FAULT TOLERANT PULSE SYNCHRONIZATION

A Thesis

by

KEERTHI DECONDA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2008

Major Subject: Computer Science

FAULT TOLERANT PULSE SYNCHRONIZATION

A Thesis

by

KEERTHI DECONDA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Jennifer L. Welch
Committee Members,	Radu Stoleru
	Deepa Kundur
Head of Department,	Valerie E. Taylor

December 2008

Major Subject: Computer Science

ABSTRACT

Fault Tolerant Pulse Synchronization. (December 2008)

Keerthi Deconda, B.Tech, National Institute of Technology Warangal, India

Chair of Advisory Committee: Dr. Jennifer Welch

Pulse synchronization is the evolution of spontaneous firing action across a network of sensor nodes. In the pulse synchronization model all nodes across a network produce a pulse, or "fire", at regular intervals even without access to a shared global time. Previous researchers have proposed the Reachback Firefly algorithm for pulse synchronization, in which nodes react to the firings of other nodes by changing their period. We propose an extension to this algorithm for tolerating arbitrary or Byzantine faults of nodes. Our algorithm queues up all the firings heard in the current cycle and discards outliers at the end of the cycle. An adjustment is computed with the remaining values and used as a starting point of the next cycle. Through simulation we validate the performance of our algorithm and study the overhead in terms of convergence time and periodicity. The simulation considers two specific kinds of Byzantine faults, the No Jump model where faulty nodes follow their own firing cycle without reacting to firings heard from other nodes and the Random Jump model where faulty nodes fire at any random time in their cycle.

To my parents and sister,

And to those who are not with us,

You will always remain in our hearts.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Jennifer Welch for her constant guidance and support. Her encouragement kept me going whenever I hit a road block. I would also like to extend my thanks to Dr. Radu Stoleru who introduced this thesis problem to me. My gratitude to Dr. Deepa Kundur, who reviewed my thesis and provided thoughtful insights. I would like to thank all my friends who have been my support system during the course of my graduate studies.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Time synchronization is an important requirement of wireless sensor networks. Sensor motes run on inexpensive hardware with limited resources. Hence their clocks are very likely to drift away with time. Since the most important function of a sensor network is to sense events, the time of occurrence of the event should be very accurate. For example, in a surveillance application the velocity of the enemy vehicle could be estimated by the time difference between the events detected at different sensors.

Centralized techniques for time synchronization in sensor networks have a single point of failure at the base station. Traditional clock synchronization algorithms involve flooding the network from a reference node [11, 12] or constructing a spanning tree and synchronizing along its edges [13]. Such algorithms have a high communication overhead and hence may not be scalable to large networks. Therefore decentralized algorithms inspired from biological systems are highly attractive for wireless sensor networks.

As mentioned above, wireless sensor networks run on inexpensive hardware with limited battery. So lifetime is limited by the battery power. At any time nodes might drop out of the network and new nodes may be added to the network. Also depending on the terrain of deployment, certain radio links might become weak and an alternative

---

The journal model is *IEEE Transactions on Information Theory*.

radio link can be discovered. Hence synchronization algorithms that are flexible and can adapt to changes in network topology are highly desirable.

Due to the nature of deployment of wireless sensor networks, the nodes are susceptible to physical capture. Its software can be overwritten and it can keep sending malicious messages into the system. Such processes can hamper the working of time synchronization algorithms and can result in the sensor node having incorrect clock time. Hence time synchronization is a critical function and it must be made fault tolerant.

This thesis is organized as follows. The remaining sections of this chapter present the pulse synchronization problem and explain the importance of fault tolerance. Related work section describes previous models and research in the area of pulse synchronization. Chapter II presents our main contribution, a fault tolerant algorithm for pulse synchronization. The design of the algorithm and its implementation are explained. Chapter III describes our simulation efforts to verify the algorithm and presents the results of those experiments. We conclude by discussing the implications of our results and future directions of this research. Appendix A contains some analysis of Mirollo and Strogatz model [1]. The simulation code is included in Appendix B.

A. Pulse Synchronization

Pulse synchronization is a very interesting phenomenon in natural and biological systems. Some examples of biological oscillators found in nature are fireflies flashing in unison and pacemaker cells of the heart. Such systems consist of identical units which communicate by exchanging “firing” messages, which is the only communication

between them. An oscillator is characterized by a phase which varies with time. The phase rises from zero to a certain threshold and when the oscillator hits the threshold it "fires" and this firing is seen by all other oscillators in the system. Whenever one oscillator fires, the other oscillator observe this firing and adjust their rhythm accordingly. This adjustment, called coupling in classical mechanics, is performed at discrete times (whenever a firing is observed) rather than smoothly over time. Therefore units exhibiting the above behavior are called pulse coupled oscillators.

It was proved by Mirollo and Strogatz [1] that in a system consisting of N identical pulse coupled oscillators, regardless of number of oscillators in the system, for almost all initial conditions it converges to a state in which all units are firing synchronously (See Appendix A for some related analysis). This synchronization occurs in a totally distributed fashion and with minimal local state information. There is no centralized node directing the actions of the other nodes.

The M&S model consists of N identical pulse coupled oscillators moving towards threshold according to a specific function. This function should be monotonically increasing and concave down. All other oscillators react by pulling up their phase by an amount or to the threshold, whichever is lower. The firing node's phase then goes back to zero. The amount by which a node is pulled up is defined as a system parameter ε. Due to this pulling up of the phase, the phase difference between nodes becomes lesser in each cycle and eventually becomes zero. Thus the nodes are firing simultaneously and remain in such state subsequently. When the system is in such a state it can be said to have achieved pulse synchronization.

Such a concept is especially attractive for wireless sensor networks. This model has the advantages of being scalable, decentralized and does not require that many message exchanges. Coordinated duty cycling and sampling across the network are some applications. However this ideal model cannot be directly applied to less than the ideal world of wireless sensor networks. For one, there is always some delay in communication between two nodes. Hence when a node fires, other nodes don't hear this message instantaneously and each node may hear it at different times. Due to lossy radio links, some nodes might not hear this firing message at all. Werner-Allen et al. [2] address this issue in their paper.

The Reachback Firefly algorithm proposed in [2] accommodates the message delay factor. Low level timestamping of the firing message is used to calculate the exact delay in transmission and hence exact firing time is known. Not only is the firing message received after a delay, but it could be received out of order as well. Hence all the received messages are placed in a queue to be dealt with at the end of the time period. When a node A fires and its neighbor B hears this firing, node B does not immediately react. It "remembers" the internal time at which it heard the firing. Now when node B reaches the end of its time period, it recalls all such firings heard and computes the overall jump and increments its internal time immediately. Therefore a node reacts to messages from the previous time period rather than the current time period.

B. Fault Tolerance

Any practical distributed system must be able to deal with process failures, repairs, clock drifts and varying message delivery times. Byzantine fault tolerance is an issue that could arise in such a self-synchronizing system. A node exhibiting Byzantine faults could behave erroneously or perform arbitrary execution of a step. This faulty behavior could be because of hardware problems, network disconnects or malicious attacks. Byzantine faults in any of the pulse synchronization algorithms could prevent stabilization in the synchronization process. A faulty node consistently firing at a wrong time could leave the system unstable forever.

Lamport et al [10] propose an algorithm to achieve clock synchronization in presence of Byzantine faults. In their fault tolerant algorithm, non-faulty processes try to reach a consensus between them. The algorithm runs in rounds and obtains clock values of all other nodes and sets its clock to average of those values. It was proved that clock synchronization can be reached successfully, if and only if less than one-third of the total numbers of processes are faulty. The interactive convergence algorithm they described handles faults in a completely connected network.

The algorithm proposed by Welch et al. in [6] improves upon the above by introducing a fault tolerant averaging function. Instead of using a simple average, this function is designed to be immune to fixed maximum number of faults f. This function throws out f highest and f lowest values from all the values collected at the end of each round. Then an ordinary averaging function is applied to the remaining values. It was

proved in [6] that this averaging function halves the distance between clocks in each round.

C. Related Work

Peskin is one of the first researchers to provide a model for the self synchronization of a cardiac pacemaker [4]. The pacemaker is modeled as network of N "integrate-and-fire" oscillators characterized by a state variable x_i , which is subject to the equation

$$\frac{dx_i}{dt} = S_0 - \gamma x_i \qquad 0 \le x_i \le 1, i = 1, \dots, N.$$

$$x_i(t) = 1 \Rightarrow\ x_j(t^+) = \min(1, x_t(t) + \varepsilon)\ \ \forall j \neq i.$$

where S_0 is a system defined constant and γ is a small dissipation constant.

When x_i=1, the i^{th} oscillator fires and jumps back to zero. It pulls up all other oscillators by an amount ε, the coupling strength. The higher the value of ε, the greater the coupling achieved at each firing. The rate at which pulse synchronization can be achieved is dependent on ε. Peskin conjectured that for arbitrary initial conditions, the system approaches a state in which all oscillators are firing synchronously. He proved it for the special case of *N*=2 oscillators.

Mirollo and Strogatz [1] analyze a more general version of Peskin's model for all *N*. Instead of the above equation, the oscillators rise towards threshold following a function which is monotonic and concave down. The assumptions are that the oscillators are fully connected and identical. Also there is no delay between firing and receiving the messages.

Hong and Scaglione [5] is one of the first papers to apply the pulse synchronization model to wireless sensor networks. They utilized the characteristic pulse of Ultra-Wide-Band (UWB) to emulate the synchronization process of pulse coupled oscillators, and included more realistic effects such as channel attenuation and noise. UWB radio can be used at very low energy levels for short-range high-bandwidth communications. Due to the wide bandwidth provided by UWB systems, nodes can emit signals with small pulse durations compared to their duty cycle and therefore emulate the mechanism of firing. By using UWB pulses, there is no delay in sending and receiving the firing signal. They also assume that a rest period exists immediately after node fires, during which it cannot couple with any other node. This is necessary to avoid feedback cycles from other nodes of the network and result in an infinite excitation between close by nodes.

Another assumption of M&S model is that all nodes are directly connected to every other node. That is all-to-all communication links in the system. In [3], this assumption is relaxed and it has been proved that the system still converges to synchronized state, even though nodes can hear only some of their neighbors firing. Lucarelli and Wang [3] relax the assumption of all-to-all coupling in the network. They have shown that in a sensor network, coupling with just the nearest neighbor is sufficient to achieve stability. Thus, they have proved that convergence can be achieved for an arbitrary topology. Again they haven't considered delay in receiving the message.

