Efficient Asynchronous Distributed Symmetry Breaking

Baruch Awerbuch * Lenore Cowen † Mark Smith ‡

Abstract

This paper considers symmetry-breaking in an asynchronous distributed network. We present and analyze a randomized protocol that constructs a maximal independent set in $O(\log n)$ expected time, and also a protocol for the dining philosophers problem that schedules a job that competes with δ other jobs in expected $O(\delta)$ time, which is optimal. The best previous algorithms for dining philosophers achieved only $O(\delta^2)$. In addition, the new protocols are *2-wait-free* which means that delays at a process are only dependent on processors or links at most distance two in the communication graph.

1 Introduction

We consider an *asynchronous* distributed network of processors with arbitrary network topology. This can be represented as a graph, where vertices represent processors, and two vertices are connected by an edge if the corresponding processors have a direct communication link. There is an arbitrary link delay function on each edge: a message sent on link ij at time t, arrives at some time $f_{ij}(t)$. No information about this function (such as upper bounds on the maximum link delay, for example) is assumed to be known to either the algorithm designer, or the protocol. Since we are interested in symmetry breaking, we do not assume processors are given unique names, or IDs: if the underlying network topology is a distance regular graph, they are indistinguishable.

Typically, algorithm designers have designed distributed algorithms for such an asynchronous distributed network in two phases:

1. Design a round-based protocol that performs well in a synchronous distributed network.
2. Implement a general *synchronizer* which allows the asynchronous network to simulate a synchronous one.

In this paper, we will show that for a fundamental class of problems that includes the maximal independent set problem and the dining philosophers problem, sometimes called *symmetry breaking* problems (because their solution relies either on unique processor IDs, which we are not assuming, or randomness, which we will use) it is possible instead to design the algorithms directly for the asynchronous network. The new algorithms are nearly as simple as the synchronous algorithms, and have similar complexity in an asynchronous network (times the usual factor for maximum link delay) as the old algorithms did in the synchronous network. In addition, they will avoid all the complexity, fault-sensitivity, and overhead involved in setting up and maintaining even the best known synchronizers, especially when the network is allowed to change dynamically. The only payment we have to make for these new algorithms turn out to be in the analysis: there is a hint of the adversarial in requiring good performance for any possible link delay function. The delicate protocols that we present insure both safety and fairness without ever resolving some inherent ambiguity as to the order of events driven by when messages are received. The challenge is the probabilistic analysis of the protocols, where the coin tosses of the algorithm can interact with the link delay function and change the orders of events.

1.1 Our results

Our results include an $O(\log n)$ expected time solution to the maximal independent set problem, and the first optimal algorithm for the dining philosophers problem, where a job which competes with δ others for resources, is scheduled in $O(\delta)$ expected time. Both protocols are 2-wait-free which makes them robust against faults. They also use constant space per edge, sending messages of size at most $O(\log \log n)$. We sketch how the protocols are extended to work in the infinite dynamic model considered for dining philosophers by Awerbuch and Saks in 1990 [3]. (In this case, the protocols will be 3-wait-free, and the complexity bounds

*Johns Hopkins University and Lab. for Computer Science, MIT. Supported by Air Force Contract AFOSR F49620-92-J-0125, NSF contract 9114440-CCR, DARPA contracts N00014-91-J-1698 and N00014-J-92-1799, and a special grant from IBM.

†Rutgers and Johns Hopkins University. Supported by an NSF postdoctoral fellowship.

‡Lab. for Computer Science, M.I.T. Supported in part by an AT&T Bell Labs Cooperative Research Fellowship.

are still $O(\delta)$, where $\delta =$ the maximum number of jobs a job competes with at any point in time).

1.2 Previous work

For the maximal independent set problem, there are the famous randomized synchronous algorithms of Karp-Widgerson [9] and Luby [10]. [1] There is an extensive literature on the dining philosophers problem and resource allocation (e.g.[7, 12, 16, 5, 11, 3, 6]) and we survey these results at the beginning of Section 6.

1.3 Outline

The structure of this extended abstract is as follows. We present the model in the next section. We then focus on re-designing Luby's elegant synchronous protocol for the maximal independent set problem. We review Luby's protocol in Section 3, demonstrate its inherent dependence on synchronous rounds, and discuss drawbacks for implementations in an asynchronous that employ a synchronizer. Section 4 presents our new asynchronous algorithm, and proves safety. Section 5 then analyzes the performance of the algorithm. Section 6 gives the application to resource allocation and the first optimal solution to the generalized dining philosophers problem.

