsed time. We approach this by exploiting the churn rate to gain an estimation on elapsed time.

Each failure detection phase for a process p begins by process p broadcasting a *fail-check* message to all processes in the system at time t. Each message has a maximum transmission delay of D. Therefore, the receiving active process q should receive the message by at most $t + D$ time and respond with an acknowledgement. The acknowledgement send by process q should be received by process p by at most time $t + 2D$ since the upper bound on message delay for the acknowledgement is also D. When the phase ends, which is guaranteed to be after at least 2D time has elapsed, process p marks all the other processes (that have not left the system) it has not heard from as failed.

The failure detection phases are run by all active joined processes repeatedly. Each phase consists of running the failure detection algorithm until at least $2D$ units of time has elapsed since the process running the failure detection algorithm broadcasted the *fail-check*

message most recently.

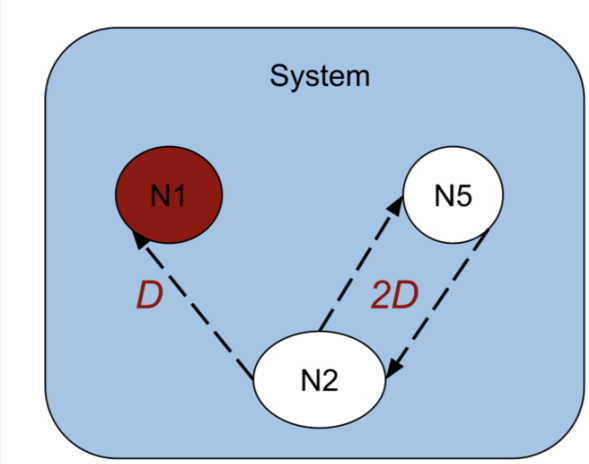

Figure 3.1: Sample failure detection by process $N2$

As seen in the example in Figure 3.1, process $N2$ starts a failure detection phase by sending a *fail-check* message to processes $N1$ and $N5$ at time t. The maximum message transmission delay on these messages is D, therefore, these messages must be received by processes $N1$ and $N5$ by time $t + D$. Since process $N1$ is crashed, it does not receive the message sent by process p and cannot reply to it. On the other hand, an active process $N5$ receives the message within time $t + D$ and replies with an acknowledgement that reaches process $N2$ by time $t + 2D$. At the end of the failure detection phase (which is after $t + 2D$), since process $N2$ has received the acknowledgement from process $N5$ but not from process $N1$, process $N2$ marks process $N1$ as failed.

The challenge here is for $N2$ to determine when the time interval $2D$ has elapsed as the processes have no way to measure time. In order to determine time, we use the churn rate.

The churn bound, α, signifies that at most $\alpha \cdot N(t)$ processes can enter or leave the

system in a given time interval, $[t, t + D]$. In other words, the maximum number of churn events that can take place in the time interval, $[t, t + D]$, is $\alpha \cdot N(t)$.

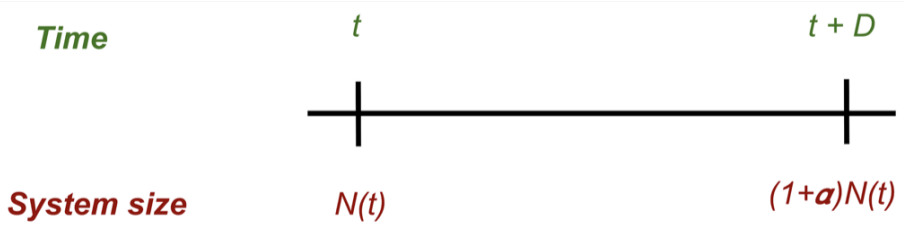

Figure 3.2: Maximum increase in system size during $[t, t + D]$

Figure 3.2 shows the maximum possible increase in system size as time elapses. The system size at any time t is $N(t)$ and since at most $\alpha \cdot N(t)$ processes can enter the system during the time interval $[t, t + D]$, the maximum possible system size at time $t + D$ is $(1 + \alpha) \cdot N(t)$.

