Fast Wait-free Symmetry Breaking in Distributed Systems

by

Mark Anthony Shawn Smith

B.S., Computer and Information Science
Brooklyn College
(1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Signature of Author____________________________________
Department of Electrical Engineering and Computer Science
May 14, 1993

Certified by____________________________________
Baruch Awerbuch
Assistant Professor of Mathematics
Thesis Co-supervisor

Certified by____________________________________
Nancy Lynch
Professor of Computer Science
Thesis Co-supervisor

Accepted by____________________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Fast Wait-free Symmetry Breaking in Distributed Systems

by

Mark Anthony Shawn Smith

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This work redesigns Luby's parallel Maximal Independent Set (MIS) algorithm at the atomic level, so that it no longer needs to depend on rounds. Thus, we present a protocol for constructing an MIS in an entirely asynchronous environment where processes are added dynamically. Using existing synchronizer methods would require $O(D)$ time overhead to adapt the synchronous protocols of Karp-Wigderson or Luby, where D is the diameter of the network when no more processes are added. We believe our protocol converges to an MIS in $O(\log n)$ expected time, which would beat the best know results. However, calculating the precise probabilities in the asynchronous environment has proved to be extremely difficult, so our protocol converges in $O(\log n)$ expected time subject to the proof of a conjecture that we make in the thesis.

We also extend the traditional definition of *wait-freedom* for the shared memory model of distributed computing, to capture important performance and fault tolerance metrics in the message passing model. Our definition formalizes the intuitive notion that a process should not be stopped or slowed by faulty processes/links that are far away.

We also show that our protocol for the dynamic MIS problem is 2-wait-free, which, by our definition of k-wait-free, means a processor only has to wait for its neighbors' neighbors in order to make progress– a slow link or failed process further than distance 2 away in the graph will not locally slow down the protocol.

Thesis Co-supervisor: Baruch Awerbuch
Title: Assistant Professor of Mathematics

Thesis Co-supervisor: Nancy Lynch
Title: Professor of Computer Science
Keywords: symmetry breaking, k-wait-freedom, maximal independent set, dynamic maximal independent set, distributed computing.

[0]This research was supported in part by a Bell Labs CRFP fellowship and DARPA grant N00014-92-J-1799.

Acknowledgments

First of all I would like to thank my thesis co-supervisor, Baruch Awerbuch, who suggested I work on this problem, and who provided me with valuable intuitive insights on the problem. Next I want to thank my other co-supervisor Nancy Lynch who was instrumental in my efforts to formalize the model of computation, the statement of the problem, and the proofs in this thesis. I would also like to thank Lenore Cowen with whom, along with my Baruch Awerbuch, I did joint work on preliminary results of this thesis.

Be Hubbard the TOC group secretary has helped me with the little things that have made my existence in the TOC group much easier. For this I thank her. I am also eternally grateful to my officemate Mojdeh Mohtashemi, who has been a wonderful and supportive friend. She has always been there when I needed a word of encouragement or a sympathetic ear.

Most importantly, I would like to thank my family, and in particular my mother. They have made all my endeavors possible.

Contents

Chapter 1

Introduction

1.1 Motivation

Symmetry breaking. Symmetry breaking is the problem of making a choice between objects that are essentially all alike. This problem is fundamental in distributed computing. When one can break symmetry in a network, one is able to resolve deadlocks [8], the chosen process takes the next step; elect a leader [1], the chosen process is the leader; achieve mutual exclusion [12], the chosen process gets access to the critical region; and allocate resources [10], the chosen process gets to use the resources.

The goal of this thesis is to design efficient symmetry-breaking algorithms in a realistic distributed system, where we want to to be able to tolerate stopping faults (faults where a process stops operating). In other words, we want the protocol to have the property of k-wait-freedom, which means the protocol tolerates ongoing faults in the sense that if some processes (links) further that distance k away from a process i stop, i continues to run the protocol at the same speed. That is, i does not depend on slow processes (links) further than distance k away.

We elaborate on this issue below.

Wait-freedom. Wait-freedom is an important property of distributed algorithms that has been traditionally defined in the shared memory model of distributed computing. In that model, an algorithm is said to be wait-free if processes will complete their operations regardless of the failure of other processes. That is, a process can only be delayed in the completion of its job by the the speed of the shared memory and the speed of its link to the shared memory.

We will define the same term for the message passing model of distributed computing in a way that we think captures an important metric for performance and fault tolerance in this model. Intuitively, we say an algorithm is *k-wait-free* in the message passing model, if progress of an individual process is only dependent on information from other processes distance k away. (We define this notion more formally in chapter 2). Our goal is to make k as small as possible. We think this is an important performance metric

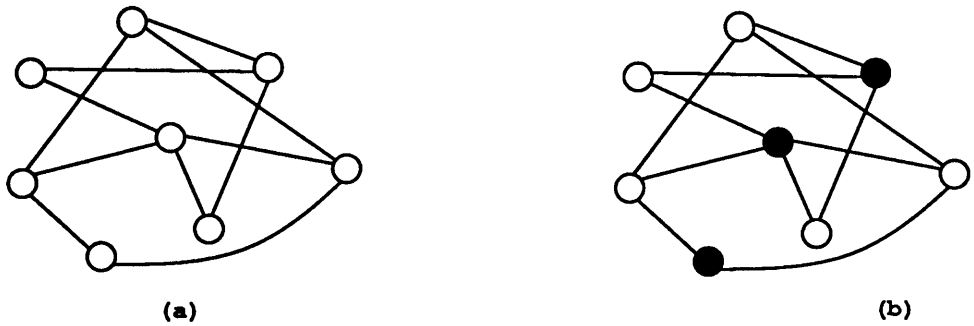

Figure 1-1: **The MIS problem.** (a) a graph (b) the shaded nodes form an MIS of the graph

because it means a process only has to wait at most k times the maximum link delay before any step in its execution. It is also a fault tolerance metric because it means that a process can only be affected by faults that are at most distance k away. Another way to look at k-wait-freedom is that it captures the notion that the operation of a process should not be slowed or stopped by a very slow or failed process or link that is greater than distance k away. If a protocol is k-wait-free, we say it has *wait-dependency k*.

1.2 The Problem

Our symmetry breaking tool will be a variation of the Maximal Independent Set (MIS) problem. We consider the underlying graph of the network to be the input graph, where the nodes are network processes, and two processes are connected by an edge if there is a direct communication link between them. An MIS on this graph is a subset of nodes such that no two nodes are connected and every process is either in this subset or has a neighbor in the subset (see Figure 1-1).

In the variation of the problem we consider, the graph is allowed to grow dynamically, that is, the graph grows by the activation of new processes to participate in the protocol. When new processes get activated, they are given the set of already active processes to which they are connected (their neighbor set). The links to these neighbors are also activated. Note that the activation of new links between already active processes is not allowed. Our system will be modeled using a probabilistic timed I/O automata. We give a formal presentation of the model and the problem in Chapter 2.

1.3 Previous results

Randomized solutions for the distributed MIS problem which run in logarithmic expected time have been found by Karp-Wigderson [15] and Luby [16], *in a properly-initialized synchronous model*. The protocols of Karp-Wigderson and Luby run in set rounds, which assumes the existence of a global clock and a consistent initial state. In the *asynchronous* model, if the network graph is static and all the processes "wake up" at the same time then the best known algorithms for constructing an MIS use the α "synchronizer" introduced by Awerbuch [5] and the algorithm of [15] or [16]. The α synchronizer generates pulse numbers which are used to simulate rounds. In the case where the network is growing dynamically the α synchronizer may add a delay of $O(D)$ to the protocol, where D is diameter of the network when it stabilizes (no new processes are added). Therefore, a protocol that uses the α synchronizer plus the protocol of Luby or Karp-Wigderson, would run in $O(D + \log n)$ expected time. This added delay occurs because the dynamic addition of processes may cause a waiting chain of length $O(D)$ in the protocol. That is, a process may have to wait for a synchronizing pulse to propagate from a process distance $O(D)$ away. This also mean if the process at the start of the chain fails, the process at the end of the chain cannot complete its execution. We elaborate on these issues in Chapter 3.