2 The model

We model each process as a *probabilistic timed* I/O automaton (see [15, 13, 14] for formal definitions). The algorithm begins when a 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. A process that has received a wakeup message as an input we will define as *active*. A newly activated process may be connected (have in its neighbor set) any previously active process. (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.) We allow processes to wakeup one by one, or all wakeup simultaneously, or anything in between.

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.

2.1 The link delay function

We use the standard method to model link delays in an asynchronous network, sometimes referred to as a weak "oblivious adversary" model. We assume that for each process i, there is a function C_i and for each directed communication link from processor i to j, there is a function L_{ij}. More specifically L_{ij} is defined as follows: If process i sends a c-bit message to process j along link ij at time t, then that message arrives at time $L^c_{ij}(t)$. Just as the delay functions L capture that that link delay is oblivious to the content of the c-bit messages sent, the process delay functions C captures the notion that the time that processor i takes to compute a particular function g (say, of k inputs) is not dependent on the values of those k inputs. All delay functions are assumed to take on positive, finite values, but otherwise can vary arbitrarily with time.

We assume that messages sent at time t across link ij arrive at time $L_{ij}(t)$ regardless of other messages on other links for process j. If each process scans incoming links in fixed order (which we can assume takes 0 time, and instead factor the delay into the link delay functions), without loss of generality we can assume that messages do not arrive simultaneously.

2.2 Complexity measures

We use the standard method of measuring complexity for asynchronous distributed algorithms. The algorithm is said to run in time $O(r)$ if, for any set of delay functions, with maximum process delay μ and maximum link delay ν, it runs in time $O(r(\mu + \nu))$. (Note that μ and ν are just for the purposes of analysis: the protocol does not know anything about the delay functions, including any upper bound on maximum delays). Alternately, when not reasoning specifically about exact delays, the normalized delays can be used, namely physical execution time divided by maximum link or process delay. This is equivalent to measuring physical time assuming message delay varies between 0 and 1 time units of a global clock [1, 8].

[1] There has also been work on MIS in the asynchronous PRAM model, but a shared memory makes coping with asynchrony a much easier problem. We note that our algorithm is also an asynchronous PRAM algorithm, and its properties are also of potential interest in this model.

2.3 The problem specification

We define both a stable and an infinite dynamic version of these problems; the infinite dynamic version in anticipation of the application to dining philosophers. In the stable version, it is assumed there is some constant delay w (not know in advance to the protocol), after the first process has woken up, after which all active processes have awakened (i.e. no processors will receive new neighbors after time w.) Then we measure the complexity of our programs after all active processors have awakened.

In the infinite dynamic version, a process may get new neighbors at anytime during an execution. It is more difficult to pin down a good global measure of complexity in this formulation, though we will see a local measure that makes sense for the generalized dining philosophers problem.

Notice that the difference in the two models is in how we measure *performance* not correctness: in both the infinite dynamic and stable cases, processes will wake up while the protocol is already in progress, and thus safety must in both cases be proved in a fully dynamic model. However, in the stable case, we do not necessarily require good performance while new processes are entering on line, while in the infinite dynamic case, we must in some sense continue to "make progress".

There is a condition on the types of problems that can be solved in these models. If a process has already produced an output, we do not want this result to be invalidated by the addition of new processes. We call the property *dynamic extendability*. 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.

2.4 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. Producing an MIS means each processor sets a flag 0 or 1 where the 0 output means the process is not in the MIS, and the 1 output means the process is in the MIS.

Lemma 2.1 *The MIS problem is dynamically extendable.* □

2.5 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.

k-wait-freedom not only protects against stopped processors far away in the network; it also protects against slow ones. If a k-wait-free algorithm has i producing an output in $O(r(\mu+\nu))$ steps, in fact the global μ and ν can be replaced by μ_i and ν_i where these are the maximum delays among links and processors in the k-neighborhood of i.