As we mentioned above, a process needs to wait for at least $2D$ time after sending the *fail-check* message to detect failures since it takes at most D units of time to reach the other process and another D units of time for the acknowledgement to reach back.

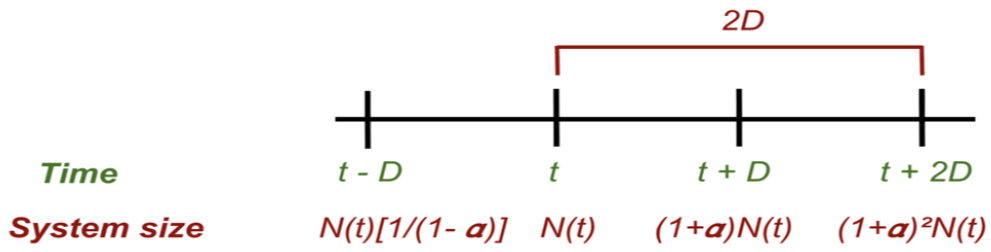

Figure 3.3: Maximum increase in system size during $[t - D, t + 2D]$

Figure 3.3 shows the maximum system size at various time intervals as a function of

the system size at time t. We estimate when at least D units of time have elapsed using the churn bound since the maximum number of churn events in the time interval $[t, t + D]$ is $\alpha \cdot N(t)$. As seen in Figure 3.3, we estimate time interval $2D$ based on the churn rate and the maximum possible system size after $2D$ units of time has elapsed. The maximum number of churn events that can occur in time interval $[t, t+2D]$ is given by $\alpha \cdot (1+\alpha) \cdot N(t)$. It could appear that waiting to receive messages about $\alpha \cdot (1+\alpha) \cdot N(t)$ churn events would be sufficient to measure $2D$ time. However, this in itself is not sufficient to guarantee that $2D$ units of time have elapsed since messages sent during $[t - D, t]$ might be received between $[t, t + D]$. We also need to take these messages into consideration. Therefore, at least $2D$ units of time have elapsed if the process counts the incoming $enter$ or $leave$ messages until it meets a target number of messages, to be calculated. An important thing to note that is that counting the incoming $enter$ or $leave$ messages until it meets the target number of messages as discussed above guarantees that at least $2D$ units of time has elapsed, i.e., the time elapsed could be greater than $2D$.

We prove that the target number of messages we need to wait for to ensure that at least $2D$ time has elapsed is a multiple of the size of $Present$ set.

Summing up, processes exploit the churn bound as a way to measure time. Therefore, during the failure detection phase, a process p sends the *fail-check* message to all processes and waits until the count of incoming $enter$ and $leave$ messaged meets the target number of messages (that signify that at least $2D$ time has elapsed) before classifying processes that did not acknowledge the message and have not left the system as failed.

4. FAILURE DETECTION ALGORITHM

We present an algorithm that enables active joined processes in the system to detect failures of other processes. All joined processes repeatedly run the failure detection algorithm in phases as long as they are active. At the beginning of each phase, the process calculates a target number of $enter$ or $leave$ messages it needs to wait for. The phase completes when the process receives the target number of messages. The target number of messages is calculated such that at least $2D$ time has elapsed since the process broadcasted the *fail-check* message.

In order for process p to begin failure detection, it has to be an active and joined process in the system. After process p joins the system, it starts executing the failure detection algorithm. Upon initiation, the algorithm maintains a counter for tracking the failure detection phase number that the process is currently executing and initializes it to zero. In addition, process p also keeps track of all failed processes as it advances through its various phases to prevent re-checking already failed processes again.

The algorithm then proceeds to execute the failure detection phases repeatedly. After the beginning of the failure detection phase, process p suspects all other active processes in the $Present$ set as failed. Process p then sends the *fail-check* message to all suspected processes to check if they are still active. After broadcasting the *fail-check* message, process p maintains a counter to count the number of $enter$ or $leave$ messages received by process p. Since the system is asynchronous, we use these churn events as a proxy to estimate elapsed time.