1.4 Our results

We present a new dynamic MIS algorithm for symmetry-breaking in an asynchronous network. We consider the time it takes for our protocol to construct an MIS after the network has stabilized, that is, when the last process is activated. At that time, if there are n active processes in the network, then we believe our protocol will converge to a correct MIS in $O(\log n)$ expected time. However, the proof of the expected running time we give in the thesis is subject to the proof of a conjecture we make about the probabilistic behavior of the protocol. Our algorithm is also 2-wait-free. This means any process i will tolerate the faults of any other processes greater than distance 2 away. Processes are only dependent on their neighbors and their neighbors' neighbors for their operations.

1.5 Structure of thesis

We present the formal network model and a more precise definition of the problem in chapter 2. In chapter 3, we give the asynchronous protocol for the dynamic MIS problem and the proof of its correctness. We also prove safety and liveness properties of the protocol in this chapter along with an analysis of the time complexity of the algorithm. Lastly we give the proof that the algorithm is 2-wait-free. Chapter 4 contains a discussion of related work and open problems.

Chapter 2

Model and Problem Statement

2.1 Introduction

We assume that the communication network is a complete graph. The active graph will be a subgraph of this graph, and our protocol runs on the active graph. We model each process as a *probabilistic timed* I/O automaton (we give a formal definition below). Each process has an input action from an external system that activates the process (WAKEUP message). The input action message has the neighbor set of the newly activated process and a value that may be used in the protocol. A process that has received a wakeup message as an input we will define as *active*. We restrict the environment such that in the set of legal inputs, the neighbor set included in the input must consist only of active processes. Thus, the first process to be activated will receive a null neighbor set in its input.

When a process gets activated, it sends a message to its neighbors telling them it has joined the protocol. The communication links between a process and its neighbors get activated when the process receives its WAKEUP message with its neighbor set. Communication is only allowed between active processes.

Each process has an external action that has a value that is the result of it running the protocol. The output set produced by active processes must have the property that the activation of new processes does not cause old processes to change their outputs.

2.2 The formal model

In this section we give a formal presentation of our model. We start by giving an overview of I/O automata and timed I/O automata. Finally we present probabilistic timed I/O automata.

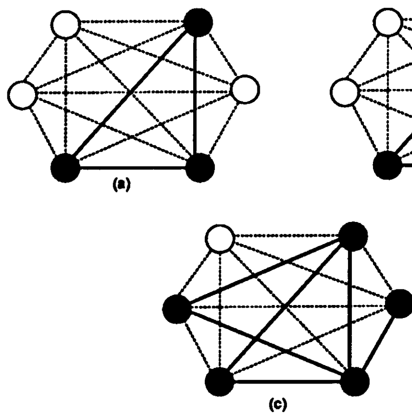

Figure 2-1: **The dynamic MIS problem** The figure shows how the active graph might grow on a network of 6 processes. The solid lines represent active links and the shaded nodes represent active processes. In (a) we have 3 active processes, (b) shows the activation of a fourth process, and (c) has the activation of a fifth process. If the network stabilized after (c), then $n = 5$.

2.2.1 I/O automata

We define I/O automata and give a brief description of some of its properties. For a more detailed description of the model see [20, 19]. An I/O automaton consists of five components:

1. an action signature *sig(A)*,
2. a set *states (A)* of *states*,
3. a nonempty set *start(A)* $\subseteq$ of *states*,
4. a transition relation *steps(A)* $\subseteq$*states(A)* $\times$ *acts(A)* $\times$ *states(A)* with the property that every state s' and input action π there is a transition (s', π, s) in *steps(A)*, and
5. an equivalence relation of *part(A)* partitioning the set *local(A)* into at most a countable number of equivalence classes.

The action signature *sig(A)* is a partition of a set *acts(S)* into three disjoint sets *in(S)*, *out(S)*, and *int(S)* of *input actions*, *output actions* and *internal actions* respectively. The union of the *input actions* and *output actions* we denote as *external actions*, those actions visible to the environment of any automaton that has S as its action signature. Each element of an automaton's transition relation represents a possible step in the computation of the system the automaton models. We refer to an element (s', π, s) of *steps(A)* as a *step*. The equivalence relation *part(A)* is used to identify the primitive components of the system being modeled by the automaton: each class is thought of as the set of actions under the local control of one system component.

When an automaton 'runs', it generates a string representing an execution of the system the automaton models: an *execution fragment* of A is a finite sequence $s_0, \pi_1, \ldots, \pi_n, s_n$

or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i. An *execution* is an execution fragment beginning with a start state. The *schedule* of an execution α is the subsequence of α consisting of all the actions of α. The *behavior* of an execution or schedule α is the subsequence of α consisting of external actions.

We can model complex systems by composing automata modeling the simpler system components. We can compose automata if they are *strongly compatible*; this means that no action can be an output of more that one component, that internal actions of one component are not shared by any other component, and that no action is shared by infinitely many components. The results of such a composition is another I/O automaton.

2.2.2 Timed automata

In this subsection we summarize the description of timed automata as presented in [18]. A *boundmap* for an I/O automaton A is a mapping that associates a closed interval of $[0, \infty]$ with each class in *part(A)*, where the lower bound of each interval is not ∞ and the upper bound is nonzero. Intuitively, the interval associated with a class C by the boundmap represents the range of possible lengths of time between successive times when C "gets a chance" to perform an action. A *timed automaton* is a pair (A, b), where A is an I/O automaton and b is a boundmap for A.

In the timed automata model we have notions of "timed execution", "timed schedule", and "timed behavior" that corresponds to executions, schedules, and behaviors for ordinary I/O automata. These will all include time information. A timed sequence is the basic type of sequence that underlies the definition of a timed execution. A timed sequence is a finite or infinite sequence of alternating states and (action, time) pairs, $s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots$, satisfying the following conditions:

1. The states $s_0, s_1, \ldots$ are in *states(A)*.
2. The actions $\pi_1, \pi_2, \ldots$ are in *acts(A)*.
3. The times $t_1, t_2, \ldots$ are successively nondecreasing nonnegative real numbers.
4. If the sequence is finite, then it ends in a state s_i.
5. If the sequence is infinite then the times are unbounded.

2.2.3 Probabilistic timed I/O automata

In this subsection we briefly present probabilistic timed I/O automata. Probabilistic timed I/O automata is essential the same as timed I/O automata, except instead of the steps being deterministic base on the current state and the actions enabled in that state, there is now a probability distribution over the next possible steps. In the probabilistic timed model we also have an adversary that has power over which step is taken next in an execution sequence.

More formally, probabilistic timed I/O automata is a triple (A_p, b, a) where A_p is a probabilistic I/O automaton, b is a boundmap for A_p and a is an adversary. A probabilistic I/O automaton differs from an I/0 automaton in the definition of the transition relation. For probabilistic I/O automata we have transition relation $steps(A_p) \subseteq states(A_p) \times acts(A_p) \times states(A_p)$ with the property that for every state s' and action π enabled in s' there is a set of states T with probability distribution $Q_{s',\pi}$ over the set such that there is a transition (s', π, s) in $steps(A_p)$ for every $s \in T$.

When a probabilistic automaton 'runs', it generates a tree, the *execution tree,* representing an execution of the system the automaton models. Nodes in the tree are states and the branches represents the different states that can be reached when some action that is enabled in that state is taken. A branch will happen with probability based on the probability distribution defined on that state and action in $steps(A_p)$. An *execution fragment* of A_p is a path in the *execution tree* and is defined as in I/O automata. *Execution, schedule* and *behavior* are also based on paths in the execution tree and are defined as in I/O automata.

To find the probability that a particular event (state or action) happens in some finite *execution* of the automaton, we look at the *execution tree* generated by the automaton, find every path on which this event occurs and sum the probabilities of the event on each path. To find the probability of an event on a path, we take the product of the probability of each successive state in the path starting with the start state.