In [2] Werner-Allen et al. propose the Reachback Firefly Algorithm (RFA), in which a node reacts to messages from the previous time period rather than the current

time period. When a firing message is received, MAC level timestamping is used to estimate the correct firing time. The node on receiving the firing message does not immediately react but places the message in a queue. At the end of the cycle, all received firing messages are sorted in order of their exact firing time. The node reaches back in time and computes all the jumps that it should have taken, had it received the messages instantaneously and in the correct order. Instead of starting from zero after firing, it starts at the overall jump computed in the previous cycle. For the special case of N=2 and $f(t)=ln(t)$, it has been proved that this scheme converges to synchrony for almost all initial conditions. Their analysis makes use of a fixed repeller point due to which the system is driven to synchrony. The time to converge is inversely proportional to ε, thus a large ε causing faster convergence and a small ε causing longer time to converge. In their simulations, some initial conditions never lead to synchrony. This happens when ε is a large value causing the nodes to make large jumps and hence overshooting above the threshold.

Clock Synchronization and Pulse Synchronization are complementary concepts. Clock synchronization means bringing all processes in a system to agree on the notion of a common clock time which can be mapped to real time. If clocks of the whole network are kept synchronized, then it is trivial to arrange pulses to be produced at scheduled times. Conversely, if a system is pulse synchronized, it can achieve clock synchronization as we now explain. First, note that the pulse synchronization produces a pulse (or a firing) periodically, which can be used for the nodes to agree on the

beginning of each period. This knowledge is used as a building block to achieve clock synchronization in two algorithms, which are described next.

Daliot et. al. [7, 8] proposed a Byzantine self-stabilizing clock synchronization algorithm that improves upon exponential convergence times and achieves convergence in linear time. Their algorithm is built on top of a self-stabilizing Byzantine pulse synchronization block. This pulse must occur simultaneously across all nodes. Briefly the pulse synchronization algorithm [8] works as follows. In a system of n processes out of which f are faulty, every Cycle time units each node sends a propose_pulse message. When (n-f) messages are collected, nodes enter a Byzantine agreement to decide on the time of message sent. Based on the time agreed a node invokes a pulse. At the occurrence of a pulse (similar to firing), nodes execute another Byzantine consensus algorithm to agree on the next clock synchronization time.

Dolev and Hoch [9] also describe a clock synchronization algorithm which is built on top of a pulse synchronization system. Their model differs from Daliot's model in that all nodes have access to a global beat system and operate in lockstep according to these beats. A pulse has to be produced for every Cycle number of beats. This pulser is constructed in a modular manner and the core module is the Byzantine Black Box (BBB) algorithm. At every beat the BBB algorithm is run and if agreement is reached then a pulse is produced. Given a Byzantine tolerant algorithm that is not stabilizing, the pulsing algorithm can be used to transform it into a stabilizing version.

Another paper by Daliot, Dolev and Parnas [16] is most similar to this thesis. The pulse synchronization model defined in their paper is the same as in this thesis. Though

their approach to the solution is quite different. Each node sums the pulses that it learns about during a recent time window and fires if the sum reaches a threshold. The counter value is broadcast at firing time. The receiving node checks the credibility of the message counter. Their algorithm can tolerate failures of up to one-third of the nodes. The identity of the sender needs to be known and verified at the receiver's side. In our algorithm which is based on the Reachback Firefly algorithm, the identity of the firing node is not required, only the time of the firing is important. Also in our algorithm the content of the firing message is not essential.

Distributed Firing Squad problem [15] is somewhat similar to the pulse synchronization problem. An algorithm is executed in response to request from a specific processor (called the general) or from an external source. If the request is from a correct processor then all other correct processors start the algorithm in unison ("fire"). However if the requestor is faulty then correct processors learn of the request at different steps. The firing mechanism in DFS problem is somewhat similar to the pulse firing in this thesis. However the firing is a one-time only process in the DFS algorithm, after which the algorithm halts. Whereas pulse synchronization keeps pulsing periodically throughout the system lifetime. The solution involves authenticated messages, whereas in our fault tolerant algorithm message sender is not known, the receiver is only concerned with the time of sending the message. The authors of the DFS problem study the fail-stop, rushing and collusion models of Byzantine failures. These fault models are different from the No Jump and Random Jump fault models in this thesis.

CHAPTER II

PROBLEM DESCRIPTION

A node acts as an oscillator with a fixed time period T. Each node has an internal clock t operating from a value of t=0 to t=T. When a node reaches the end of its time period i.e. t=T, it "fires" and starts a new time period from t=0. This "firing" can be thought of as a broadcast message sent to all the neighbors of a node. The nodes are connected in an all-to-all communication model. The system consists of n such nodes, each of which can start out with any initial clock value. A node maintains no internal state information other than this clock value. This model is based on Peskin's model of N "integrate and fire" oscillators [4].

Pulse synchronization involves getting the nodes to fire at the same time. In such a situation all nodes would start out their cycle at the same time, as well as reach the end of their cycle at the same time. For this synchrony to emerge the following algorithm has been developed by Mirollo and Strogatz [1].

When a node observes a firing from a neighbor, it reacts by adjusting its clock by a small value. This adjustment is done according to a firing function $f(t)$ and a small constant ε. We can say the clock is pulled up by a small amount ε. If the node has a clock value t when a firing is heard, the new clock value is calculated as

$$t' = f^{-1}(f(t) + \varepsilon)$$

Hence the increment Δ is

$$\Delta = t' - t$$

An important result proved by Mirollo and Strogatz [1] was that for *n* nodes starting out at any initial clock values *t*, if the function *f* is smooth, monotonically increasing and concave down, then the *n* nodes will eventually converge. By converge we mean that all nodes start firing at the same time.

In the RFA algorithm proposed by Werner-Allen [2], a node does not react instantaneously to a firing heard. It starts out its time period from a clock value t=0 and fires when it reaches t=T. Instead whenever a firing is heard, it computes the increment Δ that it was supposed to have taken. At the end of its time period, all such increments are added up and the node starts back at the sum of all those increments instead of t=0. RFA algorithm solved the problem of communication delays, where a firing message may not be heard immediately after it was sent.

The above algorithms assume that all nodes are identical. They don't take into account faulty or malicious nodes which may not follow the prescribed correct behavior. A node is said to be faulty if it does not follow the above behavior. Faulty nodes can prevent convergence from taking place. In this paper we describe two possible fault models. The two faulty behaviors for malicious nodes are:

(a).No Jump: Where a node refuses to participate in the converging algorithm. It still continues to fire at a regular period, just that it does not react to a neighbor's firing.

(b).Random Jump: When a node hears a firing from its neighbor, it responds in a random fashion. Instead of adjusting its clock in a small increment, its sets its clock to a totally random value between t=0 and t=T. This leads to the node firing at random times, rather than following the regular clock cycle.

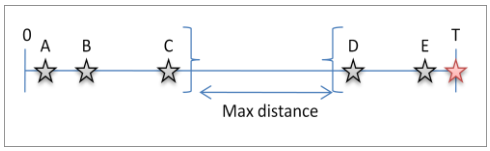
It is possible that a faulty node could incorporate both (a) and (b) fault models simultaneously. A fault node could choose to not respond to a firing or jump to a random value. In this paper we assume that a faulty node follows either (a) or (b) but not both at the same time. We leave this for future work. The next section describes an algorithm that is designed to be immune to the above faults.

A. Algorithm

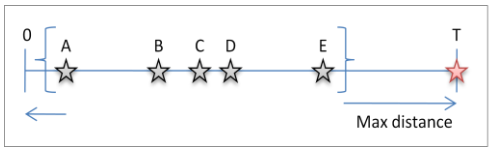
We propose a new modification to the Reachback Firefly Algorithm to tolerate the effect of faulty processes. In this algorithm a node follows its cycle from *t=0* to *t=T*, while making note of all the firings heard. Here a node merely takes note of its position in its current period when a firing was heard. If a node was at t=*t'* when a neighbor fires, the time *t'* is noted down. At the end of its time period, the overall jump is to be calculated. Until here it's the same as the RFA algorithm. But instead, when computing the overall jump a fault tolerant averaging function is used. This function is similar to the one proposed for approximate agreement problem in [14] and adapted for clock synchronization in [6]. Ideally when a node reaches the end of its cycle it would have heard *n-1* firings, not counting its own firing. Random jump faulty nodes sometimes cause more or less than *n-1* firings to be heard. This deficit is taken into account by a method which is described later. From these *n-1* clock values placed in the queue, we discard the *f* highest and *f* lowest values. The overall jump is calculated from the remaining *(n-2f)* values. It is to be noted that even though we call this function a "fault tolerant averaging function", there is no average being computed. We keep the original

name as proposed in the paper [6], but the main idea is that we throw out the f highest and f lowest values.

The best values to calculate the fault tolerant average are those firings which are closest to each other. The intuition is that the nodes closest to each other are to be driven even closer to form a group and nodes further from this group are to be discarded. Therefore we consider the values that are most distant from each other as the end points, when determining the highest and lowest values. We need to keep in mind the "wraparound" factor as well when finding the end points.



(a): C and D are end-points.



(b): A and E are end-points.

Fig. 1: Examples of outliers with and without wrap-around.

For the above example in Fig 1, the values A, B, C, D and E are the times at which firings were heard. In Fig 1(a) the firings at E, A and B are closest to each other. When the maximum distance between individual firings are measured, C and D are the farthest apart. Hence firings C and D are considered the end points. Hence after wrapping around, the order of the firings heard is D, E, A, B, C. Now if we want to discard the highest and lowest 1 value, we discard D and C. The overall jump is computed using the values E, A and B.

In Fig. 1(b) values at B, C and D are closest to each other and maximum distance is between E and A. Therefore after wrapping around the order is A,B,C,D,E. After throwing out highest and lowest, we have B,C and D remaining.

The step-by-step working of the pulse synchronization algorithm is as follows:

1. A node advances its internal clock t from 0 to T steadily.
2. When it reaches a value of T, it "fires". This firing message is sent to all nodes in the network.
3. When a neighbor hears this firing, it makes a note of the time t' at which the firing was heard and places t' in its queue. This t' is the position in its own cycle, not the global time (which is unavailable to the nodes).
4. At the end of its cycle when $t=T$ after a node has sent the "firing" message, a node runs the fault tolerant averaging function on the timings of all the firings it has heard and computes the jump for the remaining values. Say the overall jump is calculated as σ.
5. The clock time is reset to this jump value that was computed. Hence in the next cycle, clock starts from this initial value $t=\sigma$ rather than $t=0$.