When process p receives an acknowledgement from process q for a *fail-check* message process p had sent or a leave message from process q, process p removes process q from process p's list of suspects after verifying that the acknowledgement was meant for the

current phase of failure detection.

Process p also replies with an acknowledgement message to process q after it receives a *fail-check* message from process q (even if not yet joined).

Whenever process p receives an $enter$ or a $leave$ message from another process, process p increments its counter for churn events. After incrementing the counter, process p checks to see if a $leave$ message was received. If so, p removes the leaving process from its list of suspects. Process p then checks to see if the counter for churn events has reached the threshold that would guarantee that at least $2D$ units of time has elapsed. If so, process p marks the processes that process p did not receive acknowledgements from as failed processes. Lastly, process p increments its phase number since the failure detection phase has ended and proceeds to repeat this entire failure detection phase again.

4.1 Local Variables

Table 4.1 explains the local variables used in our failure detection algorithm.

Table 4.1: Variables used in the failure detection algorithm by process p

Variable	Function
Present	p's estimate of the set of processes that have entered the system but not left
θ	$\left(\frac{\alpha(1+\alpha)^2(3-\alpha-\alpha^2)}{(1-\alpha)^3}\right)$
failed_processes	set of processes marked as failed
target_churn	number of $enter$ or $leave$ messages to wait for before ending current phase
churn_counter	tracks the number of $enter$ or $leave$ messages received in the current phase
fd_phase	tracks the phase of failure detection
execute_phase	tracks whether the phase is in execution
suspect_set	set of processes that are suspected as crashed

Algorithm 1 Failure detection algorithm - Code for active joined process p

1: $fd\_phase := 0$ ▷ tracks the phase of failure detection
2: $failed\_processes := \{\}$ ▷ tracks all failed processes
3: loop forever
4: $suspect\_set := Present - \{p\} - failed\_processes$
5: $target\_churn := \theta * |Present|$
6: $churn\_counter := 0$
7: $execute\_phase := true$
8: $bcast <$ "fail-check", $p, fd\_phase >$
9: **while** $execute\_phase$ **do** ▷ failure detection phase
10: **when** $\langle$"enter", $q\rangle$ **OR** $\langle$"leave", $q\rangle$ **is received**:
11: $churn\_counter++$
12: **if** $\langle$"leave", $q\rangle$ is received **then**
13: remove q from $suspect\_set$
14: **if** $churn\_counter \geq target\_churn$ **then**
15: $failed\_processes = failed\_processes \cup suspect\_set$
16: $fd\_phase++$
17: $execute\_phase = false$
18: **when** $\langle$"ack-fail-check", $p, phase, q\rangle$ **is received**:
19: **if** $phase = fd\_phase$ **AND** $q \in suspect\_set$ **then**
20: remove q from $suspect\_set$

21: *// also executed by nodes that have entered but not joined*
22: **when** $\langle$"fail-check", $q, phase\rangle$ **is received**:
23: **if** $q \neq p$ **then**
24: bcast $\langle$"ack-fail-check", $q, phase, p\rangle$

5. CORRECTNESS

In order to prove the correctness of our algorithm, we rely on some lemmas from [6]. Recall that the churn model and the $Present$ set that is maintained using the protocol in [6] are valid for our algorithm as well. Therefore, we use lemmas about the $Present$ set and the effect of churn on system size from [6].