2.2.4 Our system

Our network is represented as a directed graph $G = (V, E)$ where the nodes of the graph, V, are the processes, and the edges E are the communication channels between processes. We define Neighbors(i) as the set of Neighbors of a process $i \in V$. We model each process $i \in V$ as a probabilistic timed IOA, A_i. However, we restrict our adversary a to what we will call the link delay function. a takes as its argument a link and the global time and returns the delay on that link. The link delay, d is bounded from below by 0 and from above by the maximum delay ν.

Each automaton A_i will have the following action signature.

Input actions:	WAKEUP$_i(N, v)$ (where v some input value and $N \subseteq$ Neighbors(i))
Output actions:	RETURN(v) (output value)

We place the following well-formedness constraint on the environment, in our model: $\forall$ WAKEUP$_i(N, v)$ and $\forall j \in N$ there was a WAKEUP$_j$ earlier in the execution sequence.

We model the communication channels as IOA. Each process has a communication channel to all its neighbors $\in V$. However, we constrain a process i and only allow it to communicate with a neighbor j when i receives the input $\text{WAKEUP}_i(N, v), j \in N$ or i receives a message from j. Each channel automaton has input actions of the form SEND_j(M) and output actions of the form RECEIVE(M). The transition relation is as follows:

SEND(M)
  Effect: messages $\leftarrow$ messages $\cup\{M\}$

RECEIVE(M)
  Precondition: M $\in$ messages
  Effect: messages $\leftarrow$ messages - $\{M\}$

2.3 The problem specification

In our model the active network is allowed to grow dynamically. A process may get new neighbors at anytime during an execution. However, if a process has already produced an output, we do not want this result to be invalidated by the addition of new processes. Problems solved in this model must have this property. We call the property *dynamic extendability*. Let I be the input set for the problem and O be the output set. For a graph problem each element $i_I \in I$ is a triple (i, N_i, v_i) and $i_O \in O$ is a pair (i, v_i') where i is the process id, N_i is the neighbor set of process i, v_i is an input value, and v_i' is an output value. We define *proc(I)* as the set of processes that are in the triples of the set I. I and O grow dynamically as the network grows. A problem P is a set of specifications that define a relationship between v_i and v_i' for every process i and also defines a relationship on v_i' for all i. A solution S to problem P is a set of pairs, where each pair is the input and output tuples for the same process, that satisfies the specifications of P. Formally we say a graph problem is *dynamically extendable* if given some problem P and a solution set S for P, then $\forall j \notin$ *proc(I)* (a new input), such that $N_j \subseteq$ *proc(I)*, and $\forall v_j$, $\exists v_j'$ such that $S \cup \{((j, N_j, v_j), (j, v_j'))\}$ solves P.

2.3.1 The MIS problem

An MIS on a graph is a subset of nodes such that no two nodes in the subset are connected (independence) and every node is either in this subset or has a neighbor in the subset (maximal). If at some time t, new processes stop entering the network, then our protocol should produce an MIS on the active network. For this problem there is no value included in the neighbor set in the WAKEUP message, so an element $i_O \in I$ looks like $(i, N_i, \_)$. For notational convenience we write this as (i, N_i). An element of $i_O \in O$ will be $(i, 0)$ or $(i, 1)$ where the 0 output means the process has a neighbor in the MIS or is not in the MIS, and the 1 output means the process is in the MIS. The MIS problem P_{mis}, has the following specifications:

1. if $(i, N_i) \in I$ and $(i, 1) \in O$, then $\forall j \in N_i$, $(j, 0) \in O$ (independence), and

2. let $|I| = n$ and $m =$ the number of processes i such that $(i, 1) \in O$. Then $|\bigcup^i N_i| = n - m$ (maximality).

Lemma 2.1 *The MIS problem is dynamically extendable.*

Proof. We can prove this lemma using induction on the number of active processes. We assume we have a protocol that given input set I produces an output set O such that the solution set S satisfies P_{mis}. The base case is the empty network where we have no active processes, thus, S is the empty set. The addition of a new process k will not violate the ***dynamic extendability*** property because with the addition of the process we have the solution $S = \{((k, N_k), (k, 1))\}$ which satisfies P_{mis}. Now assume we have $|I| = n \geq 1$ and the corresponding O such that S solves P_{mis}. Let $j_I = (j, N_j)$ be the input of some newly activated process j such that $j \notin \textit{proc(I)}$ and $N_j \subseteq \textit{proc(I)}$, then we have two cases based on j's neighbors.

1. $\exists k \in N_j$ such that $(k, 1) \in O$.
   We claim $S \cup \{((k, N_k), (k, 0))\}$ solves P_{mis}.
   Property 1 still holds because by the inductive hypothesis the property was valid before the addition of process j and since $j_O = (j, 0)$, the property will remain valid for any process $k \in N_j$.
   Property 2 holds because $|I|$ increases by 1 with the addition of j, m remains the same and $|\bigcup^i N_i|$ increases by 1 because j is in this set.

2. $\forall k \in N_j, k_O = (k, 0)$.
   We claim $S \cup \{((j, N_j), (j, 1))\}$ solves P_{mis}
   Property 1 holds because $\forall k \in N_j$, we have $k_O = (k, 0)$ so the property holds for j and by the inductive hypothesis it holds $\forall i \in \textit{proc(I)}$.
   Property 2 holds because $|I|$ increases by 1 with the addition of j, m also increase by 1 because j will become a member of this set, and $|\bigcup^i N_i|$ remains unchanged.
   □

2.4 k-wait-freedom

We say that a protocol is k-wait-free if for any process i, if all the processes in the distance k neighborhood of i stop receiving new neighbors and continue to take steps, then i will accomplish its task, that is, i will produce an appropriate output value.

Chapter 3

The MIS protocol

In this chapter we present the asynchronous dynamic MIS protocol. We first review the elegant synchronous protocol of Luby [16], and discuss the difficulties in simulating such a protocol in an asynchronous environment. In the rest of the chapter we give the analysis for expected time and wait-dependency.

3.1 Review of Luby's protocol

Luby's synchronous MIS protocol, as given in [16] proceeds in rounds. In each round, process i flips a coin c_i, where

$$c_i = \begin{cases} 1 & \text{with probability } 1/(2d(i)) \\ 0 & \text{otherwise,} \end{cases}$$

where $d(i)$ is the degree of node i in the underlying graph.

Process i then compares the value of its coin to the coins of its neighbors, and enters the MIS if its coin is 1, and for all its neighbors, j, such that either $d(j) > d(i)$ or $d(j) = d(i)$ and $j > i$, j's coin is 0. When a process gets in the MIS, all its neighbors also get removed from the protocol. Luby shows that in $O(\log n)$ expected rounds, this constructs an MIS.

3.1.1 Luby's Analysis

Luby's analysis focuses on the number of edges that get removed in each round of the protocol. Let E' be the set of edges in the graph, I be the set of processes in the MIS, $N(I')$ be the set of neighbors of processes in the MIS and Y_k be the number of edges in E' before the kth round of the protocol. The number of edges removed from E' due to the kth round of the protocol is $Y_k - Y_{k+1}$. The theorem that Luby proves is:

$$\exp[Y_k - Y_{k+1}] \geq \frac{1}{8} \cdot Y_k$$

Proof of theorem Let $G' = (V', E')$ be the the graph before the kth round of the protocol. The edges removed due to the kth round of the protocol are edges with at least one endpoint in the set $I' \cup N(I')$. Thus,

$$\begin{aligned}\exp[Y_k - Y_{k+1}] &\geq \frac{1}{2} \cdot \sum_{i \in V'} d(i) \cdot \Pr[i \in I' \cup N(I')] \\ &\geq \frac{1}{2} \cdot \sum_{i \in V'} d(i) \cdot \Pr[i \in N(I')]\end{aligned}$$