3 Review of Luby's protocol

Luby's synchronous MIS protocol, as given in [10] 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 $d_j \geq d_i$, 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 The difficulty of asynchrony

Even in a static 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 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. One could instead try adding, for example, the α synchronizer of [2] to generate global pulses, but this may add an overhead of $O(D)$ for the first step in the protocol, where D is the diameter of the network. In the dynamic case, this $O(D)$ could have to be paid repeatedly.

4 The new MIS protocol

In this section we present the new asynchronous MIS protocol. For ease of presentation, in this section and

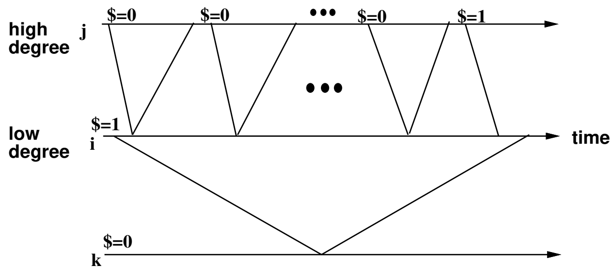

Figure 1: **The difficulty of asynchrony.** Why Luby's protocol does not work in the asynchronous network. Process j with degree bigger than that process i communicates quickly with i, while the link between i and k is slow. We cannot have j just freeze, whenever i is waiting to to hear from k, without 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.

the next, we will present the protocol for the stable model (see section 2.3). We then talk about extensions to an infinite dynamic measure of complexity for dining philosophers at the end of section 6.

The protocol presented will include messages where processors send the value of their degrees in the graph. Thus the messages will be of size $O(\log n)$. We show an easy way to reduce this to messages of size $O(\log \log n)$, while only increasing the expected running time by a constant factor, below.

4.1 Main ideas

Somehow, without paying for the delays of a global network pulse, we would like to control the flipping rates of neighboring processors: namely, a process i which flips a 1 would like its neighbors to have flipped 0, to "freeze" and not try to enter the MIS until it checks to see if it survives. Except, if we allow each of j's neighbors to freeze j in turn, j can stay frozen a long time with no chance to enter the MIS. This can lead to deadlocks.

Our solution lies in two simple ideas, which we call

- The snapshot freeze
- The near-equivelant rate.

The snapshot freeze means that instead of allowing processes i to each freeze j in turn, when j flips a 0, j will take it upon itself as a "public citizen" to take a "snapshot" look at the current coins of all its neighbors (it queries each neighbor to find out the current value of the neighboring coin, at the time when the query reaches the neighbor). Based on coins in this snapshot, j will freeze until the last neighbor i which had a 1 has entered the MIS and flipped a new coin (and the snapshot is expired for j). *If any neighbor flips a new coin after the snapshot is taken which is a 1, j does not freeze further for this new coin.* When j is unfrozen by its tardiest neighbor, j flips again.

The near equivelant rate, means that we will show based on the snapshot freeze, while we will not be able to insure that each neighbor flips once to every one of i's flips, we will be able to insure that each neighbor in some sense has only two chances to flip against each one of i's flips.[2] Thus, we adjust the probability that a node tries to enter the MIS down by a constant factor from Luby's algorithm.

4.2 The code

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

d represents the degree of process i. It is updated when i flips a coin. It is set to ∞ if i enters the MIS.

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 "to the best of i's knowledge", ($d_j < d_i$ and coin $= 1$) or (j's coin $= 0$); and 1 if, to the best of i's knowledge, ($d_j \geq d_i$ 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.

[2] We say "in some sense" because as will be seen in the analysis section, there is interaction between the random coinflips of the protocol and the link delay function, so that which two flips are the two flips which count can be influenced by the adversary; we show however, that the probability of a neighbor killing a process's 1 will still only increase from the synchronous protocol by a constant. See the analysis section for details.

(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 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 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.

The code for a process i is shown in below in figure 2.

4.3 Safety

Lemma 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 $d_j \geq d_i$, 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 $winner_i = winner_j = 1$. Assume without loss of generality, that t_i is before t_j. Notice that $winner_i$ is the flip on which i enters the MIS, by definition, and so i doesn't flip again, and so i doesn't update its degree d after time t_i (except to maybe enter the MIS). Thus until i enters the MIS, i has $d = d_i$ the value of its degree at time t_i, which is fixed. Now define t_{i+q} to be the time at which i's query to find out the value j's current coin reaches j. There are several cases.