Lemma 1 shows that the maximum system size at time t is a known multiple of the size of the $Present$ set. This is useful in proving Lemma 2 which shows that the minimum number of $enter$ or $leave$ messages needed to guarantee that $2D$ time has elapsed is also a known multiple of the $Present$ set, namely the value of θ in Table 4.1. Next, Theorem 3 proves the perfect accuracy of our failure detector by showing that every process marked by our failure detector as crashed has actually crashed. Lemma 4 comments on the completeness property of our failure detector algorithm stating that process q will be detected as crashed by process p in the current failure detection phase (in which q crashed) or the consecutive phase of p if p joins at least $2D$ time after q has entered. Lastly, Theorem 5 extends Lemma 4 to show that with infinite churn and a process that stays on in the system forever, that process can detect all failed processes.

Lemma 1. *For every process p and every time $t \geq t_p^{join}$, where t_p^{join} is the time when process p joins the system, at which p is active,*

$$N(t) \leq |Present_p^t| \cdot \left(\frac{1+\alpha}{1-\alpha}\right)^2$$

Proof. We begin by calculating an upper bound on $N(t-2D)$ as a function of $|Present_p^t|$ and use that to calculate an upper bound on at $N(t)$ as a function of $|Present_p^t|$.

From Lemma 7 from [6]:

$$(1-\alpha)^2 \cdot N(t-2D) \leq |Present_p^t|$$

$$\implies N(t-2D) \leq \frac{|Present_p^t|}{(1-\alpha)^2}$$

From Lemma 2 from [6]:

$$N(t) \leq (1+\alpha)^2 \cdot N(t-2D)$$

$$\implies N(t) \leq (1+\alpha)^2 \cdot \frac{|Present_p^t|}{(1-\alpha)^2}$$

$$\implies N(t) \leq |Present_p^t| \cdot \left(\frac{1+\alpha}{1-\alpha}\right)^2$$

□

Lemma 2. *If p receives more than or equal to $\theta \cdot |Present_p^t|$ enter and/or leave messages (for distinct processes) during the time interval $[t, t']$, with $t \geq t_p^{join}$, then $t' - t \geq 2D$.*

Proof. The maximum number of enter/leave messages that p can receive in $[t, t+2D]$ is the maximum number of enter/leave events that can occur in $[t-D, t+2D]$ since messages caused by events that occur in $[t-D, t]$ can take up to D time to reach p.

Step 1: We estimate the maximum possible system size at time $t - D$ as a function of $N(t)$.

From Lemma 1 from [6]:

$$(1-\alpha) \cdot N(t-D) \leq N(t) \leq (1+\alpha) \cdot N(t-D)$$

$$\implies N(t-D) \leq \frac{N(t)}{1-\alpha}$$

Step 2: We estimate the maximum possible system size at time $t + D$ as a function of $N(t)$.

From Lemma 1 from [6]:

$$(1 - \alpha) \cdot N(t) \leq N(t + D) \leq (1 + \alpha) \cdot N(t)$$

$$\implies N(t + D) \leq (1 + \alpha) \cdot N(t)$$

We are going to do a proof by contrapositive, so let us assume

$$t' - t < 2D$$

Let CE be the maximum number of enter or leave messages received by process p in $[t, t')$.

$$CE < CO_{[t-D,t]} + CO_{[t,t+D]} + CO_{[t+D,t+2D]}$$

where CO_I is the maximum number of enter or leave events that can occur in the time interval I.

From Step 2:

$$CE < \alpha \cdot N(t-D) + \alpha \cdot N(t) + \alpha \cdot N(t+D)$$

$$< \alpha \cdot [N(t-D) + N(t) + N(t+D)]$$

$$< \alpha \cdot \left(\frac{N(t)}{1-\alpha} + N(t) + (1+\alpha)N(t) \right) \text{... from Lemma 1 from [6]}$$

$$< \alpha \cdot N(t) \cdot \left(\frac{1}{1-\alpha} + 2 + \alpha \right)$$

$$< \alpha \cdot N(t) \cdot \left(\frac{3-\alpha-\alpha^2}{1-\alpha} \right)$$

$$< \alpha \cdot |Present_p^t| \left(\frac{1+\alpha}{1-\alpha} \right)^2 \cdot \left(\frac{3-\alpha-\alpha^2}{1-\alpha} \right) \text{... from Lemma 1}$$

$$< \left(\frac{\alpha(1+\alpha)^2(3-\alpha-\alpha^2)}{(1-\alpha)^3} \right) \cdot |Present_p^t|$$

$$< \theta \cdot |Present_p^t|$$

where $\theta = \left(\frac{\alpha(1+\alpha)^2(3-\alpha-\alpha^2)}{(1-\alpha)^3} \right)$