For all $i \in V'$ such that $d(i) \geq 1$, let

$$\mathrm{sum}(i) = \sum_{j \in \mathrm{adj}(i)} \frac{1}{d(j)}$$

The remainder of the proof is based on a lemma that states the $\Pr[i \in N(I')] \geq 1/4 \cdot \min\{\mathrm{sum}(i)/2, 1\}$. Thus,

$$\begin{aligned}\exp[Y_k - Y_{k+1}] &\geq \frac{1}{8} \cdot \left(\frac{1}{2} \cdot \sum_{\substack{i \in V' \\ \mathrm{sum}(i) \leq 2}} d(i) \cdot \mathrm{sum}(i) + \sum_{\substack{i \in V' \\ \mathrm{sum}(i) > 2}} d(i) \right) \\ &\geq \frac{1}{8} \cdot \left(\sum_{\substack{i \in V' \\ \mathrm{sum}(i) \leq 2}} \sum_{j \in \mathrm{adj}(i)} \frac{d(i)}{2 \cdot d(j)} + \sum_{\substack{i \in V' \\ \mathrm{sum}(i) > 2}} \sum_{j \in \mathrm{adj}(i)} 1 \right) \\ &\geq \frac{1}{8} \cdot \left(\sum_{\substack{(i,j) \in E' \\ \mathrm{sum}(i) \leq 2 \\ \mathrm{sum}(j) \leq 2}} \frac{1}{2} \cdot \left(\frac{d(i)}{d(j)} + \frac{d(j)}{d(i)} \right) + \sum_{\substack{(i,j) \in E' \\ \mathrm{sum}(i) \leq 2 \\ \mathrm{sum}(j) > 2}} \left(\frac{d(i)}{2 \cdot d(j)} + 1 \right) + \sum_{\substack{(i,j) \in E' \\ \mathrm{sum}(i) > 2 \\ \mathrm{sum}(j) > 2}} 2 \right) \\ &\geq \frac{1}{8} \cdot |E'| = \frac{1}{8} \cdot Y_k\end{aligned}$$

Lemma $\Pr[i \in N(I')] \geq 1/4 \cdot \min\{\mathrm{sum}(i)/2, 1\}$
Proof. $\forall j \in V'$, let E_j be the event that $\mathrm{coin}(j) = 1$ and let

$$p_j = \Pr[E_j] = \frac{1}{2 \cdot d(j)}.$$

Without lost of generality let $\mathrm{adj}(i) = \{1, \cdots, d(i)\}$ and let $p_1 \geq \cdots \geq p_{d(i)}$.
Let E'_1 be the event E_1 and for $2 \leq j \leq d(i)$ let

$$E'_j = (\bigcap_{k=1}^{j-1} \neg E_k) \cap E_j.$$

Let

$$A_j = \bigcap_{\substack{v \in \mathrm{adj}(j) \\ d(v) \geq d(j)}} \neg E_v.$$

Then,

$$\Pr[i \in N(I')] \geq \sum_{j=1}^{d(i)} \Pr[E'_j] \cdot \Pr[A_j | E'_j].$$

But

$$\Pr[A_j | E'_j] \geq \Pr[A_j] \geq 1 - \sum_{\substack{v \in \mathrm{adj}(j) \\ d(v) \geq d(j)}} p_v \geq \frac{1}{2}$$

and

$$\sum_{j=1}^{d(i)} \Pr[E'_j] = \Pr\left[\bigcup_{j=1}^{d(i)} E_j\right].$$

For $k \neq j, \Pr[E_j \cap E_k] = p_j \cdot p_k$. Thus, by the principle of inclusion-exclusion, for $1 \leq l \leq d(i)$,

$$\Pr\left[\bigcup_{j=1}^{d(i)} E_j\right] \geq \Pr\left[\bigcup_{j=1}^{l} E_j\right] \geq \sum_{j=1}^{l} p_j - \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j \cdot p_k.$$

Let $\alpha = \sum_{j=1}^{d(i)} p_j$. The technical lemma that follows implies that

$$\Pr\left[\bigcup_{j=1}^{d(i)} E_j\right] \geq \frac{1}{2} \cdot \min\{\alpha, 1\} \geq \frac{1}{2} \cdot \min\{\mathrm{sum}(i)/2, 1\}.$$

It follows that $\Pr[i \in N(I')] \geq 1/4 \cdot \min\{\mathrm{sum}(i)/2, 1\}$.□

Technical Lemma. *Let $p_1 \geq \cdots \geq p_n \geq 0$ be real-valued variables. For $1 \geq l \geq n$, let*

$$\begin{aligned} \alpha_l &= \sum_{j=1}^{l} p_j, \\ \beta_l &= \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j \cdot p_k, \\ \gamma_l &= \alpha_l - c \cdot \beta_l, \end{aligned}$$

where $c > 0$ is a constant. Then

$$\max\{\gamma_l | 1 \leq l \leq n\} \geq \frac{1}{2} \cdot \min\{\alpha_n, 1/c\}.$$

Proof omitted. See [16] for proof.

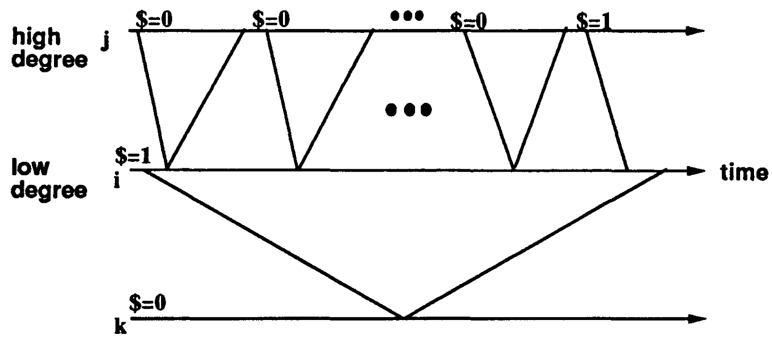

Figure 3-1: **The difficulty of asynchrony.** Why Luby's protocol does not work in the asynchronous network. Process j with degree bigger than that process i communicates quickly with i, while the link between i and k is slow. We cannot have j just freeze, whenever i is waiting to to hear from k, without causing deadlocks. However, while i waits to hear from k that it is safe to enter the MIS, j flips again many times and finally flips a 1, killing i's chances of getting in the MIS. An adversary can set link delays to cause such bad performance.

3.2 The difficulty of asynchrony

In an asynchronous environment, it is not clear how to implement a protocol like Luby's. Without a global clock, there is no way to insure that processes flip at the same rate. If we do not control the rate a process flips as compared to its neighbors, many things can go wrong. For instance, a fast-flipping process might have multiple chances to flip a 1 and kill slower-flipping neighbors (see Figure 3-1). Luby's protocol worked because in each round every process only had one chance to kill a neighbor that flipped 1. With asynchrony this is no longer guaranteed.

3.2.1 Synchronizer slowdown

To guarantee that neighbors only get one chance to kill a process that flipped a 1 we can add the α synchronizer of [5] to adopt Luby's protocol to the asynchronous environment. However, this may add an overhead of $O(D)$ for the first step in the protocol after the network has stabilized, where D is the diameter of the network after stabilization. The α synchronizer adopts a synchronous protocol for an asynchronous environment by generating pulses to simulate the rounds of the synchronous protocol (the pulse numbers corresponds to rounds). In the simulation, a process can only take a step if all its neighbors have the same pulse number. To see how the α synchronizer could add $O(D)$ for the first step in the protocol consider the following example of a dynamic MIS problem (see Figure 3-2):

1. 3 processes, p_0, p_1, p_2, get activated initially all with pulse 0.
2. Process p_1 is very fast so it sees all its neighbors are 0 and increments its pulse to 1. However, the link delay function has set the link (p_0, p_2) to be very slow, so p_0