1. t_j is before time t_{i+q}. Notice, as above, that j's degree becomes fixed at time t_j, and then doesn't change. Denote its degree at time t_j and following by d_j. At time t_{i+q}, j sends a message to i saying its coin is 1, and degree is d_j. When i receives this message, i sets the flag Coin(j), and i will not update its setting of this flag again, since by assumption j doesn't flip again. Thus, since i enters the MIS, it must have set Coin(j) to 0, which implies that $d_i > d_j$. But j must also query i after time t_j, and set the flag Coin(j) before it can enter the MIS. But since t_i was before t_j, j will learn that i flipped a 1, and i's degree is d_i, or that i is in the MIS, and its degree is ∞. In either case i's degree is greater than j's so j will set its flag $Coin(i)$ to 1. But this prevents j from entering the MIS with coin $winner_j$.

2. t_j is after time t_{i+q}. Define t_{j+q} to be the time at which j's query to find out the value i's current coin reaches i. Notice that both i's degree and j's degree are fixed after time t_j: call their degrees at this time d_i and d_j. If i has already entered the MIS by time t_{j+q}, then i will have 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. Otherwise, if j queries with $d_j \geq d_i$, since it is also the case that j's coin $winner_j$ is 1, i *resets* coin(j) to 1 (see protocol): j never flips again by definition, so the flag coin(j) will prevent i from entering the MIS. Otherwise, $d_j < d_i$, and so j will set coin(i) to 1 (and never reset it since i never flips again) and this prevents j from entering the MIS with coin $winner_j$. □

5 Analysis of the algorithm

We now show, for the protocol of the previous section, that a constant fraction of the processors expect either enter the MIS or have some neighbor enter the MIS within a constant times the length of the maximum link delay to distance two of the process. We present the analysis for the stable model (see section 2.3), which means that we can measure performance *after* the active network has stabilized, so all process degrees can assumed to be fixed. We sketch how to generalize this to a protocol for dining philosophers with a infinite dynamic measure of complexity, at the end of section 6.

Before delving into the analysis, to motivate the reader to stick with us through a careful probabilistic analysis, we wish to give some intuition of the sort of interaction between coins and the link delay function that can make the analysis difficult. Figure 3 gives an example where the timing of i's next coinflip can be influenced by processor j in a negative way: if processor j flips a 1, then i reflips immediately, however if processor j flips a 0, the link delay function can force j to flip again and have a new coin by the time i flips again. Thus i can be more likely to flip again when a neighbor is holding a 1 coin than a 0 coin, and since this increases the possibility that i's 1 coin is killed and i does not then enter the MIS, the reader should be worried about flips and link delay interactions. In Lemma 5.7 below, we bound the number of extra flips per coinflip the situation in Figure 3 can cause. First we prove some additional lemmas.

Lemma 5.1 (The Flip Again Lemma). *Let ω_j be an upper bound on the maximum link delay plus process delay to distance 2 from j in the graph, Then j gets*

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

C = WAKEUP_i(N)
  Effect:       Neighbors ← N
                d ← |N|
                ∀j ∈ Neighbors put in send-buffer(j)
                Coin = 0
                ∀j ∈ Neighbors
                  Coin(j) ← UNSET
                  Freeze(j) ← UNMARKED
                  put (Query,Coin,d) in send-buffer(j)