As this process continues the firings of non-faulty nodes will start moving closer together. Ideally the firings should converge, i.e. all non-faulty nodes firings simultaneously at the same instant. But because of the corrections performed exact synchrony is hard to achieve. Hence for practical purposes we say that convergence is achieved when firings occur close together within a small time window w. Once nodes

come together within this time window they remain together. This behavior was observed through simulations of the algorithm.

B. Implementation

The simulation of this algorithm was done in Java. In the implementation of this system each node is a separate object with clock time as its internal state, in addition to some other data. We maintain a notion of global time. This is only to keep track of sequence of events, rather than to influence the pulse synchronization algorithm. The time period of the clock cycle is taken as 1. Hence the clock cycle starts from t=0 and node fires when t=1.

A queue is used to simulate events that occur in the system. Nodes are placed in the queue sorted according to their clock times. Nodes with higher clock values appear at the head of the queue and lower clock values at the end of the queue. Therefore the next node that has to fire is found at the head of the queue.

To simulate the next firing, the node at the head of the queue is removed from the queue. The clock state of this head node is advanced to 1, to move forward the system time to the actual firing time. All other nodes are also advanced by the same time. When this firing occurs, all other nodes make a note of the time in their own cycle when they heard the firing. These values are used to compute the overall jump that the node was supposed to make. Hence the head node uses a fault tolerant averaging function to compute this jump and sets its internal clock to this jump. Therefore the node starts its new cycle from the jump that was calculated from the previous period.

In an ideal situation, a node would have heard exactly *(n-1)* firings during its time period not counting its own firing. However due to reachback a node may jump ahead of other nodes when starting out a new time period. Hence it would hear less than *(n-1)* firings in that period. In the case of Random Jump faults, the faulty nodes keep firing out of turn and hence it is possible to hear more than *(n-1)* firings in a cycle. There is no problem in the latter case, as we can still discard the highest f and lowest f values. Whereas when there are less than *(n-1)* values and if we discard *2f* values, there might not be enough values left to compute the jump. The solution here is to pad the queue with dummy values. The dummy value should be chosen such that it does not badly affect any of the good values.

(a) B's firing is not heard (b) After padding

Fig. 2: Padding with dummy values.

Given a set of values representing the firings heard during a cycle, the end points are the values that are most distant from each other. The midpoint between the two end-points is chosen as the dummy value to pad the queue. From the example in Fig. 2 shows firings heard when n=6 and f=1. In Fig 6(a) C and D are most distant from each other and hence they are chosen as the end points. The dummy value is chosen as point B which is the mid-point between C and D. Now when the algorithm discards outliers, B and C are the new end points (breaking the tie randomly since B and D which are also

equally distant). Thus we ensure that more of the dummy values end up as outliers and less of the actual values.

CHAPTER III

RESULTS

The performance of the algorithm has been tested through simulation. For a system of *n* nodes and f faulty nodes given the same input, the following sets of experiments were performed.

1. Reachback Firefly algorithm (orig): All nodes are non faulty. All values are used for computing jump.
2. No Jump faults (r1): Exactly f nodes are faulty in system of *n* nodes. The faulty nodes don't react in response to a firing heard. For such faulty nodes the initial state after a firing is always 0. All values are used for computing jump.
3. Random jump faults (r2): Exactly f nodes are faulty in system of *n* nodes. Faulty nodes don't follow the curve function to compute the jump. When a firing is heard these nodes jump to any random value between 0 and 1, following a uniform distribution. All values are used for computing jump.
4. Fault tolerant algorithm (ft0): All nodes are non-faulty. But fault tolerant function is used to discard outliers at the end of the cycle. This is to analyze the overhead introduced by the fault tolerant algorithm.
5. No Jump faults with fault tolerance (ft1): Exactly f nodes are faulty in system of *n* nodes. The faulty nodes don't react in response to a firing heard. But instead fault tolerant function is used to correct the faults at the end of cycle.

6. Random jump faults with fault tolerance (ft2): Exactly f number of faulty nodes which jump to a random value. Fault tolerant function is used to discard outliers at the end of cycle.

For each of the 6 variations of the algorithm, we run 5 simulations of each algorithm and take the average of convergence time. *n* input values are chosen uniformly at random. These input values are assigned to node ids in increasing order (i.e., the smallest input value is assigned to node 0, the second smallest to node 1, etc.). Then the identity of the f faulty nodes is chosen. The faulty nodes are selected such that there are equal number non-faulty nodes between faulty nodes. For example, for a simulation of n=7 nodes and f=2 faulty nodes, n_0 and n_4 are chosen as faulty nodes. The placement of the faulty nodes among the non-faulty nodes is chosen such that all faulty nodes are not placed consecutively. Note that this placement of faulty nodes is only for the first cycle. During subsequent cycles the position of faulty nodes may vary.

The main parameter that influences the firing cycle is Firing Function Constant (FFC). As mentioned in the previous chapter, when a firing is heard at time t the jump is calculated as $\Delta(t) = f^{-1}(f(t) + \varepsilon) - t$. FFC is defined as 1/ε, the inverse of ε. Small values of FFC cause larger jumps and hence causing convergence at a faster rate. But conversely it also causes overshooting of the clock value above 1, hence less stability. We vary the FFC from 70 and 500. The experiment methodology has been adapted from [2] where they use the same FFC values.

A. Group Spread

In the M&S model [1] when all nodes converge and their firings become synchronous, they fire at exactly the same instance and with the exact time period between each firing. However in the Reachback algorithm and our fault tolerant algorithm, due to corrections being made at the end of the cycle and not instantaneously, exact synchrony is hard to achieve. Hence we define what could be a reasonable measure for convergence. These measures are same as the ones used in RFA algorithm [2].

Group spread is defined as the maximum difference between firings of non-faulty nodes in a given time cycle. Given a time period consisting of firings, the firings can be put into clusters such that every node firing must fall within exactly one cluster and firings closest to each other form a cluster. Firings in two different clusters must be distant from each other. We define convergence as time for all nodes to come together into a single cluster and when the group spread becomes smaller than an acceptable value called a time window. A small time window parameter takes a longer time for the nodes to converge. We say that nodes are converged when the group spread remains within the time window for the last 9 out of 10 firings.

The graph in Fig.3 shows the evolution of group spread between the nodes. This figure shows a system of n=10 and f=3, initially starting from random initial clock values following a uniform distribution. Time window is 0.1 and FFC is 100. Group spread in the initial state is 0.70, as time passes group spread decreases gradually until it drops to a less than the time window. When it has remained in that state for that last 9 out of 10 firings, the simulation stops.

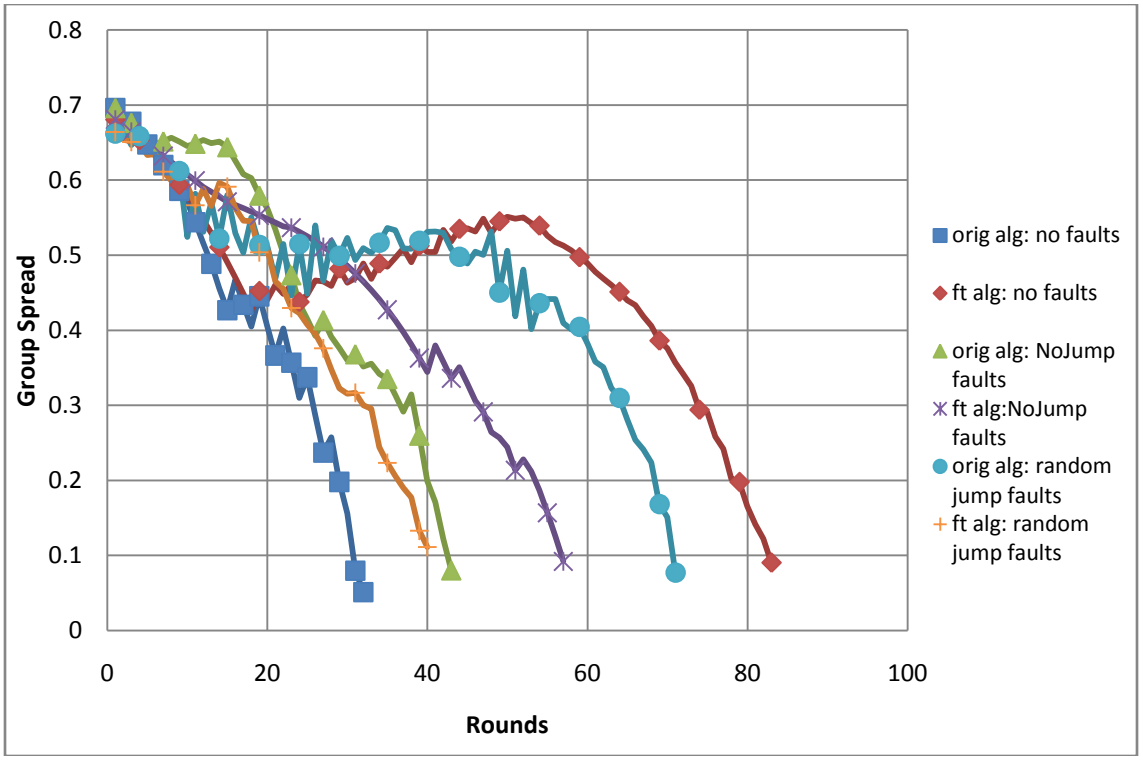


Fig. 3: Evolution of Group Spread.

B. Time to Converge

As described above, the time to converge is defined as the time for nodes to come close together in their clock values, and remain that way for last 9 out of 10 firings. We run multiple experiments of the same algorithm variant and take the average convergence time. Fig. 4 compares the convergence time for the original RFA algorithm against our fault tolerant algorithm when there are zero faults. The unit of time is the logical time unit, considering T=1.0. Hence the time period goes from t=0.0 to t=1.0. Hence the unit of time is the same as one cycle.

The graph shows the convergence times for *n*=4, 7 and 10 nodes for increasing FFC values, when all nodes start from random initial states. For almost all cases ft0 introduces an overhead in the convergence. In the case of *n*=10 when FFC=100, 250 and 500, there is almost a 100% increase in convergence time. The convergence time shown is the average of 5 runs, with percentage error varying from 20%-50%.