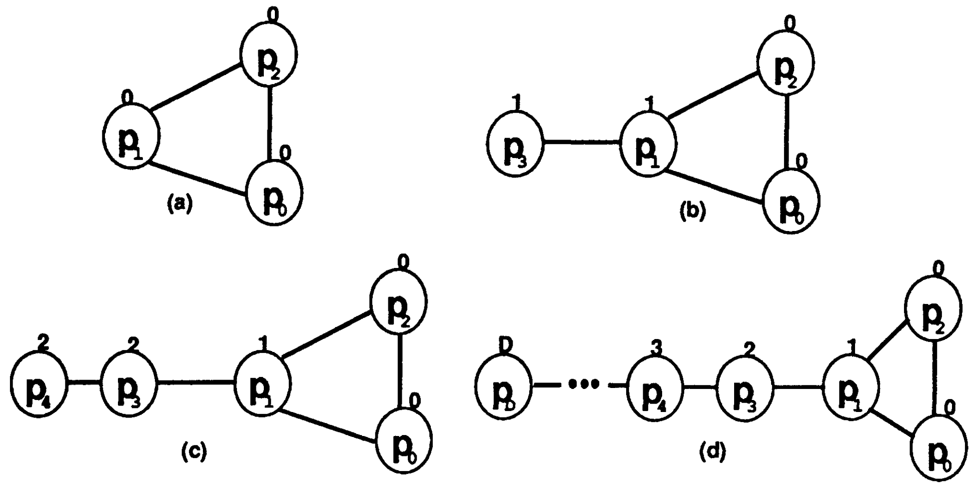

Figure 3-2: **Synchronizer slowdown.** (a) shows the 3 initial processes with pulse 0. In (b) p_1 has incremented its pulse to 1, and the newly activated process p_3 takes that pulse number. In (c) process p_3 has incremented its pulse to 2, so the newly activated p_4 takes that pulse. In (d) we have the worst case scenario where p_d has pulse number D (D is the diameter of the network). Thus, we now have a situation where p_d is dependent on p_0, so we have a waiting chain of length D.

will take a very long time to update its pulse.

3. Now a new process p_3 gets activated and it is only connected to p_1. When p_3 comes in it sees the pulse of p_1 is 1, so it sets its pulse to 1. Its neighbor is 1, so p_3 can update its pulse to 2.

4. Next p_4 comes in and is only connected to p_3. It sees p_3's pulse is 2 so it sets its pulse to 2. Its neighbors pulse is also 2, so p_4 can update its pulse to 3.

5. New processes get activated in the same manner until we have a chain whose length is the diameter of the network.

In such a scenario, the process at the end of the chain may have to wait for the pulse to propagate from p_0 before the first step it takes in the execution of the protocol. Additionally if p_0 fails, then all the processes in the chain, even a process as far as $O(D)$ away, will stop making progress.

3.2.2 Freezing fast processes

The crux of the problem is as follows: suppose a process i flips a 0. Then its neighbors who have flipped 1's should have a chance to enter the MIS before i flips again, but we do not want to pay the overhead of a synchronizer. What we do instead is when a process flips a 1 its neighbors *freeze,* while it checks to see if it will survive and enter the MIS. Except, if we allow each of i's neighbors to freeze i in turn, i can stay frozen a long time with no chance to enter the MIS.

The solution to this in our protocol is when a process i flips a 0, it freezes for each neighbor's current coin flip before it flips again. When a neighbor flips again, this unfreezes i, *and a neighbor who unfreezes i's new coin cannot freeze i again until i has flipped a new coin too.* What this gives us is that a fast neighbor may get at most 2 chances to kill a slower neighbor. That is, in our protocol if a process i flips a 1, then each neighbor has at most 2 flips that has a chance to kill i's flip of 1. This is sufficient to maintain the $O(\log n)$ expected time bound (see section 3.4.2 for proof). The freezing mechanism also gives us wait-dependency of only 2 (proof in section 3.5).

3.3 The asynchronous MIS protocol

We start by describing the internal variables used by the process.

d$_i$: represents the pair $(d(i), i)$ in process i.

Coin: the current value of i's flipped coin.

Coin(j): records information about neighbor j's coin. It has three possible values, UNSET if process i's coin was just flipped and neighbor j's coin is unknown; 0 if ($d_j < d_i$ and coin $= 1$) or (j's coin $= 0$); and 1 if ($d_j > d_i$ and j's coin $= 1$) or (j's coin $= 1$ and coin $= 0$).

Freeze(j): when coin $= 0$, this variable keeps track of whether i froze its current coin for neighbor j, has already frozen and then unfrozen its current coin for neighbor j, or has not yet frozen for j. These notions correspond to the values 1, 0, and UNMARKED respectively.

Neighbors: set of adjacent vertices not known to be in the MIS nor to have a neighbor in the MIS.

MIS-flag: flag that indicates i's status in the MIS.

Update-flag(j): flag that indicates that j is no longer in the protocol and should be removed from the neighbor set of i.

All flags have initial value UNSET.

Below is a description of messages received by the process.

WAKEUP$_i$(N, v): message from the environment to become active in the protocol.

(New,j): message from neighbor j saying than it is newly activated.

(Query, F, d_j): message from neighbor j indicating j's coin $= F$, j's ID and current degree, and also that this is a query message, requesting the value of i's coin.

(ACK,F, d_j): message from neighbor j indicating j's coin $= F$, j's ID and current degree, and also that this is an ack message in response to a query.

(InMIS,j): message from j saying it is in the MIS.

(Remove,j): message from neighbor j saying that it has a neighbor in the MIS and should be removed from the set of active nodes.

Here we give an informal description of the how the protocol works. The crux of the protocol lies in how a process responds to Query and ACK messages. Depending on whether it flipped a 0 or a 1, i will respond differently to ACK and Query messages from its neighbors. We first describe how it responds if the current flip of its coin is a 0. When an ACK message is received from a neighbor j, Coin(j) is set to the value of the coin j sent in the ACK message. If Coin(j) is 1 i does not want to flip again until j has had a chance to try to get in to the MIS, so Freeze(j) is set to 1. If the Coin(j) is 0, Freeze(j) is set to 0. If none of the neighbors had a coin that was 1 at the time it received i's Query, then for all j Freeze(j) will be 0, so i can flip again. Meanwhile, when i receives a Query message from a neighbor j, the sending of an ACK message is enabled. If Freeze(j) is 1, it means j had frozen i on its previous flip, but did not get in the MIS and so has flipped again. Since i will not allow j to freeze it a second time before i gets a chance to flip again, it sets Freeze(j) to 0. If now for all neighbors k Freeze(k) equal 0, i flips again.

Now we describe how i responds if its coin is 1. On an ACK from neighbor j, it will set Coin(j) to 0 if j's coin was 0, or if $d_j < d_i$. Otherwise Coin(j) is set to 1. If for all neighbors k Coin(k) is 0, i enters the MIS by setting its MIS-flag to 1 and sending a message to all i's neighbors saying that it is in the MIS. If i did not beat out all its neighbors to get in the MIS, and has received ACK's from all of them, it flips again. Upon receiving a Query from j, the sending of an ACK message is enabled in i. Freeze(j) is unaffected because Coin is equal to 1. However, we could have a scenario where j was frozen by i and then unfrozen by i on i's latest flip. When i sent a Query to j, j could have still been held by some other process and so sent a Coin = 0 in its ACK to i. However, before i could receive ACK's from all its other neighbors, j could have since gotten unfrozen by the neighbors who were holding it and flipped again. Thus, the Query message could have a new value of Coin from j. Therefore, i might have to update its value of Coin(j) based on this message. This update is only significant if the new coin is 1, because it may affect the safety condition.