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

Flip-Coin
  Precondition: ∀k Freeze(k) = 0
  Effect:       ∀j s.t. update-flag(j) = 1, Neighbors ← Neighbors - {j}
                d = |Neighbors|
                Coin = 1 with probability 1/8d
                     = 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
  d ← ∞
  ∀j ∈ Neighbors put InMIS in send-buffer(j)
```

Figure 2: MIS Algorithm

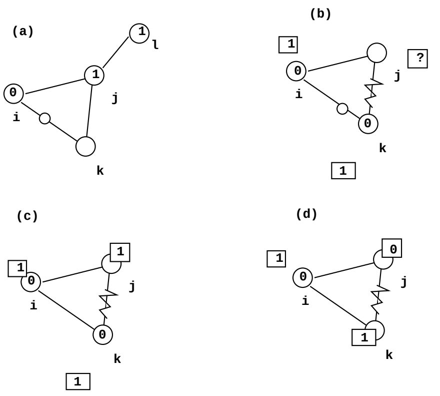

Figure 3: **A pathological case.** Assume all nodes have the same degree. The link from i to k is slow. (a) i has flipped a 0, and needs to talk to all neighbors and hear back, in order to know whether to flip again or freeze. i has already been frozen by j's 1, and then unfrozen again, as j has heard from all j's neighbors, and been prevented from entering the MIS by l. (b) i's message still has not reached k. k now flips again, and flips 0. j now flips again, and k will take his current coin value from j's new coin. (c) If j flips a 1, then k's current coin will freeze for j at 0. Thus i will not freeze for k, i will flip a 1, and be killed by j. (d) If j flips 0, however, j will not freeze k, which can then flip 1. i will now freeze for k. When k's 1 is killed (not pictured) i will flip again, and j will have a *second* chance to try to kill i's coinflip, even though j has already "lost" by flipping a 0..

to flip again (i.e. try to enter the MIS) in time at most $5\omega_j$, if neither j or any of its neighbors have yet managed to enter the MIS.

Proof. Omitted. $\Box$

Definition 5.2 *Suppose processor i flips a 1. Process i then queries all neighbors to find out their current coins, at the time that the query reaches them. Define c_j be this value of j's current coin as he reports it to processor i and define $next_j$ be the value of j's coin on j's next flip.*

Lemma 5.3 *Suppose processor i flips 1. Then if c_j and $next_j$ are both 0 for all neighbors j of i such that $d_j \geq d_i$, then i will enter the MIS in time $2\omega_j$.*

Proof. Omitted. $\Box$

Definition 5.4 *Define the function $1_i(t)$ to be the time it takes for i to flip again, if i flips a 1 at time t.*

Proposition 5.5 *$1_i(t)$ depends only on the delay functions.* $\Box$

Proposition 5.6 *The function $1_i(t)$ is a lower bound on how soon i flips again, if i flips a 1 at time t.* $\Box$

Lemma 5.7 *Consider any possible execution of the protocol, where at time t_0 processor i flips 0. Consider i's next flip, and let c_j be defined as above, in relation to processor i's next flip. Then the probability that c_j is 0 for all j such that $d_j \geq d_i$ is at least*

$$\prod_{j\,|\,d_j \geq d_i} (1 - \frac{1}{8d_j})^2.$$

Thus the probability that c_j and $next_j$ are both 0 for all j such that $d_j > d_i$ is at least

$$\prod_{j\,|\,d_j \geq d_i} (1 - \frac{1}{8d_j})^3.$$

Proof. Fix a partial execution E to time t_0 We give an algorithmic definition for a set of executions E', which are a subset of the executions compatible with E. (We remark that the construction of E' is needed for analysis only: it is defined in terms of the functions $1_i(t)$ which are defined in terms of the link delay functions.)

We then show that for executions in E', c_j is 0 for all j such that $d_j \geq d_i$. We then show that an execution falls in the set E' with probability at least $\prod_{j\,|\,d_j \geq d_i}(1 - \frac{1}{8d_j})^2$.

Here's how we define an execution in E'. At time t_0, compute $1_i(t_0)$. (This depends only on the delay functions by Proposition 5.5 so is computable independent of later coin tosses). We now simulate the protocol from time t_0 up to time $1_i(t_0)$ with processes j flipping coins as required as follows:

- For j such that j is not a neighbor of i, or a neighbor j for which $d_j < d_i$, we flip any new coin as before: 1 with probability $1/8d_j$ and 0 otherwise.
- For neighbors j such that $d_j \geq d_i$, if j is to flip a coin at time t_k, we ask if $1_j(t_k) > 1_i(t_0)$. If not, j just flips an ordinary coin as above. If so, however, j deterministically sets its coin at t_k to 0.

Now consider the resulting execution to time $1_i(t_0)$. The time when i flips again, and thus the time when i learns c_j occurs after $1_i(t_0)$ by Proposition 5.6. *Notice, however, that this time is completely determined by the delay function and coins already flipped by time $1_i(t_0)$* (This is because i will freeze or not freeze based on the values of neighbors coins, and it has heard back from all neighbors by time $1_i(t_0)$. Neighbors it freezes for hold 1s, which never freeze in turn; thus if they do not enter the MIS, they will flip again and unfreeze i in time dependent only on the link delay functions). Call t_j^{E*} the time at which i learns the coin c_j. (The superscript $E*$ is simply a reminder that this time does depend on coins flipped before time $1_i(t_0)$.)