Therefore, if $t' - t < 2D$, we prove that $CE < \theta \cdot |Present|$. This is logically equivalent to the proving that if $CE \geq \theta \cdot |Present|$, then $t' - t \geq 2D$.

Since p receives more than or equal to $\theta \cdot |Present_p^t|$ enter and/or leave messages during the time interval $[t, t']$,

$$CE \geq \theta \cdot |Present_p^t|$$

$$\implies t' - t \geq 2D$$

□

Theorem 3. *Every process q in the $failed\_processes$ variable of process p of any active p has crashed.*

Proof. The reason process q is in process p's $failed\_processes$ set is that at the end of some failure detection phase of p, say the k^{th} phase, process q was in p's $suspect\_set$ variable. Since q was in the $suspect\_set$ of p, q was in the $Present$ set of p at the beginning of the k^{th} failure detection phase. According to analysis from [6], we know that q was in the system at the beginning of the k^{th} failure detection phase. If q had left the system during the k^{th} phase, it would have been removed from the $suspect\_set$ of p during the k^{th} phase. Therefore, since q was in the $suspect\_set$ of p at the end of the k^{th} failure detection phase, q was in the system throughout the k^{th} failure detection phase.

Let us assume for contradiction that process q has not crashed and let the failure detection phase k for process p run during time interval $[t, t']$.

At time t, p sends a *fail-check* message to q which is received by q by time $t + D$ due to the upper bound on the message delay, D.

After receiving the *fail-check* message, q replies with the *ack-fail-check* message. The *ack-fail-check* message also has an upper bound on message delay, D, and is received by p before or at time $t + 2D$. Upon the receipt of *ack-fail-check* from q, p removes q from the $suspect\_set$ set.

According to the failure detection algorithm, p receives $\theta \cdot |Present_p^t|$ enter/leave messages during the failure detection phase.

According to Lemma 2,

$$t + 2D \leq t'$$

Therefore, at time t', p does not have q in the $suspect\_set$ set at the end of the k^{th} failure detection phase, a contradiction. Thus, q has crashed. □

Lemma 4. *Suppose process q crashes at time t, with t in active process p's k^{th} failure detection phase, and q is in p's $Present$ set at the start of p's k^{th} failure detection phase. If p completes its $(k+1)^{th}$ failure detection phase, then p detects q as crashed.*

Proof. In other words, process q will be detected as crashed by process p in the current failure detection phase (in which q crashed) or the consecutive phase of p if q is in the $Present$ set of p at the start of the current failure detection phase.

Let the k^{th} failure detection phase for process p run for the time interval $[t_k, t_{k+1})$. Similarly, let the $(k+1)^{th}$ failure detection phase for process p run for the time interval $[t_{k+1}, t_{k+2})$.

Since q is in p's $Present$ set at the start of the k^{th} failure detection phase, q is also in the $suspect\_set$ of p at the start of the k^{th} phase. During the k^{th} failure detection phase, process p sends the *fail-check* message at time t_k, which reaches process q time $t_k + F$ where $F \leq D$ (upper bound on message delay). The time t at which process q crashes can lie in the time interval $[t_k, t_k + F)$ or time interval $[t_k + F, t_{k+1})$.

Case 1: Time t lies in the time interval $[t_k, t_k + F)$. Process q has crashed in the time interval $[t_k, t_k + F)$, it is not able to process the *fail-check* message, and, therefore, is not able to reply with an acknowledgement. Process p will complete its failure detection phase k and since it did not receive an acknowledgement from process q by the end of the k^{th} phase, process p will detect and mark process q as crashed.