3.3.1 The code

Our protocol works in the model described in the previous chapter, where we also gave the problem specification. Let i be the process with ID = i, and let $d(i)$ be its degree in the network. For notational convenience we will define d_i to be the ordered pair $(d(i), i)$ and say that $d_i > d_j$ if ($d(i) > d(j)$ or $d(i) = d(j)$ and $i > j$). Also we write WAKEUP$_i(N)$ as shorthand for WAKEUP$_i(N, _)$. The code for a process i is shown in below in figure 3-3.

```
Program RECEIVE(C):                                              /* program on process i */

C = WAKEUP_i(N)
  Effect:        Neighbors ← N
                 ∀j ∈ Neighbors put (New,ID) in send-buffer(j)
                 ∀j Freeze(j) ← 0

C = (New,j)
  Effect:        Neighbors ← Neighbors + {j}
                 if MIS-flag = 1, put (In-MIS,ID) in send-buffer(j)
                 if MIS-flag = 0, put (Remove,ID) in send-buffer(j)
                 if MIS-flag = UNSET, put (Query,Coin,d) in send-buffer(j)

C = Query(F,d_j)
  Effect:        put (ACK,Coin,d) in send-buffer(j)
                 if Freeze(j) = 1,
                   then Freeze(j) ← 0
                 if d_j > d and F = 1 and Coin = 1 and Coin(j) = 0,
                   then Coin(j) ← 1

C = ACK(F,d_j)
  Effect:        if Coin = 1
                   if d_j < d or F = 0,
                     then Coin(j) ← 0
                     else Coin(j) ← 1
                   if ∀k Coin(k) = 0,
                     then Enter-MIS
                     else if ∀k Coin(k) ≠ UNSET,
                           then ∀k Freeze(k) ← 0
                 if Coin = 0,
                   then Coin(j) ← F
                   if Coin(j) = 1,
                     then Freeze(j) ← 1
                     else Freeze(j) ← 0

C = Remove(j)
  Effect:        update-flag(j) ← 1

C = InMIS(j)
  Effect:        MIS-flag ← 0
                 ∀k ∈ Neighbors, put (Remove,ID) in send-buffer(k)

Flip-Coin
  Precondition:  ∀k Freeze(k) = 0
  Effect:        ∀j s.t. update-flag(j) = 1, Neighbors ← Neighbors - {j}
                 Coin = 1 with probability 1/4|Neighbors|
                      = 0 otherwise
                 ∀j ∈ Neighbors
                      Coin(j) ← UNSET
                      Freeze(j) ← UNMARKED
                      put (Query,Coin,d) in send-buffer(j)

SEND_j(m)
  Precondition:  m ∈ send-buffer(j)
  Effect:        send-buffer(j) ← send-buffer(j) − m

procedure Enter-MIS
  MIS-flag ← 1
  ∀j ∈ Neighbors put (InMIS,ID) in send-buffer(j)
```

Figure 3-3: MIS Algorithm

3.4 Analysis of the algorithm

In this section we prove the safety and liveness properties of our algorithm, and we also give the analysis of the expected running time.

3.4.1 Safety

Lemma 3.1 *If i has MIS-flag = 1, then for all neighbors j of i, j has MIS-flag = 0.*

Proof. In the protocol, a process i will join the MIS only if $\forall j$ such that $d_j > d_i$, Coin(j) = 0 and it has set Coin = 1. Assume, by contradiction, that two neighbors i and j, both enter the MIS. Let $winner_i$ be the last coin that i flipped before joining the MIS, and let $winner_j$ be the last coin that j flipped before joining the MIS. Then there must be some time t_i at which i flips the coin $winner_i$ and similarly define time t_j. Notice that by definition of the coins $winner_i = winner_j = 1$. After a process flips its winning coin, that coin will stay at 1 for all time. There are several cases.

1. t_j is before t_i, and at time t_j, $d_j > d_i$. Then at time t_i, when i queries all its neighbors, the ACK from j will say Coin $j = 1$. Since j doesn't update its degree until it flips again, and j hasn't flipped since time t_j by assumption, j's ID remains what it was at time t_j, or possibly j has entered the MIS by time t_i. In either case $winner_j$ will not allow i to enter the MIS.

2. t_i is before t_j, and at time t_i, $d_i > d_j$. Same as Case 1, by symmetry.

3. t_i is before t_j, and at time t_i, $d_i < d_j$. There are several subcases.

   (a) t_j occurs while Coin(j) at i is still UNSET. Then this is the same as Case 1, above.

   (b) t_j occurs after i has received an ACK from neighbor and j set its flag Coin(j), but before i has heard from all neighbors. Then i has set its flag according to the previous coin of j (which didn't win). j cannot enter the MIS until j has queried all its neighbors (including i) and received ACKs. If i has not yet heard from all its neighbors when it receives j's new query, it resets its flag Coin(j) = $winner_j$, and $winner_j$ will not allow i to enter the MIS.

   (c) t_j occurs after i has already received ACK's from all its neighbors. By assumption, all of i's neighbors of higher ID reported a flip of 0. Therefore, i has already sent an InMIS message to j which will reach j before it receives an ACK from i by the FIFO property of the links, and so j will never enter the MIS.

4. t_i is before t_j, and at time t_i, $d_j > d_i$. Same as case 3, by symmetry. □

3.4.2 Analysis of expected running time

For a process i that flips a 1, let coin_j be the coin process j reports to i after i's flip of 1, and let next_j be j's next coin.

Lemma 3.2 *If i flips a 1 and for all neighbors j of bigger degree $coin_j = 0$ and $next_j = 0$, then i enters the MIS.*

Proof. In our protocol, a process k gets in the MIS if the following 3 conditions are satisfied:

1. it flips a 1,
2. at the moment the query arrives at the neighbor, all its neighbors of higher degree each has coin $= 0$, and
3. no neighbor of higher degree flips again and flip a 1 after it sent an ack to k and before k gets a chance to enter the MIS.

Condition 3 is needed because a process j might have been frozen when it responded to the query from its neighbor and subsequently gets unfrozen. If that happens and the next flip of j produces a 0, then when j queries k it will freeze based on k's coin being 1. All three conditions are clearly satisfied by i by the statement of the lemma. □

Lemma 3.3 *A process will flip again or enter the MIS within time 2ν after flipping a 1, and will flip again within time 5ν after flipping a 0, where ν is the maximum link delay.*

Proof. If coin $= 1$, then a process only needs to receive ack's from all its neighbors before it enters the MIS or flip again. Thus, the delay is at most 2ν.

If coin $= 0$ for some process i, then clearly the worst case occurs when Freeze flags get marked 1 since a process will have to wait until all these flags get marked 0 before it can flip again. Freeze flags get marked 1 only after i receives ack's in response to queries. Process i sends the queries and receives the acks within time 2ν. If Freeze(j) got marked 1 it means j must have flipped a 1. We are interested in the case where j loses and does not enter the MIS since if it did enter the MIS, both it and j would get removed from the protocol. Freeze(j) will get marked 0 when i receives a new query message from j. Since j flipped a 1 and we assume it does not enter the MIS, it will flip again within time 2ν and thus send a query message to i which will arrive at i within time 3ν. When i gets a query message it will flip again for a total delay between flips of at most 5ν. □.

Lemma 3.4 *Let α_0 and α_1 be two paths in the execution tree generated by our MIS protocol. Suppose that α_0 and α_1 are the same up to time t_0, but in α_0 j flips 0 and in α_1 j flips 1 at t_0. Assume j does not get in the MIS on its flip at t_0. Then the amount of time it takes j to flip again in α_0 is greater than or equal to the time it takes j to flip again in α_1 regardless of the coin flips of other processes.*

Proof. By the design of the protocol, at t_0 for both α_0 and α_1 j sends a query to all its neighbors and gets acks in response. The time it takes for the j to send the queries and receive the acks from its neighbors is identical for α_0 and α_1. The times are identical because the link delay function sets the delays based on time the message is being sent and the link on which it is been sent on. This is identical for both α_0 and α_1. When it gets the last ack from its slowest neighbor, j will immediately flip again in α_1. However, in α_0 when j gets the last ack, it might immediately flip again if none of its neighbors had a coin of 1. In this case the time to flip again for α_0 would be the same as for α_1. However, if there was a neighbor that had flipped a 1, then j will get frozen, and will not flip again until that neighbor has flipped again or got in the MIS. In this case the time before j flips again will be greater in α_0 than in α_1. □

Lemma 3.2 proves that a process i gets in the MIS if it flips a 1 and coin_j and next_j is 0 for all its neighbors j of bigger degree. However, for the purpose of our analysis we take a step back and look at the situation where i flips 0 and then 1. Let t_0 be the time i flips 0.