Here is the rest of the simulation from time $1_i(t_0)$, where processors j flip coins as follows:

- For j such that j is not a neighbor of i, or a neighbor j for which $d_j < d_i$, we flip any new coin as before: 1 with probability $1/8d_j$ and 0 otherwise.
- For neighbors j such that $d_j \geq d_i$, if j is to flip a coin at time t_k, we ask if it is the first time for j that $1_j(t_k) > t_j^{E*}$. If not, j just flips an ordinary coin as above. If so, j deterministically sets its coin at t_k to 0.

This ends the definition of E'.

Claim 5.8 *An execution will belong to the class E' with probability at least $\prod_{d_j > d_i}(1 - (1/8d_j))^2$.*

The first claim is true, because for each such j, we are deterministically setting at most two coins, one coin flipped before time $1_i(t_0)$ and one coin flipped after. We only set one coin after $1_i(t_0)$ by definition, we set the first coin such that $1_j(t_k) > t_j^{E*}$. Before time $1_i(t_0)$, we set a coin 0 if $1_j(t_k) > 1_i(t_0)$. We claim that the next coin j flips must be flipped after time $1_i(t_0)$, since $1_j(t_k)$ is a lower bound on the time j will flip again by Proposition 5.6.

Claim 5.9 *In an execution defined above, c_j will be 0 for all j with $d_j \geq d_i$.*

To see the second claim, we work backward. If the coin c_j was flipped at time t_k after time $1_i(t_0)$, then it must be the case that $1_j(t_k) > t_j^{E*}$, otherwise j would flip again before time t_j^{E*} and a later coin would be c_j. Also c_j must be the first coin for which $1_j(t_k) > t_j^{E*}$, because if there was an earlier coin flipped at time t_y for which $1_j(t_y) > t_j^{E*}$, then that coin would still be current, by Lemma 5.6. Thus we have set $c_j = 0$. If the coin c_j was flipped before time $1_i(t_0)$ but after time t_0, then the coin that was current is the coin that was current after time t_0, which we have set to 0, by design. Finally, if j does not flip a coin at all between times t_0 and t_j^{E*} then c_j is the coin current at time t_0. Can this be a 1? No, because then j has to flip again before time t_j^{E*} in any execution, because such a 1 would be current at time $1_i(t_0)$ and thus freeze i.

Finally, $next_j$ is the next coin processor j flips after c_j, an independent cointoss which is 0 with probability $(1 - 1/8d_j)$ for each j. □

Now an analysis similar to [10], using Lemma 5.7 that for a constant fraction of the nodes i, we expect that some neighbor k of i flips 01, while all same or higher degree neighbors j of k flip c_j, and $next_j$ all 0. (Details omitted for lack of space, they appear in the full paper). Then Lemma 5.3 guarantees that k will enter the MIS at time $t + 2\omega_j$ *entirely independent of variation in link delays.* Lemma 5.1 showed that a node k will always reflip in time $5\omega_j$, again entirely independent of variation in link delays. Thus independent of the link delay function, a constant fraction of the nodes will enter the MIS in time 7ω (5ω time to re-flip, and then a delay of 2ω to get ACKS from all neighbors.) Notice also, from Lemmas 5.1 and 5.3, that when a node j will enter the MIS is only dependent on ω_j, its local link and process delay to distance 2 in the graph. Normalizing maximum link and process delays to lie between 0 and 1, we thus have proved the following theorem.

Authors	Response time	Wait-dependency
[12]	$O(c^\delta)$	$O(\delta)$
[5]	$O(n)$	$O(n)$
[17]	$O(\delta^{\log \delta})$	$O(\log \delta)$
[3]	$O(\delta^2)$	$O(\delta)$
This paper	$O(\delta)$	2
Lower bound	$\Omega(\delta)$	1

Figure 4: Our Resource allocation algorithm versus existing ones; c is the number of colors used to color the network graph, n is the number of nodes in the network, and δ is the number of conflicting jobs.

Theorem 5.10 *The asynchronous protocol is 2-wait-free, and produces an MIS in $O(\log n)$ expected time.* □

We remark the above protocol uses only constant space per edge (each process only stores a (0,1) value for j's coin, and a 3-valued (0,1, or UNSET) special flag called Freeze, for each neighbor j. However, the above protocol involves sending messages of size up to $\log n$, since nodes communicate their current degrees in the communication graph. To reduce this, instead, each node could round its degree up to the next largest power of two, and try to enter the MIS with probability inversely proportional to the same constant times the rounded value, rather than its precise degree. This will only increase the expected running time by a constant; and the rounded degrees can be communicated with messages of size $\log \log n$.