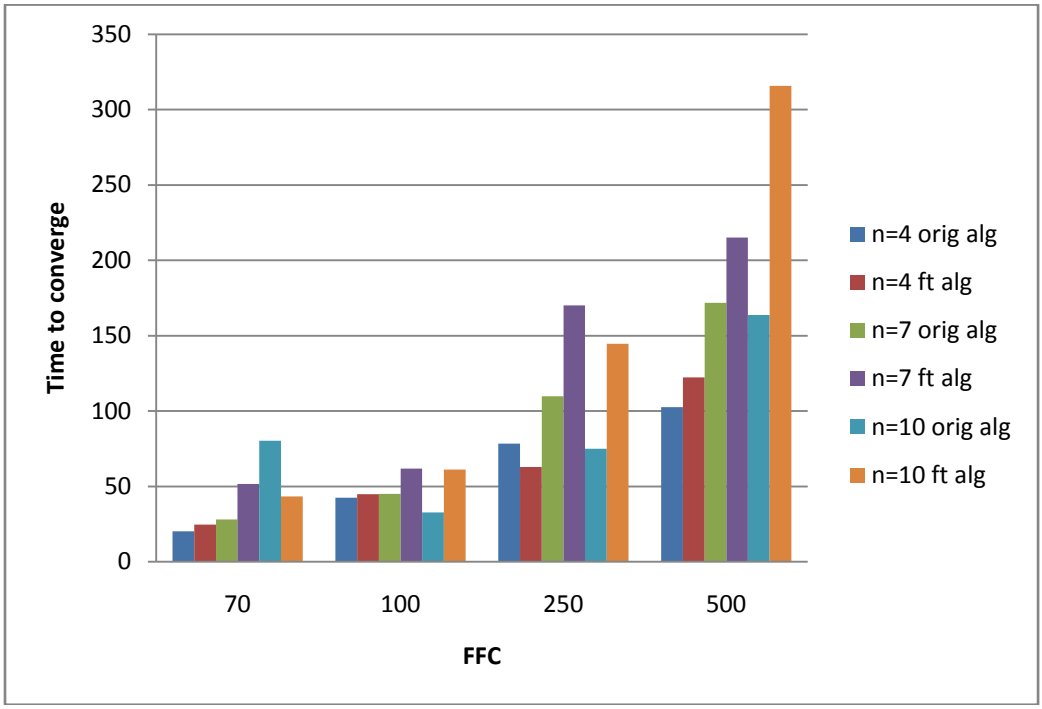
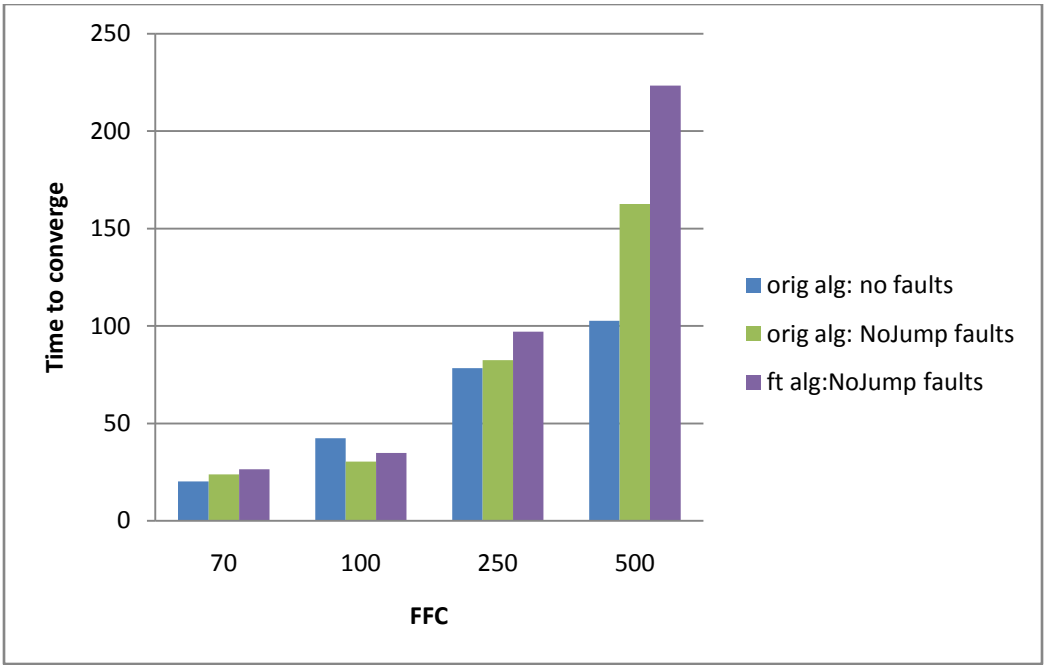
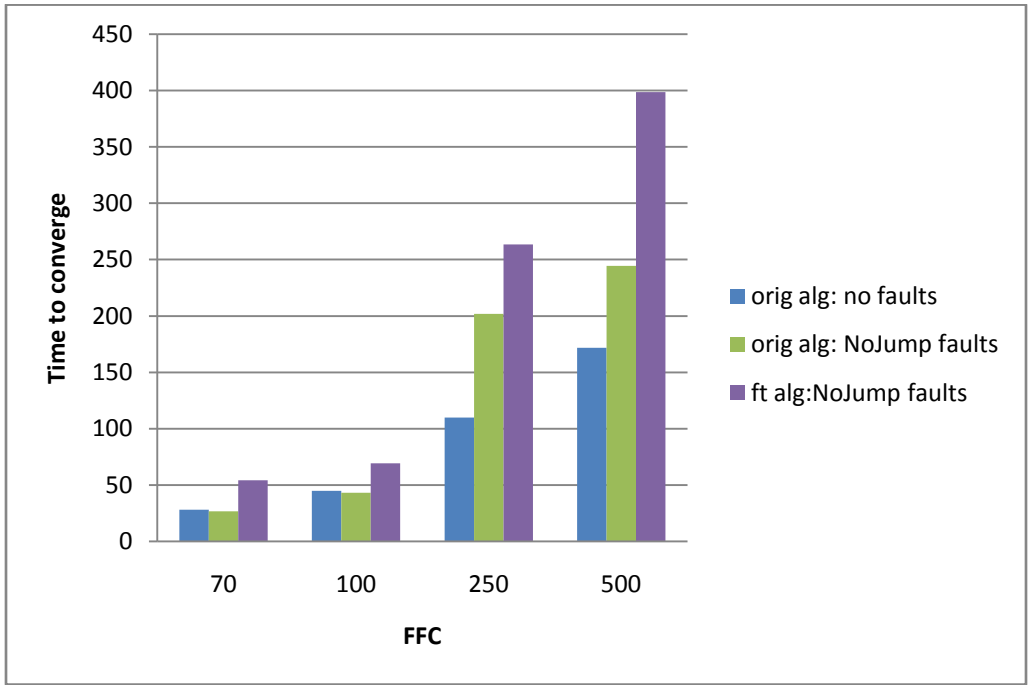


Fig 4. Comparison of Reachback and Fault Tolerant algorithm when no faults.

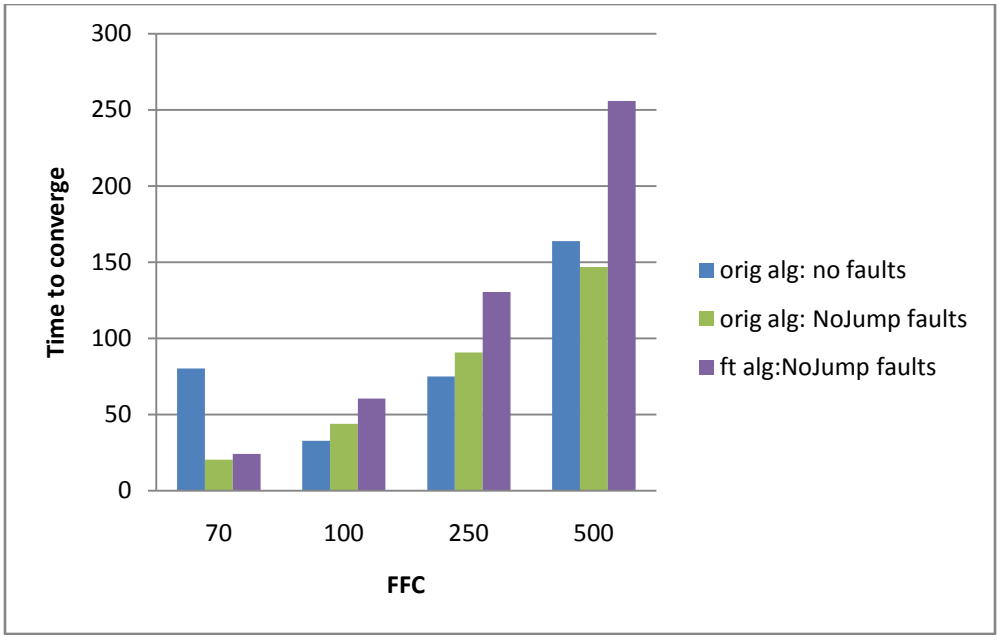


(a) n=4, f=1.



(b) n=7, f=2.

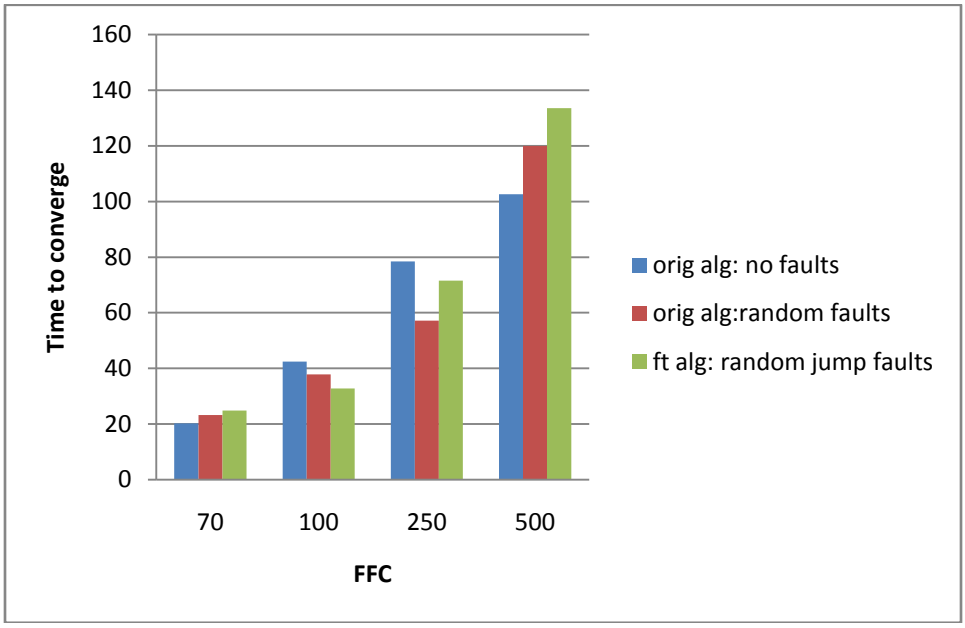
Fig. 5: Convergence Time with No Jump faults.