In the execution tree generated by our protocol, we call the occurrence of the action where any process k flips a coin c_k. For process j, if this flip of j is the last one before c_i on a branch of the execution tree, we call this action c'_j. The coin of j at c'_j will be coin_j, and the coin of j after will be next_j. The execution tree we look at will start at time t_0. We can start our execution tree at this point because of the next lemma.

Lemma 3.5 *If c'_j occurred before time t_0 then $coin_j = 0$ with probability 1.*

Proof. We examine the case where the last c_j before t_0 produced a 1, and the case where it produced a 0.

1. If when i sent a query to j after it flipped at time t_0 it got an ack from j indicating that j's coin is 1, then i will freeze and cannot flip again until j flips again and unfreezes it. This flip of j happens after t_0. Thus, when c_i happens, the flip of j after t_0 or some later flip will be c'_j.

2. If when i sent a query to j after it flipped at time t_0 it got an ack from j indicating that j's coin is 0, then i may not feeze and could flip again before j gets a chance to flip. In this case c'_j could be before t_0. □

Since we will only need to show that the $\Pr[\text{coin}_j = 0] \geq (1 - 1/4d(i))$, this case only helps us.

Conjecture 3.1 *If a process i flips 0 and 1 on consecutive flips, then $\Pr[coin_j = 0 \cap next_j = 0] \geq (1 - 1/4d(i))^2$ for any neighbor j.*

The intuition that forms the basis for this conjecture is based on two properties of the algorithm.

1. Lemma 3.5 proves that if c'_j happened before t_0, then coin_j will be 0 with probability 1, so we can ignore this case.

2. If we are not in case 1, so there is at least one flip of j after t_0, then lemma 3.4 says that j will sit on a 0 at least as long as it sits on a 1. The intuition behind why this lemma helps us is that if j could sit on a 1 for a long time then it could sit on that 1 until c_i occurs thereby increasing the probability that $\text{coin}_j = 1$. However, since j cannot sit on a 1 longer than it does on a 0, the probability that $\text{coin}_j = 1$ is not increased by how long j sits on a 1.

However, analyzing all the various cases that can develop in the execution tree has proved to be extremely complex, so we have not been able to prove the conjecture as yet.

The lemmas and the theorem that follow in this section are proved subject to the condition that conjecture 3.1 is true.

For all $j \in V'$, let E_j be the event that $\text{coin}_j = 1$ and let

$$p_j = \Pr[E_j] = \frac{1}{4d(j)}.$$

Without lost of generality let, $\text{adj}(i) = \{1, \cdots, d(i)\}$ and let $p_1 \geq \cdots \geq p_{d(i)}$.
Let F_j be the event that j flipped 01 on consecutive coins. Let F_k^j be the event that $\text{coin}_k = 1$ when the query j sends after it flips 1 arrives at k. Let E'_1 be the event E_1 and for $2 \leq j \leq d(i)$ let

$$E'_j = (\bigcap_{k=1}^{j-1} \neg E_k) \cap E_j.$$

Let F'_1 be the event F_1 and for $2 \leq j \leq d(i)$ let

$$F'_j = (\bigcap_{k=1}^{j-1} \neg F_k^j) \cap F_j.$$

Lemma 3.6 $\Pr[F'_j] \geq 3/4 \Pr[E'_j]$.

Proof.

$$\Pr[F_j] = \left(1 - \frac{1}{4d(j)}\right) \cdot \frac{1}{4d(j)}.$$

Since $1 - 1/4d(i) \geq 3/4$, $\Pr[F_j] \geq 3/4 \Pr[E_j]$. Also Since

$$(\bigcap_{k=1}^{j-1} \neg F_k^j) \text{ and } F_j \text{ are independent events, we get}$$

$$\Pr[F'_j] = 3/4 \cdot \Pr[(\bigcap_{k=1}^{j-1} \neg F_k^j) \cap E_j.$$

If we assume conjecture 3.1, then we get $\neg F_k^j = (1 - 1/4d(i)) = \neg E_k$. Thus, we have $\Pr[F_j'] \geq 3/4 \Pr[E_j']$.□

Let I' be the set of processes in the MIS, $N(I')$ be the set of neighbors of processes in the MIS and

$$A_j = \bigcap_{\substack{v \in \mathrm{adj}(j) \\ d(v) \geq d(j)}} (\mathrm{coin}_v = 0 \cap \mathrm{next}_v = 0)$$

For all $i \in V'$ such that $d(i) \geq 1$, let

$$\mathrm{sum}(i) = \sum_{j \in \mathrm{adj}(i)} \frac{1}{d(j)},$$

and let $\alpha = \sum_{j=1}^{d(i)} p_j$.

Lemma 3.7 *In any time interval of* $7\nu, \Pr[i \in N(I')] \geq 3/16 \cdot \min\{sum(i)/4, 1\}$.

Proof. From lemma 3.2 we know

$$\Pr[i \in N(I')] \geq \sum_{j=1}^{d(i)} \Pr[F_j'] \cdot \Pr[A_j | F_j'].$$

But

$$\begin{aligned}
\Pr[A_j \mid F_j'] &\geq \Pr[A_j] \\
&\geq \bigcap_{\substack{v \in N(j) \\ d(v) \geq d(j)}} (1 - p_v)^2 \\
&\geq \bigcap_{\substack{v \in N(j) \\ d(v) \geq d(j)}} 1 - 2p_v \\
&\geq 1 - \sum_{\substack{v \in N(j) \\ d(v) \geq d(j)}} 2p_v \\
&\geq 1/2.
\end{aligned}$$

From Luby's proof given in section 3.1 we know

$$\begin{aligned}
\sum_{j=1}^{d(i)} \Pr[E_j'] &\geq \frac{1}{2} \cdot \min\{\alpha, 1\} \\
&= \frac{1}{2} \cdot \min\{\mathrm{sum}(i)/4, 1\}
\end{aligned}$$

From lemma 3.6 we get

$$\begin{aligned}\sum_{j=1}^{d(i)} \Pr[F_j'] &= \frac{3}{4}\sum_{j=1}^{d(i)} \Pr[E_j'] \\ &\geq \frac{3}{8}\cdot \min\{\mathrm{sum}(i)/4, 1\}\end{aligned}$$

Thus,

$$\begin{aligned}\sum_{j=1}^{d(i)} \Pr[F_j']\cdot \Pr[A_j|F_j'] &\geq \frac{1}{2}\cdot\frac{3}{8}\cdot \min\{\mathrm{sum}(i)/4, 1\} \\ &= \frac{3}{16}\cdot \min\{\mathrm{sum}(i)/4, 1\}\end{aligned}$$

Lemma 3.3 shows that processes always flip again within time 5ν. If that flip is a 1, then it takes 2ν time units to check with its neighbors to see if it gets in the MIS. Thus, a process can flip and check whether is in the MIS in time 7ν.□

Let E' be the set of edges in the graph, Y_t be the number of edges in E' at time t and let $Y_{t+7\nu}$ be the number of edges in E' at time $t+7\nu$. The number of edges removed from E' in that time interval due to the protocol is $Y_t - Y_{t+7\nu}$.

Subject to the proof of conjecture 3.1 we can show the following theorem.