6 Dining philosophers

6.1 History and definitions

The dining philosophers problem, and its extensions, model resource allocation in a distributed system. We consider the generalized dining philosophers problem, as modeled in [3], first in the stable model defined in section 2.3, and then extend this below. The dining philosophers problem, studied by Dijkstra and others [7, 12, 16, 5, 11, 3, 6], models a set of resources (such as printers, disk drives) which can only be used by one competing process at a time. The situation can be represented by a graph on the set of processors called the *conflict graph*, with an edge between two nodes if they share some resource. In this formulation, which is the standard one, we are assuming that the *conflict graph* is a subgraph of the communication graph. The intuition behind the notion of a *conflict graph* is that if two jobs are local to the same disk, for example, then they are also local to each other, so they can communicate. Each processor may handle a sequence of jobs, but tries to schedule only one job at a time; each job has a resource requirement which is a subset of the resources accessible to that processor. For a job to be executed, all of the required resources must be available for exclusive use by its processor.

We are interested in bounding the *response time* of a job. The response time is the time between when a job is assigned to a processor, and when it is executed. We will modify the MIS protocol of the previous sections to construct a dining philosophers schedule with optimal expected response time of $O(\delta_j)$ for job j, where δ_j is the number of jobs that compete with j. This meets the lower bound, where the best known previous algorithms had response times $O(\delta^2)$. (The best known deterministic algorithms for dining philosophers are still [3] with a response time of $O(\delta^2 \log n)$ or [6] with a response time of $O(\delta^2)$ (special hardware assumptions)). It is also worth mentioning a recent paper of Bar-Ilan and Peleg [4] who have a variety of algorithms, including optimal algorithms for dining philosophers in the *synchronous* model.

We point out again that all the above bounds are stated under the normalizing assumption that job execution time, and the maximum link delay, or time it takes for a message sent by one processor to be received by its neighbor, are both less than 1 unit of time. If μ is an upper bound on the maximum length of time it takes a job to complete execution and ν is an upper bound on the maximum *link delay,* then our protocol has more precisely expected response time less than $O(\delta_j \mu + \delta_j \nu)$, for job j. (And the lower bound is similarly $\Omega(\delta_j(\mu+\nu))$). In fact, we get the *2-wait-free* property and ν and μ can be replaced by the local delays ν_1 and μ_1.

6.2 The reduction from asynchronous MIS

Using our asynchronous protocol, we get the optimal expected *response time* for dining philosophers. The relation between dining philosophers and MIS was noticed earlier by [4]. Our protocol is quite simple and works as follows. Each process with no neighbor in the MIS and with an unscheduled job, runs the MIS protocol of Section 4. When a process enters the MIS, it first sets its ID to ∞, so that none of its neighbors can enter the MIS while its job is running. The job is then scheduled. When the job is finished executing, it sends a done message to all its neighbors, which then remove that job from their neighbor set.

We now analyze our algorithm. We note that *safety* and follows from the corresponding proof of MIS; we now prove that a job j will execute in expected $O(\delta_j)$ time.

Theorem 6.1 *Let $\nu(k)$ be an upper bound on the link delay of the neighbors of job k, and let ν_j be the max. over all neighbors k of job j of $\nu(k)$. Let μ_j be an upper bound on the amount of time it takes to execute any of j's neighboring jobs. Then job j will execute in $O(\delta_j(\nu_j + \mu_j))$ time.*

Proof: Let E_0 denote the event in Lemma 5.7, that is, j flips 01, and c_i, and $next_i$ are both 0 for all neighbors i of j such that $d_i \geq d_j$ then,

$$\begin{aligned}
\Pr[E_0] &\geq \frac{7}{8}\frac{1}{8\delta_j} \prod_{k \in N(j), \delta_k \geq \delta_j} (1 - 1/8\delta_j)^3 \\
&\geq \frac{7}{16}(\frac{1}{8\delta_j}) \\
&\geq \frac{1}{32\delta_j}.
\end{aligned}$$

