Expected Robustness in Dining Philosophers Algorithms

A Senior Honors Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Bachelor of Science with Distinction in Computer and Information Science

The Ohio State University

By

Daniel Galron,

* * * * *

The Ohio State University

2006

Thesis Committee:

Paul A. G. Sivilotti, Advisor

Approved by

Advisor

Department of Computer Science and Engineering

TABLE OF CONTENTS

Page

Chapters:

CHAPTER 1

Introduction

The dining philosopher's problem is a classic synchronization problem that has been extended to the realm of distributed resource allocation. The dining philosopher's problem is an abstraction of the problem where processes running on a system compete for shared resources (resources such as printers, memory, hard disk space, or wireless frequency). In the dining philosopher's problem, there is a set of philosophers with a neighbor relation on the set. Two philosophers may be neighbors because they share a resource. The philosophers have three states: *thinking*, *hungry* and *eating*. Initially, all philosophers are thinking. Philosophers may become hungry, and upon doing so they must eventually eat (i.e., transition into the *eating* state). There are two requirements on solutions to the problem. One is *mutual exclusion*, the notion that, in the presence of some global clock, no pair of neighboring philosophers may simultaneously eat. In a real-world system, this is analogous to the mutual exclusivity of a resource, for instance that no two processes may share the same memory space. The other is *progress*, the notion that all philosophers who become hungry eventually eat. In order to guarantee mutual exclusivity, the notion of forks are used. Every pair of neighboring philosophers shares a single fork. In order to transition from the *hungry* state to the *eating* state a philosopher must hold all shared forks. Furthermore, no

two philosophers may simultaneously hold a fork. For the purposes of presenting the algorithms in this thesis, the problem is represented in graph-theoretic terms, where philosophers are represented as vertices, and where there is an edge between vertices which share a fork.

There are several algorithms that solve this problem. Some algorithms optimize for *response time*, a metric that measures how quickly a hungry philosopher gets to eat. Others optimize for *message complexity*, a metric that measures the number of messages sent as a result of a philosopher becoming hungry. Still others optimize for a metric called *failure locality*, which is a measure of the robustness of a system. When a philosopher crashes its immediate neighbors may not be able to eat should they become hungry. In turn, their neighbors may be unable to eat, and this effect can cascade throughout the entire system. The state where a philosopher is unable to eat indefinitely despite being hungry is called *starving*. Failure locality is a measure of the spread of starving processes due to a crashed philosopher. It is this latter class of algorithms that we examine in this thesis.

The class of algorithms that optimize for failure locality measures it in terms of *worst-case failure locality* — that is, the distance between the failed philosopher and the furthest starving philosopher under the most pessimistic failure scenario. For some algorithms, such as Chandy and Misra's hygienic algorithm ([1]), the worst-case failure locality is the entire diameter of the graph. For others, such as Choy and Singh's bounded doorways algorithm ([3]), the worst-case failure locality is 2 (i.e., the farthest philosopher in the graph which cannot eat is only two edges away from the crashed philosopher). Oftentimes, however, the worst-case failure locality

is not an accurate representation of how failure spreads in average case situations. Partly, this is because there are no good metrics defined to measure the average-case behavior. Furthermore, trying to discover the average case behavior analytically is often infeasible, since there are many algorithm-specific effects which impact the average failure locality.

In the course of our research, we wanted to measure the average case failure locality for several algorithms purported to limit the spread of starvation throughout a system. To do this, we derived a new metric to measure the average case failure locality, called *Integrated Failure Locality (IFL)*. IFL is defined to be the sum of the ratio of starving to non-starving philosophers over successive distances, or:

$$IFL = \sum_{i=1}^{\Delta} \frac{\#\text{ of starved processes within distance } i \text{ of failed node}}{\text{total }\#\text{ of processes within distance } i \text{ of failed node}} \tag{1.1}$$

In order to calculate the IFL for the algorithms, we implemented simulations using the AnyLogic 5.0 simulation toolkit and engine. These simulations implement the algorithms, as well as methods to collect statistics to calculate the IFL of the algorithms.

We implemented simulations of six algorithms: the hygienic algorithm [1], the thresholds algorithm [6], the biserial and strict biserial algorithms, the double doorway algorithm [2], and the bounded doorway algorithm [3]. As it turned out, the thresholds and biserial algorithms (which have worst-case failure locality 2) had the best performance on average, with an IFL close to 1. However, the bounded doorway algorithm, which also has worst-case failure locality 2, had better IFLs than the thresholds algorithm under certain failure scenarios. Predictably, the double doorway

algorithm, which has worst-case failure locality 4, has a fairly high IFL, higher than the worst case failure locality of the thresholds and double doorways algorithm.

In this thesis, we present our work and analysis of the average case failure locality of dining philosophers algorithm. In Chapter 2 we elaborate the dining philosophers problem, and present the hygienic solution. In Chapter 3, we present and justify our new failure locality metric, IFL. In Chapter 4, we outline our experimental methods and describe the simulation framework we implemented for evaluating the algorithms. In Chapters 5-8, we describe the algorithms, present our data measuring their IFLs, and discuss the algorithmic properties and effects that lead to the observed results. Finally, in Chapter 9, we summarize our discussion of the analyses of the algorithms' IFLs.