Case 2: Time t lies in the time interval $[t_k + F, t_{k+1})$. Process p's k^{th} failure detector phase does not detect process q as crashed since process q responds with an acknowledgement for the *fail-check* message arriving in the time interval $[t_k, t_k + F)$.

Process p then starts the $(k+1)^{th}$ failure detector phase and sends the *fail-check* message at time t_{k+1}. Since time t at which process q crashed lies in the time interval $[t_k + F, t_{k+1})$, process q does not process the *fail-check* message sent at time t_{k+1}. Process p completes its failure detection phase $k+1$ and since it did not receive an acknowledgement from process q by the end of the $(k+1)^{th}$ phase, process p detects and marks process q as crashed.

Therefore, if process q crashes at time t with t in process p's k^{th} failure detection phase, q is in p's $Present$ set at the start of the k^{th} phase, and process p completes its $(k+1)^{th}$ failure detection phase, then p detects process q as crashed. □

Theorem 5. *If the churn continues forever and there exists an active process p that never leaves or crashes, then process p detects all crashed processes.*

Proof. Let q be any process that enters the system and does not leave. By the protocol in [6], there is a time t such that q is in p's $Present$ set for all times at or after t.

Let k be the index of the first failure detection phase of p that starts after q is in p's $Present$ set. The existence of k follows from the fact that the churn is infinite and thus the number of failure detection phases of p is also infinite. Thus, p completes its k^{th} and $(k+1)^{th}$ failure detection phases. By Lemma 4, p detects q as crashed. □

5.1 Discussion on θ

As shown in Figure 5.1, θ is monotonically increasing with the churn fraction, α. Based on the constraints on α from the CCR_{EG} algorithm in [6], α cannot exceed 0.04. Therefore, θ can never exceed 0.14467. Even with the highest churn, the number of $enter$ or $leave$ messages the process needs to wait for is not significant. Therefore, each failure detection phase only waits to hear from a very small fraction of the processes believed to be in the system.