If E_0 occurs, then by Lemma 5.3 j will enter the MIS in $2(\mu_j + \nu_j)$ time. Thus with normalized link delays j gets scheduled in expected $32\delta_j$ time, which is $O(\delta_j)$. The link delays, by Lemmas 5.1 and 5.3, are bounded by ν_j; and the execution delay of j's neighbors is bounded by definition by μ_j; yielding the result. □

We remark that the dependence on local link delays and job execution times in Theorem 6.1 shows that our dining philosophers algorithm is 2-wait-free.

6.3 The infinite model

We now discuss extending this protocol to the dynamic non-stable case considered by [3] (see also section 2.3), where new jobs can be added online to the neighbor set of old jobs throughout the protocol, and we require that a job j is scheduled in time $O(\delta_j)$, where δ_j is the maximum number of jobs that conflict with j at any point in time. Because of space requirements, we will only be able to give a sketch of the modifications. Safety is proved in a fully dynamic model for the stable case already, so there is nothing to be argued in terms of correctness. We need to show that a job still expects to be scheduled in time $O(\delta)$. If we try to follow the same analysis as for the stable case, Lemma 5.1, and Propositions 5.5 and 5.6 go through unchanged. Lemma 5.3 is replaced by a more complicated statement where up to two coins per neighbor, both of which are flipped when the neighbor's degree is greater than processor i, are set to 0 implies that i will enter the MIS. However, when we try to present a subset of legal executions which will set the proper coins to 0, to mimic the argument in Lemma 5.7, we run into trouble because since at t_0, the time at which processor i flips again can be influenced by how we set subsequent coins, we do not know processor i's degree when i flips again!

We fix this by modifying the protocol: when new processes first come in, a non-new process i's degree is *not* updated to account for them on i's next coin flip, but rather on i's following coinflip. Unitl all new processes non-new neighbors have flipped again twice, new processes have their coins set deterministically to 0. We make three remarks. First, this modification does not influence safety (as new processes all hold 0 coins), Second, i's degree is always set less than or equal to its actual degree in the conflict graph, but even when not yet updated to reflect new neighbors, it is set greater than or equal to the number of neighbors that can compete with i on i's current coin flip. Third, process i knows what its degree will be at its next coin flip when it flips its current coin, so we can mimic the argument in Lemma 5.7 (see the full version of the paper). However, note that since a new process can have a non-new neighbor frozen at 0, this means the delay for a new processor to initially begin competing can depend up to delays up to distance 3 in the graph. Thus the protocol is now 3-wait-free for new jobs to enter the protocol, and 2-wait-free thereafter.

Acknowledgement

Many thanks to Nancy Lynch for most helpful comments and suggestions, along with careful reading of preliminary drafts, and in particular, for her help in nailing down a precise asynchronous model and a formal framework for probabilistic analysis. Thanks to Mike Saks for helpful comments.

References

[1] B. Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, Oct. 1985.

[2] B. Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, Oct. 1985.

[3] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st IEEE Symp. on Found. of Comp. Science*, 1990.

[4] J. Bar-Ilan and D. Peleg. Distributed resource allocation algorithms. In *6th International Workshop on Distributed Algorithms*, Nov. 1992.

[5] K. Chandy and J. Misra. The drinking philosophers problem. *ACM TOPLAS*, 6(4):632–646, October 1984.

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

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

[8] R. G. Gallager, P. 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, Jan. 1983.

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

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

[11] J. Lundelius and N. Lynch. Synthesis of efficient drinking philosophers algorithms. unpublished manuscript, Jan. 1988.

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

[13] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6(2), 1992.

[14] N. Lynch and R. Segala. Notes on probababilistic automata. Unpublished manuscript., 1993.

[15] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[16] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th POPL*, pages 133–138, 1981.

[17] E. Styer and G. Peterson. Improved algorithms for distributed resource allocation. In *Proc. 7th ACM Symp. on Principles of Distrib. Computing*, pages 105–116. ACM SIGACT and SIGOPS, ACM, 1988.