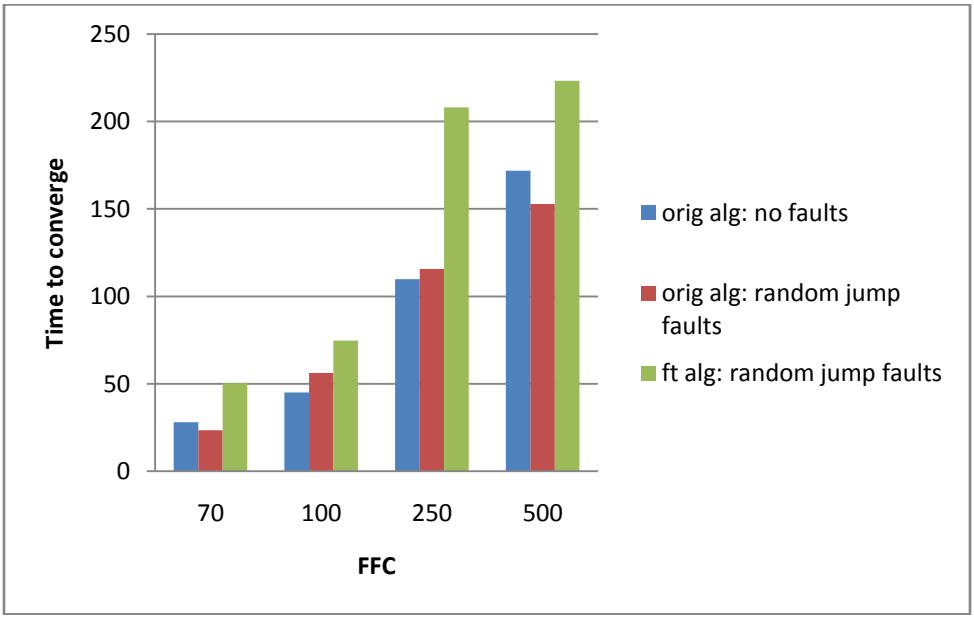
(c) n=10, f=3.

Fig. 5 (Continued)

Fig. 5(a)-(c) show the convergence time for increasing values of FFC. We compare the RFA algorithm with no faults, the RFA algorithm with No Jump faults and the fault tolerant algorithm with No Jump faults. The initial state is random and time window is 0.1. From the results we can see that smaller FFC values cause fewer rounds to converge. But there is no observable pattern for small FFC values such as FFC=70. With a few exceptions, No Jump faults increase the time taken for convergence. We discuss the possible reasons for this anomaly in the discussion section below. Our fault tolerant algorithm always increases the convergence time compared with the original algorithm with faults. This is in line with our observation that having a fault tolerant averaging function introduces an overhead. The convergence time shown is the average of 5 runs, with percentage error varying from 26%-55%.

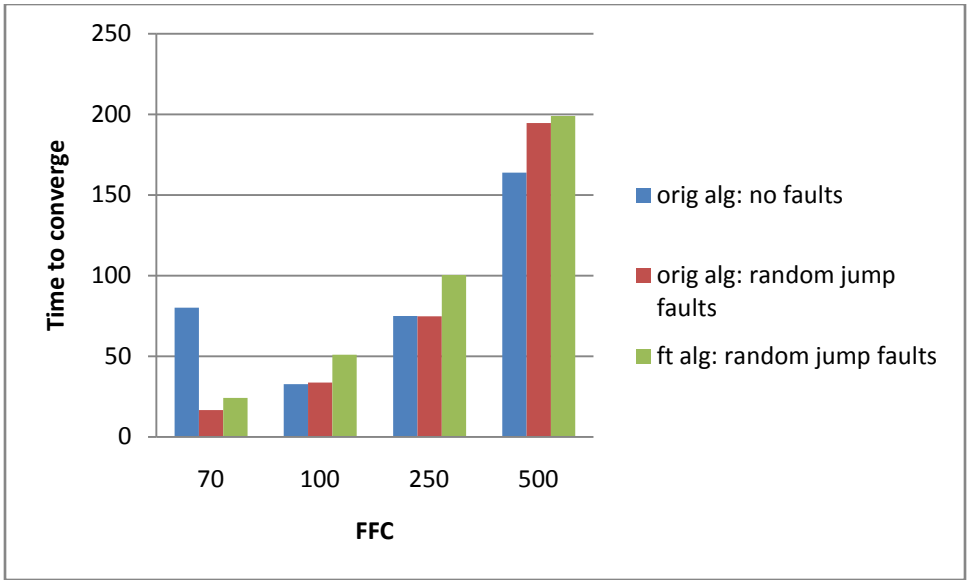


(a) n=4, f=1.



(b) n=7, f=2.

Fig. 6: Convergence Time with Random Jump Faults.



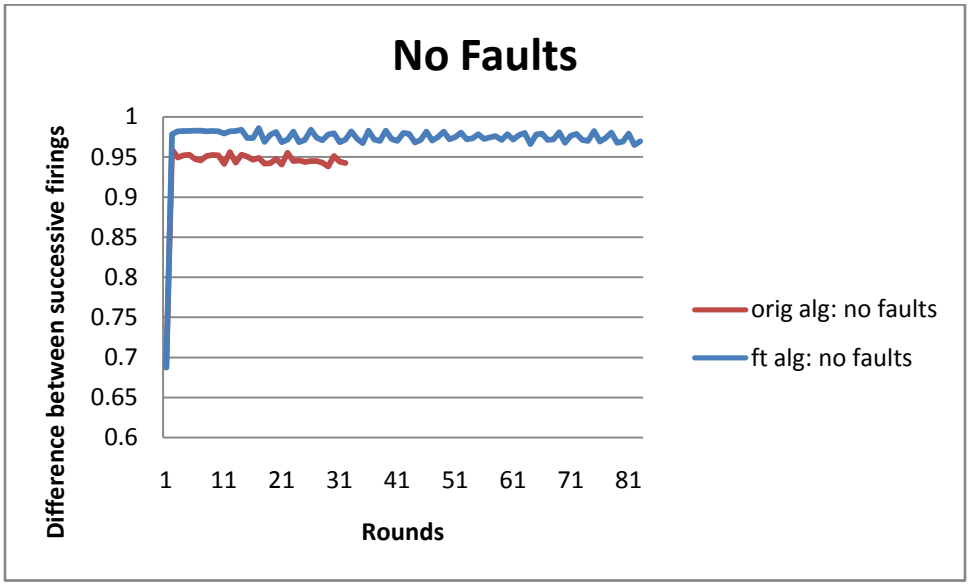
(c) n=10, f=3.

Fig. 6 (Continued)

The above graphs from Fig 6(a)-(c) shows the performance of RFA algorithm with no faults, the RFA algorithm with Random Jump faults and the fault tolerant algorithm with Random Jump faults. The initial state is random and time window is 0.1.The difference from Fig 4 graphs is that the RFA algorithm with Random Jump faults does not always decrease convergence time. Because a faulty node randomly switches its firing time, the convergence time is not really predictable. In this example, fault tolerant algorithm always increases the time towards convergence. But this is not always the case, for a different set of inputs another trend might show up. The convergence time shown is the average of 5 runs, with percentage error varying from 36%-68%.

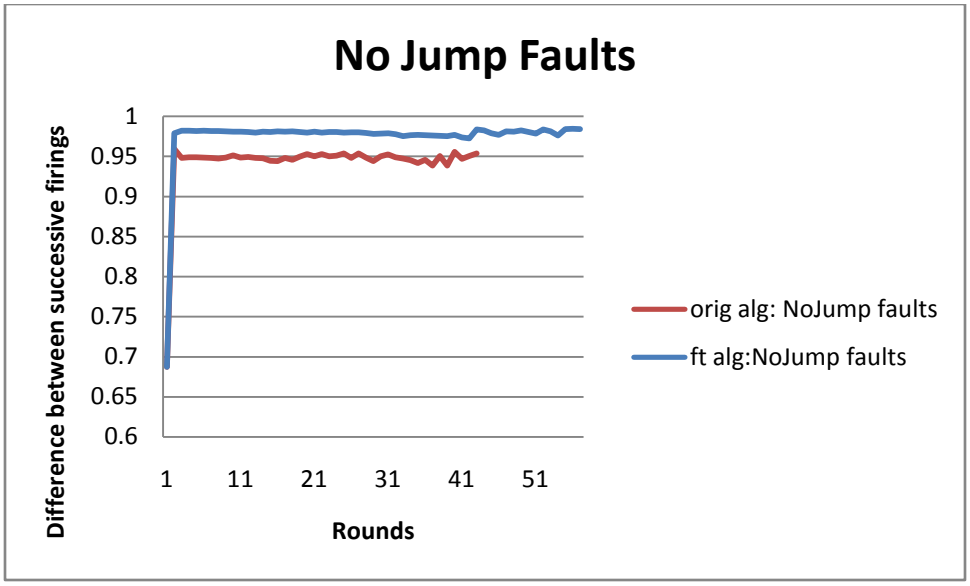
C. Periodicity of Consecutive Firings

In addition to time taken for convergence, another property required for pulse synchronization is the periodic nature of firings. In the original Mirollo and Strogatz model [1], a node follows the entire clock cycle from t=0 to t=T. Therefore once convergence is reached, all nodes are in sync and time difference between subsequent firings is exactly T units. In the RFA algorithm this property is violated, because a node does not start its cycle from t=0. The cycle starts from the adjusted value computed from the previous cycle, hence the length of the cycle is shortened. Therefore periodicity is not achieved in the Reachback Firefly algorithm. From our simulations we have observed the periodicity of firings, for each variant of the algorithm.