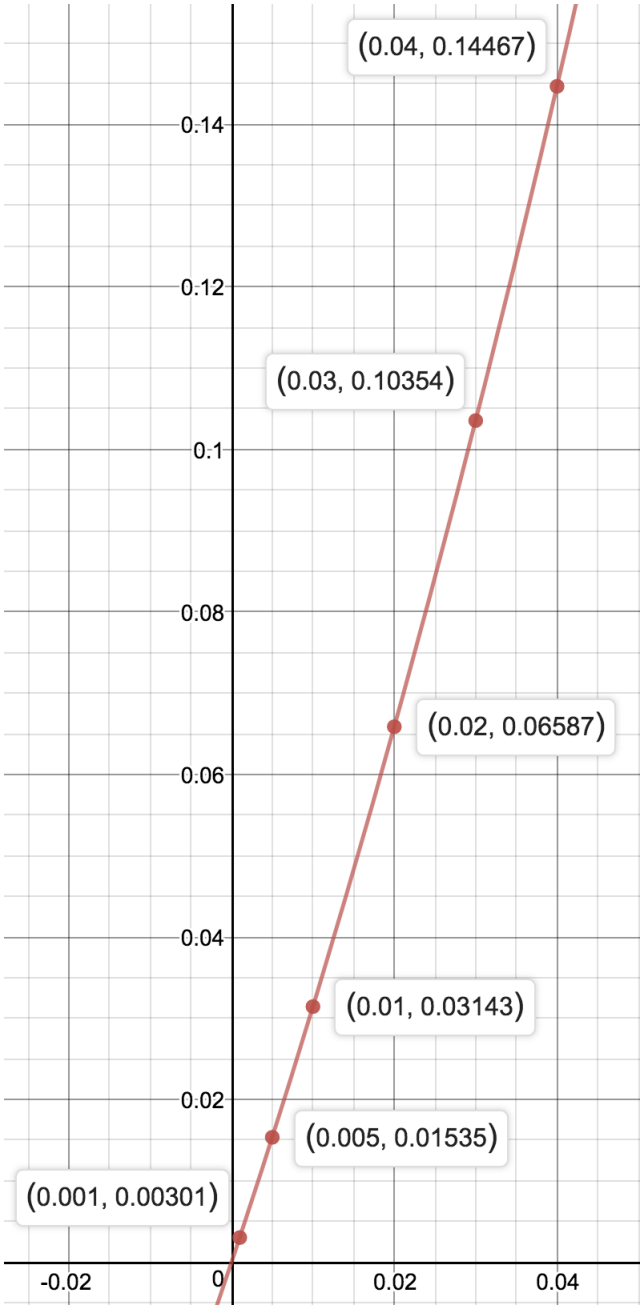

Figure 5.1: Graph plotting θ (y-axis) and α (x-axis)

6. CONCLUSION AND FUTURE WORK

Asynchronous systems with churn are prevalent in computing and a prime example is peer-to-peer networks. Previous work in this field has enabled such systems to simulate a read/write register but the problem of detecting failures had been largely unsolved.

In this thesis, we have shown the implementation of a failure detector for an asynchronous system with churn under the given churn model. All messages being transmitted in the system have an upper bound on message delay, D. Our churn model states that the number of processes entering or leaving the system must not exceed a fraction of the system size for a given time interval of length D. We exploit the churn bound to get an estimate of elapsed time since processes do not have way to measure time in an asynchronous system. We prove the correctness of our failure detector in addition to proving that under some circumstances, all crashed processes will eventually be identified by the failure detector if the churn continues forever.

6.1 Limitations

One known limitation of our work is that the failure detector is not able to detect failures if the churn stops completely, as the only sense of elapsed time the system measures comes from the churn. If the churn were to stop entirely, there would be no way the system would have any context about time, therefore, preventing the detection of failed processes.

6.2 Future Work

Finalizing a way to get crashed processes to exit the system after they have been detected as failed would be an interesting avenue of future work. Currently, we plan on using *announced leaves* as a mechanism for crashed processes to exit the system where processes that detect crashed processes are able to announce leaves on the behalf of crashed

processes to all the other processes in the system. However, some work still needs to be done on making sure the synchronization and communication in cases where multiple processes detect the same processes as crashed works smoothly.

We would also like to explore strengthening our failure detector by modifying the algorithm to allow processes to share their sets of identified failed nodes with each other. Propagating this information could improve our theorems and relax constraints about a process being in the system forever to detect all crashed nodes.

Another direction of future work could be exploring the possibility and applications of achieving consensus if the churn continues forever. Since we prove that under some circumstances, all crashed processes can be detected if the churn continues forever, we believe this could give rise to some form of consensus. However, some more work needs to be done to define what consensus in systems with churn would look like and to explore its various applications.

REFERENCES

[1] E. Talmage and J. L. Welch, "Message-passing implementations of shared data structures." UIUC Distributed Computing Tele-Seminar, 2017.

[2] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," Journal of the ACM (JACM), vol. 42, no. 1, pp. 124–142, 1995.

[3] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, pp. 272–281, IEEE, 1997.

[4] R. Baldoni, S. Bonomi, and M. Raynal, "Implementing a regular register in an eventually synchronous distributed system prone to continuous churn," IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 1, pp. 102–109, 2012.

[5] S. Gilbert, N. A. Lynch, and A. A. Shvartsman, "Rambo: a robust, reconfigurable atomic memory service for dynamic networks," Distributed Computing, vol. 23, no. 4, pp. 225–272, 2010.

[6] H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch, "Simulating a shared register in an asynchronous system that never stops changing," in International Symposium on Distributed Computing, pp. 75–91, Springer, 2015.

[7] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," Journal of the ACM (JACM), vol. 43, no. 2, pp. 225–267, 1996.