Theorem 3.1 $\exp[Y_t - Y_{t+7\nu}] \geq \frac{3}{64}\cdot Y_t$. *Thus, we will get an MIS on the graph in expected* $O(\log n)$ *time.*

Proof. We follow Luby's proof, except lemma 3.7 gives a different bound for $\Pr[i \in N(I')]$. Thus,

$$\begin{aligned}\exp[Y_t - Y_{t+7\nu}] &\geq \frac{1}{2}\cdot \sum_{i\in V'} d(i)\cdot \Pr[i \in I' \cup N(I')] \\ &\geq \frac{1}{2}\cdot \sum_{i\in V'} d(i)\cdot \Pr[i \in N(I')]\end{aligned}$$

From lemma 3.7 we know $\Pr[i \in N(I')] \geq 3/16 \cdot \min\{sum(i)/4, 1\}$. Thus,

$$\begin{aligned}\exp[Y_t - Y_{t+7\nu}] &\geq \frac{3}{32}\cdot\left(\frac{1}{4}\cdot \sum_{\substack{i\in V' \\ \mathrm{sum}(i)\leq 4}} d(i)\cdot \mathrm{sum}(i) + \sum_{\substack{i\in V' \\ \mathrm{sum}(i)>4}} d(i)\right) \\ &\geq \frac{3}{32}\cdot\left(\sum_{\substack{i\in V' \\ \mathrm{sum}(i)\leq 4}} \sum_{j\in \mathrm{adj}(i)} \frac{d(i)}{4\cdot d(j)} + \sum_{\substack{i\in V' \\ \mathrm{sum}(i)>4}} \sum_{j\in \mathrm{adj}(i)} 1\right)\end{aligned}$$

$$\geq \frac{3}{32} \cdot \left(\sum_{\substack{(i,j)\in E' \\ \text{sum}(i)\leq 4 \\ \text{sum}(j)\leq 4}} \frac{1}{4} \cdot \left(\frac{d(i)}{d(j)} + \frac{d(j)}{d(i)} \right) + \sum_{\substack{(i,j)\in E' \\ \text{sum}(i)\leq 4 \\ \text{sum}(j)>4}} \left(\frac{d(i)}{4 \cdot d(j)} + 1 \right) + \sum_{\substack{(i,j)\in E' \\ \text{sum}(i)>4 \\ \text{sum}(j)>4}} 2 \right)$$

$$\geq \frac{3}{64} \cdot |E'| = \frac{3}{64} \cdot Y_t \square$$

3.5 Proof of 2-wait-freedom

In Chapter 2 section 2.4 we said a process is k-wait-free if for any process i, if all the processes in the distance k neighborhood of i stop receiving new neighbors and continue to take steps, then i will accomplish its task.

Theorem 3.2 *The dynamic MIS protocol is 2-wait-free.*

Proof. If i flips a 1, then it only needs to get acks from its immediate neighbors in order to proceed with the protocol. By the definition of 2-wait-free, all of i's neighbors will continue to take steps, so they will respond accordingly. Even if the neighbors of i are waiting for acks from some of their neighbors, by the design of the protocol, as long as they are still taking steps they will still respond to queries with an ack.

If i flips a 0, then it's dependency may extend out to the distance 2 neighborhood. This happens because if i has a neighbor j that flipped a 1, then i will get frozen by j. i can only flip again after j flips again or gets in the MIS. For j to flip again it has to get acks from all its neighbors. Thus, i will need all of j's neighbors to be taking steps for it to proceed. We showed above that when j flips a 1 it is only be dependent on its immediate neighbors, so i's dependency would not extend beyond these neighbors of j. By the design of the protocol frozen processes still send acks in response to queries. They are only frozen in the sense that they will not flip until unfrozen, and since processes can only get frozen on flips of 0, and processes that flip 0 cannot freeze other process, there is no chain of frozen processes. □

Chapter 4

Conclusion

In the thesis we provided a randomized solution, that we believe runs in $O(\log n)$ expected time, for the asynchronous dynamic MIS problem. Significantly, the algorithm is also 2-wait-free. The previous best known solutions using the α synchronizer of [5] plus the algorithm of [15] or [16] and constructs an MIS in $O(D + \log n)$ expected time, where D is the diameter of the network after it stabilizes. These algorithms are also D-wait-free.

We think that the fact that our algorithm is 2-wait-freedom is an important property because k-wait-freedom as we have defined it in this thesis captures important properties of distributed algorithms. Our definition matches the idea that in an asynchronous system, a process should not be slowed or stopped by slow or faulty processes or links that are far away. If a distributed protocol is k-wait-free then the operations of a process cannot be affected by processes greater than distance k away. Thus, in our protocol a process will tolerate any failure outside of its distance 2 neighborhood.

4.1 Related work

Symmetry breaking has many applications in distributed computing. It is essential for solutions to deadlock resolution, leader election, mutual exclusion, and resource allocation. In [6], the main idea in our dynamic MIS protocol is employed to solve the general resource allocation problem of which the dining philosophers problem is one formulation. They way this is done is that the MIS protocol is run at every process, when a process gets in the MIS this means it has access to all its resource and can execute it job. When that process finishes executing its job, its neighbors continue to run the MIS protocol until they get in the MIS. The solution to the dining philosophers problem we believe runs in expected optimal time $O(\delta)$, where δ is the maximum number of competing processes any process has. However, we get this expected running time subject to the proof of the conjecture we make in chapter 3. The previous best known randomized result of Awerbuch and Saks, [10], had a expected running time of $O(\delta^2)$, and the best know deterministic protocol due to Choy and Singh [11] has a running time of $O(\delta^2)$. The wait-dependency of 2 also holds for the dining philosophers algorithm. The algorithm with the previous best known wait-dependency was that of [11] which has a wait-dependency of 4.

4.2 Open problems

The most obvious open problem that this thesis poses is proving the conjecture we make in the previous chapter. Proving probabilistic statements about asynchronous protocols are notoriously difficult, and a proof for this conjecture might give insights into proving claims about other probabilistic asynchronous protocols.

The property of k-wait-freedom needs to be studied further in message passing asynchronous systems. Finding k-wait-free solutions, where k is a small constant, for other distributed problems is an area for exploration.

Our protocol for the dynamic MIS problem is a randomized protocol. Finding and a deterministic protocol that achieves comparable response time is an important problem. While Luby can remove randomness from his MIS algorithm to make it deterministic in the PRAM model [16], computing an MIS in the distributed model of computation in poly-logarithmic time remains an open question. (The best known deterministic running time for MIS is $O(n^{O(1/\sqrt{\log n})})$ [22].) Finding a deterministic poly-logarithmic solution to the MIS problem would also give a deterministic optimal solution to the resource allocation problem.

Bibliography

[1] Y. Afek, G.M. Landau, B. Schieber, and M. Yung. The power of multimedia: combining point-to-point and multiaccess networks. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 90–104, Toronto, Canada, August 1988.

[2] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370, October 1987.

[3] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 131–148. ACM SIGACT and SIGOPS, ACM, 1988.

[4] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 1991.

[5] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[6] Baruch Awerbuch, Lenore Cowen, Mark Smith. The philosophers eat at Wendy's: A fast wait-free self-stabilizing symmetry breaker. Unpublished manuscript, November 1992.

[7] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. End-to-end communication with polynomial overhead. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989.

[8] Baruch Awerbuch and Silvio Micali. Dynamic deadlock resolution protocols. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, October 1986.

[9] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 268–277, October 1991.

[10] Baruch Awerbuch and Mike Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

[11] M. Choy and A.K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[12] Edsger W. Dijkstra. Solution of a problem in concurrent programming control *Comm. of the ACM*, 8(9):569, September 1965.

[13] E.W. Dijkstra. Hierarchical ordering of sequential processes. *ACTA Informatica*, pages 115–138, 1971.

[14] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[15] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. of the ACM*, 32(4):762–773, October 1985.

[16] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Comput.*, 15(4):1036–1053, November 1986.

[17] N. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal Of Computation And Systems Sciences*, 23(2):254–278, October 1981.

[18] N. Lynch and H. Attiya. Using Mappings to Prove Timing Properties. Technical Report MIT/LCS/TM-412.e, Laboratory for Computer Science, MIT April 1992.

[19] N. Lynch and M. Tuttle. Hierarchiacal Correctness Proofs for Distributed Algorithms. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1987, pp 137-151.

[20] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. CWI-Quarterly, 2(3), 1989.

[21] N. Lynch and B. Patt-Shamir. Distributed Algorithms: Lecture Notes for 6.852J Unpublished manuscript, Fall 1992.

[22] Alessandro Pasconesi and Aravind Srinivasan. Improved algorithms for network decompositions. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.