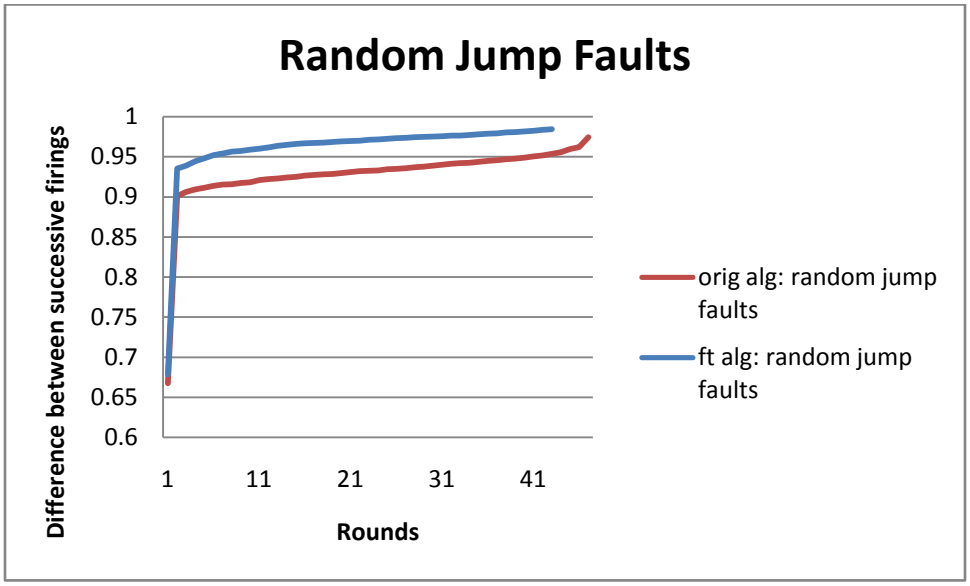


(a) Periodicity with zero faults.

Fig. 7: Periodicity.



(b) Periodicity with No Jump faults.



(c) Periodicity with Random Jump faults.

Fig. 7 (continued)

Periodicity is defined as the time difference between a firing in the n^{th} round and the $n\text{-}1^{th}$ round. The following graphs in Fig. 7 plot the average periodicity of all non faulty nodes measured once every cycle, when n=10, f=3, FFC=100, w=0.1

From the graphs we can see RFA algorithm does not ensure perfect periodicity. In Fig. 7(a) and (b), the periodicity remains around 0.95 for the RFA algorithm with zero faults and No faults. But the fault tolerant algorithm increases the periodicity closer to 1.0 than ever. In Fig 7(c) show periodicity when there are Random Jump faults. Both the RFA algorithm and the fault tolerant algorithm start off badly, but periodicity increases steadily and almost becomes 1 for the fault tolerant algorithm. In general we can conclude that the fault tolerant algorithm improves the periodicity.

D. Discussion

From the above results we can conclude that the original RFA algorithm is not fully fault tolerant. In almost all cases when No Jump faults are introduced, the RFA algorithm takes a longer time to converge. Smaller values of FFC are not exactly good indicators to compare convergence time, because small FFC causes a large jump and cause overshooting the other nodes. The parameter ε (or 1/FFC) is an important factor in determining convergence time. The time taken to converge is inversely proportional to the ε parameter. Hence it would be reasonable to assume that if we pick a large value for ε, it will lead to faster convergence. But in reality large ε causes the nodes to take big jumps and less smooth in going to convergence.

Since this algorithm is inspired from biological systems we can look up to the firefly flashing phenomenon for an understanding of ε parameter. In biological systems the parameter ε translates to the pulse strength of each unit. For example in firefly synchronization, ε is the strength of the pulse emitted by each firefly. When two fireflies have their firings synchronized they emit a pulse of combined strength. The pulse strength also determines the jump taken whenever a firing is heard. A firefly can only emit a pulse of so much strength. Due to physical limitations of the unit taking part in synchronization process, it may not be possible to choose a large ε value.

We have observed that sometimes introducing faults into the system makes it converge faster than the original RFA algorithm. There could be a number of reasons for this odd behavior. Consider when there is only one faulty node in the system and it follows No Jump behavior. The one faulty node becomes the leader and all other non-faulty nodes follow its lead. So it may lead to faster convergence. We have observed similar behavior even sometimes when f>1. A probable explanation for this could be that the faulty nodes were all close together to start with. Hence they were not able to split up the non faulty nodes into unstable clusters.

The above experiments were all performed for random initial states, with faulty nodes placed evenly in between non-faulty nodes. If we choose to start simulation with initial state such that all nodes are equally apart, the results of the experiments could come out differently. Even when the fault model is Random Jump model, the results for convergence time do not seem encouraging.

We can see from the results that periodicity improves for the fault tolerant algorithm. The periodicity is almost close to ideal (0.97~0.98). Thus we can say that the fault tolerant algorithm, though it is slow to converge, gives better periodicity from the very beginning. As of now we have counted as system stability for the nodes to remain within one cluster for the last 9 out of 10 firings. It remains to be seen whether the system can continue in this state forever.

CHAPTER IV

CONCLUSIONS AND FUTURE WORK

A. Conclusions

In this paper we present a new solution to the problem of pulse synchronization in the presence of faulty processes. We propose a fault tolerant algorithm, based on Welch et al. paper [6], to correct the errors introduced by faulty processes. Our algorithm is an extension to the RFA algorithm presented by Werner-Allen et al. in [2]. In this algorithm, each process listens to firings of all other processes. From all the N clock values collected, it throws out f highest and f lowest values and computes the overall jump based on the remaining values. An adjustment is made to its next clock cycle based on this jump computed. We have tested the performance of this algorithm through simulation.

Two kinds of fault models have been analyzed. In the No Jump fault model, faulty processes do not react in response to firings from all other nodes. In the Random Jump faults, faulty processes do not follow any pattern and fire at any random time. Through simulation it was found that fault tolerant averaging function introduces a significant overhead in the time taken to achieve convergence, even when there are no faulty processes. In the presence of Random Jump faults, the effect of averaging function on convergence time is arbitrary. Whereas for No Jump faults, the fault tolerant algorithm takes more time for convergence than when no averaging is performed.

The advantage of using fault tolerant averaging function comes into light when measuring periodicity of the firing interval. For the original RFA algorithm, periodicity is not fixed and spikes up and down. With No Jump and Random Jump faults introduced, the periodicity becomes even more erratic. But when the fault tolerant averaging function is applied, the periodicity curve flattens and ultimately settles down to a constant value.

B. Future Work

In this thesis we have introduced the problem of fault tolerance in the Reachback Firefly algorithm [2] for pulse synchronization. While we have discussed only two fault models, there is scope for more. A cleverer fault model could disrupt the RFA algorithm only in critical stages and remain like a non-faulty node rest of the time. That would make it harder for the fault tolerant algorithm to detect these outliers and prevent the system from converging forever. For instance in the No Jump and Random Jump fault models, faulty processes follow their respective non-conforming behavior throughout their lifetime. A more malicious fault model could be that the faulty processes combines both the above models and varies its behavior with time.

The methods used in this paper for verification of the algorithm are simulation and experimentation. Theoretical analysis of the algorithm is still needed to be done. Werner-Allen et.al paper [2] contains proof of convergence for the RFA algorithm when $N=2$ and $f(t)=ln(t)$. Their proof uses eigenvector decomposition theorem around the concept of a fixed repelling point. There is scope for working out a theoretical proof

along similar lines for our algorithm, perhaps combined with analysis techniques for the fault tolerant averaging functions from [6].

The communication network was assumed to be all-to-all communication model. It would be interesting to relax this assumption and see if the same results hold for an arbitrary network.

REFERENCES

[1] R.M. Mirollo, S.H. Strogatz, “Synchronization of pulse-coupled biological oscillators,” *SIAM J. Appl. Math*. vol. 50, pp. 1645—1662, 1990.

[2] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh and R. Nagpal, “Firefly-inspired sensor network synchronicity with realistic radio effects," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pp. 142-153, 2005.

[3] D. Lucarelli and I. Wang, “Decentralized synchronization protocols with nearest neighbor communication,” in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pp. 62-68, 2004.

[4] C. S. Peskin. *Mathematical Aspects of Heart Physiology*. New York University, New York, 1975.

[5] Y.-W. Hong and A. Scaglione, “Time synchronization and reach-back communications with pulse-coupled oscillators for UWB wireless ad hoc networks,” in *Proc. IEEE Conference on Ultra Wideband Systems and Technologies*, pp. 190–194, 2003.

[6] J.L. Welch and N. Lynch. “A new fault-tolerant algorithm for clock synchronization,” *Information and Computation*, vol. 77(1), pp. 1-36, 1988.

[7] A. Daliot, D. Dolev, H. Parnas, “Linear Time Byzantine Self-Stabilizing Clock Synchronization,” in *Proceedings of 7th International Conference on Principles of Distributed Systems*, pp 7-19, 2003.

[8] A. Daliot, D. Dolev, H. Parnas, "Self-Stabilizing Byzantine Pulse Synchronization," August 2007. [online] Available http://arxiv.org/abs/cs/0608092.

[9] D. Dolev, E.N. Hoch. "On Self-stabilizing Synchronous Actions Despite Byzantine Attacks," in *Proc. the 21st Int. Symposium on Distributed Computing*, pp 193-207, 2007.

[10] L. Lamport and P.M. Melliar-Smith,"Synchronizing clocks in presence of faults," *J.ACM*, vol. 32, pp 52-78, Jan 1985.

[11] M. Maróti, B. Kusy, G. Simon and Á.Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pp 39-49, November 2004.

[12] J. Elson, L. Girod and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[13] S. Ganeriwal, R. Kumar and M.B. Srivastava, "Timing-sync protocol for sensor networks," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pp 138-149, 2003.

[14] D. Dolev, N.A. Lynch, S.S. Pinter, E. W. Stark and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *J. ACM*, vol 33, pp 499-516, May 1986.

[15] B. A. Coan, D. Dolev, C. Dwork and L. Stockmeyer. "The distributed firing squad problem," in *Proceedings of the Seventeenth Annual ACM Symposium on theory of Computing*, pp 335-345, May 1985.

[16] A. Daliot, D. Dolev and H. Parnas, "Self-stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks," in *Proc. of the 6th Symposium on Self-Stabilizing Systems*, pp. 32-48, 2003.

APPENDIX A

ANALYIS OF MIROLLO & STROGATZ MODEL

This appendix shows another way of explaining the working of Mirollo and Strogatz algorithm when N=2. The proof of convergence for the specific case of 2 nodes and general case of n nodes has been developed in [1]. The contribution here is to show that in the case of N=2, the two nodes always fire alternately and do not overtake each other. Since the jump computed is proportional to ε, we show that there must be an upper bound on ε if we don't want nodes to overshoot each other.

A. Illustration of M&S algorithm when N=2

Consider nodes A and B that have duration timers that exactly match the passing of real time. Every time a duration timer reaches some value T (which can also be normalized to 1), the process "fires". Every time a process fires, it sets its duration timer to some value $\Delta \in [0,T)$.

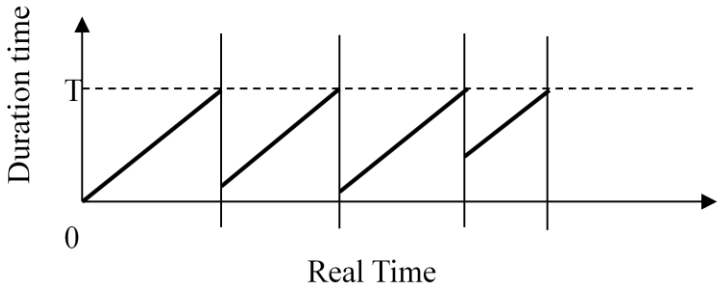


Fig. 8: Duration time vs. Real time.

We want to calculate Δ so that A and B can fire at the same time. Assume A and B can observe each other firing with no delay.

Algorithm Schema:

A computes Δ at real time t (a firing time), to be a function of the time when B fired most recently. Since A has only the duration timer, it records the time on its duration timer. Assume that initially each of them record 0 as the last firing time of the other. Δ(φ) is computed with the help of another function f, which must be smooth, monotonic, increasing and concave down.

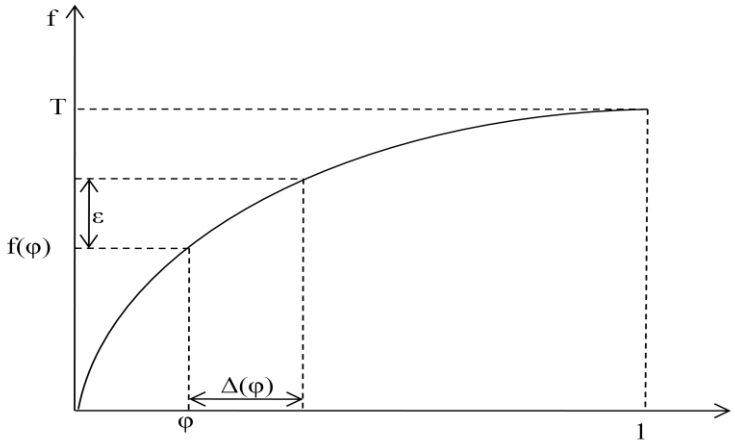


Fig. 9: Function f is smooth, monotonic increasing and concave down.

$$\Delta(\varphi) = f^{-1}(f(\varphi) + \varepsilon) - \varphi$$

So if φ is closer to the beginning of the phase, then Δ is smaller.

For simplicity and concreteness in the following analysis, let

$$f(\varphi) = ln(\varphi).$$

$$f^{-1}(x) = e^{x}.$$

Then

$$\Delta(\varphi) = e^{\ln(\varphi)+\varepsilon} - \varphi = \varphi e^{\varepsilon} - \varphi = (e^{\varepsilon} - 1)\,\varphi.$$

This can be approximated as $\Delta(\varphi) \approx \varepsilon\varphi$.

B. Showing that nodes A and B fire alternately

We want to show that A and B always fire alternately. Without loss of generality assume that A fires first. Let t_{2i-1} be the real time of the i^{th} firing by A and t_{2i} be the real time of i^{th} firing by B, $i \geq 1$. Note that $t_k \leq t_{k+2}$, $\forall\, k \geq 1$.

Lemma: $\forall\, k \geq 1$, $t_k \leq t_{k+1}$, i.e. A and B are alternately firing.

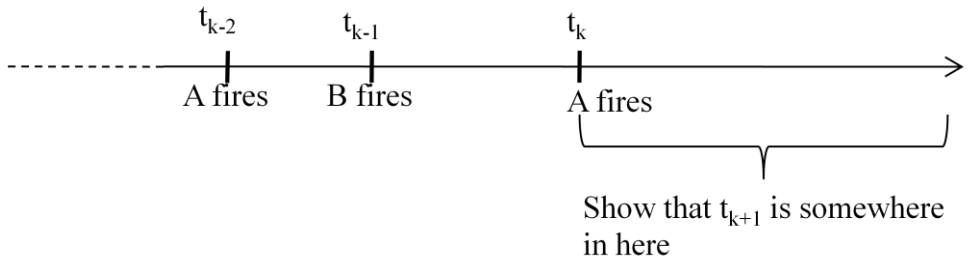
Proof: By strong induction on k.

Basis: k=1, show $t_1 \leq t_2$. This follows from assumption that A fires first.

k=2. We must show that $t_2 \leq t_3$. By assumption of above initialization, since at time t_1, A has no information about B's firing, it sets Δ to 0. Thus $t_3 = t_1 + T$. Since T is the maximum inter-firing time, B must fire somewhere in [t1, t3], so $t_2 \leq t_3$.

Induction: Show $t_k \leq t_{k+1}$, for all $k \geq 3$, assuming $t_i \leq t_{i+1}$, for all $i < k$.

Without loss of generality, assume k is odd (so t_k is a firing by A).



We must show that the jump calculated by B at t_{k-1}, based on hearing A fire at real time t_{k-2}, is less than A's timer at t_{k-1}.

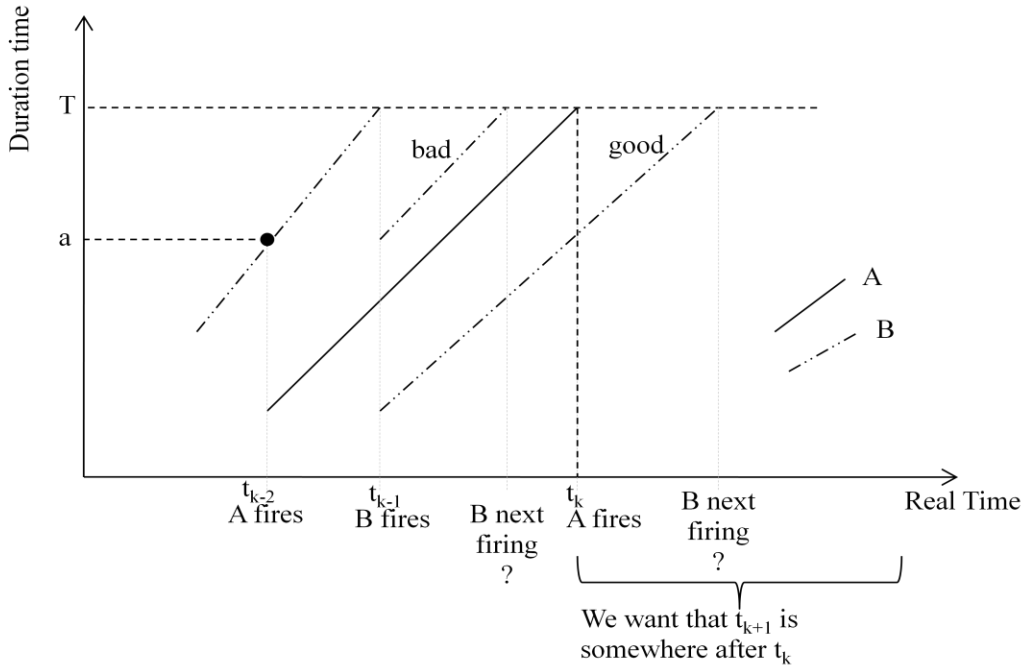


Fig. 10: Illustration of A and B firings in real time.

The value of A's timer at real time t_{k-1} is (t_{k-1} - t_{k-2} + Δ), where Δ is the jump calculated by A at t_{k-2} . Let a be the value of B's timer at t_{k-2} (when B observes A's firing). The jump calculated by B at t_{k-1} is $(e^{\varepsilon} - 1)\frac{a}{T}$. Fig. 9 shows the sequence of A and B's firings. If the jump computed by B at t_{k-1} is greater than A's timer at t_{k-1}, then extrapolating B's curve we can see that B will fire again before A, hence leading to out-of-turn firings (bad behavior indicated in Fig. 9). Therefore we need that the jump computed by B at t_{k-1} is less than A's timer at t_{k-1} (good behavior curve from Fig. 9). So we need that $(e^{\varepsilon} - 1)\frac{a}{T} < t_{k-1} - t_{k-2} + \Delta$.

Note that $t_{k-1} - t_{k-2} = T - a$ (look at Fig. 9)

So we need a and ε such that

$$(e^{\varepsilon} - 1)\frac{a}{T} < T - a + \Delta . \qquad (1)$$

Recall that $0 \leq \Delta \leq T$. So if we can show that $(e^{\varepsilon} - 1)\frac{a}{T} < T - a$ (2)

Then (2) will ensure (1).

If we normalize T to be 1. Then we need to show that $(e^{\varepsilon} - 1)a < 1 - a$, where a is between 0 and 1. This impliese$^{\varepsilon} < \frac{1}{a}$.

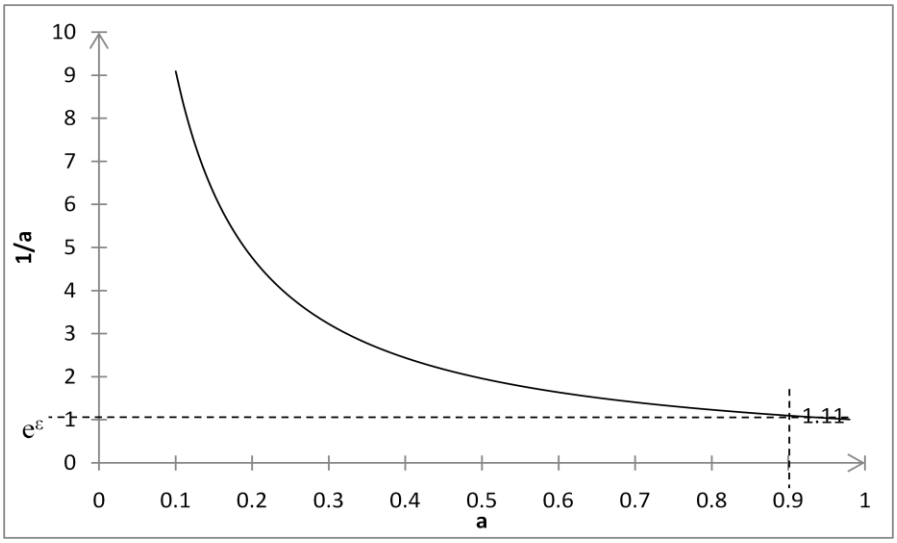


Fig. 11: Plot of *a* vs. *1/a.*

We can ensure this by setting ε small enough, so that a computed is never very large. For instance if $\varepsilon = \frac{1}{10}$, then $e^{\varepsilon} \approx 1.11$. This crosses the plot of 1/a when a is about 0.9.

So if process A observes the other process B firing very close to the end of its own cycle, instead of setting its jump very large so that it will fire again before the other

process B, A sets its phase to either 1 or the value after the jump, whichever is smaller. If process A reaches the threshold 1, it will fire immediately and go back to 0. Therefore at most A and B could start again at 0, but A will not overtake B.

If we use the approximation for $(e^{\varepsilon} - 1) \approx \varepsilon$

From (1) it follows that we have to ensure that $\varepsilon \frac{a}{T} < \mathrm{T} - \mathrm{a} + \Delta$.

Since $0 \leq \Delta \leq \mathrm{T}$, it is sufficient to show that $\varepsilon \frac{a}{T} < \mathrm{T} - \mathrm{a}$

Normalize T to be 1. Then $\varepsilon a < 1 - \mathrm{a}$

i.e. $\varepsilon < \frac{1-\mathrm{a}}{\mathrm{a}}$ or $a < \frac{1}{1+\varepsilon}$

Hence the lemma holds as long as $a < \frac{1}{1+\varepsilon}$. If ε=0.1 then a < 0.99 □

APPENDIX B

PROGRAM CODE

The following is the program code for simulating RFA algorithm and our fault tolerant averaging function.

Class representing a node process:

```
public class Oscillator implements Comparable
{
	public int nodeId;
	public double phase;
	public double pulse;
	public boolean faulty = false;
	public double internalPhase;
	public double lastFiring;
	public ArrayList<Double> firingsHeard;


	public Oscillator(int n, double phi, double epsilon)
	{
		nodeId=n;
		phase=phi;
		pulse = epsilon;
		internalPhase = phi;

		firingsHeard = new  ArrayList<Double>(n);
	}

	public int compareTo(Object anotherOsc) throws
ClassCastException {
    if (!(anotherOsc instanceof Oscillator))
      throw new ClassCastException("A Oscillator object expected.");

    if ( this.phase < ((Oscillator)anotherOsc).phase)
      return -1;
    else
      return 1;
  }
}
```

Main function which implements RFA logic:

```
public static int reachback(double[] phase)
{
	/* Creates an Array List structure to store the clock values.
	Makes it easy for sorting, dequeing */
	ArrayList<Oscillator> queue = new
	ArrayList<Oscillator>(numNodes);

	for (int i = 0; i < numNodes; i++)
	{
		Oscillator osc = new Oscillator(i, phase[i], epsilon);
		queue.add(osc);
		/* Assign particular nodes as faulty processes */
		if (assignFaulty(i)) osc.faulty = true;
	}

	/*Reverse sort the nodes*/
	Collections.sort(queue);
	Collections.reverse(queue);
	int iterations = 0, globalTime = 0.0;

	/* Main loop of the algorithm*/
	while (true)
	{
		/*Measure group spread in each round*/
		double gs=groupSpread(queue);
		if (gs < window)
			lastCount++;
		else
			lastCount = (lastCount==0)? 0 : (--lastCount);
		/*if group spread remained within window for last 9
rounds.
		Exit condition for the main while loop */
		if (lastCount >= 9)
			break;

		/* Remove head of the queue */
		Oscillator head = queue.remove(0);
		double phi = head.phase;

		/*Add (1-phi) to remaining nodes */
		for (Iterator it = queue.iterator(); it.hasNext(); )
		{
			Oscillator p = (Oscillator)it.next();
			p.phase += (1 - phi);
			p.internalPhase += (1 - phi);
		}

		globalTime += (1 - phi);
		print_v(globalTime + " : n"+head.nodeId+" fires");
```

(cont….)

```
            /*remaining nodes jump by epsilon */
            for (int i = 0; i < queue.size(); i++)
            {
                  Oscillator p = queue.get(i);
                  //Make a note of the time at which firing was heard
                  p.firingsHeard.add(p.phase);

                  double jump =
                        calculateJump(p.faulty, p.internalPhase);
                  //if firing has brought this oscillator to
                  threshold
                  if (jump >= 1.0)
                        p.internalPhase = 1.0;
                  else
                        p.internalPhase = jump;
            }

            /* Compute the fault tolerant average for the head node
            and set the new phase of the head to the overall jump*/
            head.phase = FT_Avg(head);

            //reset variables and add head to the end of the queue
            head.firingsHeard.clear();
            queue.add(head);

            iterations++;
            Collections.sort(queue);
            Collections.reverse(queue);
      }
      return iterations;
}
```

Function to compute fault tolerant average:

```
public static double FT_Avg(Oscillator osc)
{
	/* Faulty nodes donot perform averaging function */
	if (osc.faulty)   return 0.0;
	/* f is the upper bound on number of faulty nodes */
	int f = (int)Math.ceil(((double) n)/3) - 1 ;

	//Step 1: find max distance between consecutive firings heard
	/* partition indicates the end point, padding is the mid value
	between end points */
	int partition = 0;
	double padding = 0.0, prevValue = 0.0, maxdistance = 0.0;
	for (int i = 0; i < osc.firingsHeard.size(); i++)
	{
		double currValue= osc.firingsHeard.get(i).doubleValue();
		double distance = prevValue - currValue;
		if (distance > maxdistance)
		{
			maxdistance = distance;
			partition = i;
			/*select midpoint as the padding value*/
			padding = (prevValue + currValue) / 2;
		}
		prevValue = currValue;
	}

	/*wrap-around case: prev=last non-faulty node, queue.get(nf)=
	first non-faulty node */
	double lastValue = prevValue;
	double firstValue = osc.firingsHeard.get(0).doubleValue();
	double distance = 1 - (lastValue - firstValue);
	if (distance > maxdistance)
	{
		maxdistance = distance;
		partition = 0;
		padding = lastValue + distance / 2;
		if (padding > 1.0) padding = padding - 1.0;
	}

	/*Step 2: padding
	//If there are less than n values heard, pad up the in the
	center of the queue with midpoint values */
	int deficit = 0;
	while (osc.firingsHeard.size() < numNodes)
	{
		osc.firingsHeard.add(partition, padding);
		deficit++;
	}
	Collections.sort(osc.firingsHeard);
	/*move partition to middle of padded values*/
	partition += deficit / 2;                                (contd….)
```

```
	/* Step 3: throwing out f highest and f lowest values.
	Partition is the end point, so f values on either side of
	partition to be thrown out*/
	double currentPhase=0.0, delta = 0.0, sumdelta=0.0;

	/* Case 1: not enough f values found on the left side of
	partition index*/
	if( (partition -f) < 0 )
	{
		/* Start adding up from f values right of partition*/
		for (int i = partition + f; i < (n + partition - f); i++)
		{
			double phase =
		osc.firingsHeard.get(i).doubleValue();
			currentPhase = phase + sumdelta;
			delta =
		calculateJump(osc.faulty, currentPhase) - currentPhase;
			sumdelta += delta;
		}
	}

	/* Case 2: not enough f values found on the right side of
	partition index*/
	else if ((partition + f - 1) >= n)
	{
	/* Start from beginning after leaving out wrap around f lowest
	values */
	for (int i = partition + f - n; i < (partition-f); i++)
	{
		double phase = osc.firingsHeard.get(i).doubleValue();
		currentPhase = phase + sumdelta;
		delta =
		calculateJump(osc.faulty, currentPhase) - currentPhase;
		sumdelta += delta;
	}

	/* Case 3: enough f values found on both left and right side of
	partition index*/
	else
	{
		for (int i = 0; i < n; i++)
		{
			double phase =
			osc.firingsHeard.get(i).doubleValue();
			if ((i < (partition - f)) || (i > (partition + f -
1)))
			{
				currentPhase = phase + sumdelta;
				delta =
		calculateJump(osc.faulty, currentPhase) - currentPhase;
				sumdelta += delta;
			}
		}
	}
	return sumdelta;
}
```

Function to compute group spread

```
static double groupSpread(ArrayList<Oscillator> queue)
{
	double maxdistance = 0.0;
	int nf = 0;

	/* Step 1: find out first non-faulty node */
	for (nf = 0; nf < queue.size(); nf++)
	{
		Oscillator curr = queue.get(nf);
		if (!curr.faulty)
			break;
	}

	/* Step 2: start measuring distance between adjancent non-
	faulty nodes */
	Oscillator prev = queue.get(nf);
	for (int i = nf + 1; i < queue.size(); i++)
	{
		Oscillator curr = queue.get(i);
		if (!curr.faulty)
		{
			double distance = prev.phase - curr.phase;
			if (distance > maxdistance)
				maxdistance = distance;
			prev = curr;
		}
	}

	/* Step 3: wrap-around case: prev=last non-faulty node,
	queue.get(nf)= first non-faulty node */
	Oscillator first = queue.get(nf);
	if ((first.phase > prev.phase))
	{
		double distance = 1 - (first.phase - prev.phase);
		if (distance > maxdistance)
			maxdistance = distance;
	}

	/* Step 4: Group Spread = (1-maxdistance) */
		return (1 - maxdistance);
}
```

VITA

Name:	Keerthi Deconda
Permanent Address:	Chaitanya Hospital Mukurampura Karimnagar, Andhra Pradesh, 505001 India
Email Address:	keerthi.deconda@gmail.com
Education:	B.Tech., Computer Science and Engineering, National Institute of Technology, Warangal, India, 2004 M.S., Computer Science, Texas A&M University, 2008