CHAPTER 2

Background

2.1 The Dining Philosophers Problem

The problem of distributed resource allocation and synchronization consists of a set of processes sharing a set of resources. As part of their execution, the processes may request access to needed resources which are allocated in accordance to two properties: a single resource cannot be simultaneously used by multiple processes (*mutual exclusion*), and every resource request must eventually be satisfied (*progress*). After a process obtains its requested resources, the process may use the resources for a finite amount of time, after which it relinquishes them so that other processes in the system with which it shares the resources may use them.

This problem has been formalized as the dining philosophers problem, originally defined by Dijkstra in 1971 [4] as a synchronization problem. In the original formulation of the problem, five philosophers are sitting around a table all sharing a plate of food. To the left and right of each philosopher, there is a fork, and in order to eat the food, a philosopher must acquire both the left and the right fork. Note that the left fork of a philosopher is the right fork of another, and the right fork of a philosopher is

the left fork of another. That is to say, philosophers that sit next to one another share a single fork. A philosopher's life cycle transitions between three states: thinking, hungry, and eating. Initially, all philosophers are thinking. After some time period, a philosopher may become hungry. When it becomes hungry, it tries to acquire the two forks which it shares with its neighbors. Once it acquires all its forks, it eats, after which it must relinquish forks to neighboring hungry philosophers.

Dijkstra's formulation of the problem is an abstraction of the problem of distributed resource allocation. A philosopher corresponds to a process, and a fork corresponds to a mechanism to ensure mutual exclusion between processes contending for a set of resources. Thus, a pair of neighboring philosophers that share a fork is analogous to two processes competing to acquire a resource. The three states of thinking, hungry, and eating correspond to the life cycle of a process: *thinking* is analogous to not using a resource or set of resources; *hungry* is analogous to requesting a set of resources; and *eating* is analogous to using a set of resources.

In this formulation, mutual exclusion is ensured by requiring that no pair of neighboring philosophers (i.e, philosophers sharing a fork) both simultaneously hold a fork, and progress is ensured by requiring that all hungry philosophers eat within a finite time period. There are two situations that must be avoided in order to ensure progress. The first is *deadlock*—a situation which arises when a collection of philosophers are simultaneously waiting upon forks from one another, but which none will acquire. For example, if, in the case of five philosophers sitting around a table, all are hungry and all hold the fork that they share with the philosopher to their right, all the philosophers are deadlocked, since they are all waiting to acquire the fork that they

share with their neighbor to the left. The second is *livelock*—the situation where one or more philosophers conspire to become hungry in such a way that prevents other philosophers from eating. For example, if one philosopher repeatedly becomes hungry immediately after last eating, its neighbors may not be able to acquire the forks they share with it, and thus they will not be able to eat if they become hungry. A simple solution to ensure progress (this is, in fact, the solution that Dijkstra proposed) is to introduce an asymmetry when collecting forks. This can be done by designating a single philosopher to first request its left fork when it becomes hungry, while having all the other philosophers first request their right forks. This would therefore prevent the deadlock situation where all philosophers are hungry and all hold a single fork (see Figure 2.1).

In order to make this more useful for distributed resource allocation, Dijkstra's formulation of the problem has been generalized to extend to arbitrary topologies—instead of five philosophers sitting around a table, we instead have an undirected graph, where vertices represent philosophers and edges indicate which philosophers are neighbors (see Figure 2.2). This subsequently means that every pair of adjoining vertices (i.e, neighboring philosophers) share a single fork. Such graphs are called conflict graphs. Notice in the figure, that all the forks are near a philosopher. Forks are always held by philosophers, even if they are not being used; that is, thinking philosophers continue to hold forks as long as the forks have not been requested by other neighboring philosophers.

A *dining philosophers algorithm* is a solution to the dining philosophers problem that ensures mutual exclusion and progress. Typically, these solutions involve schemes

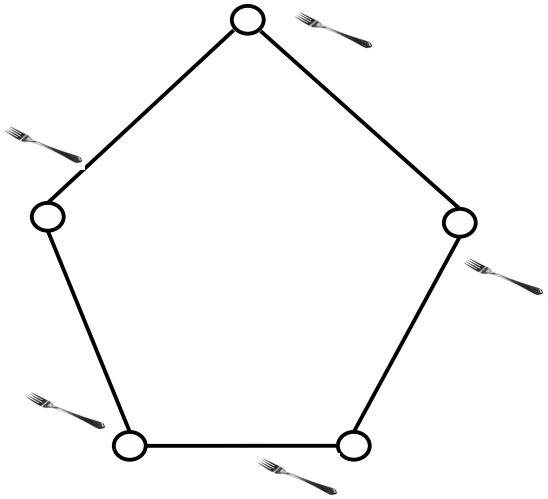

Figure 2.1: Conflict graph representation of original formulation of dining philosophers problem

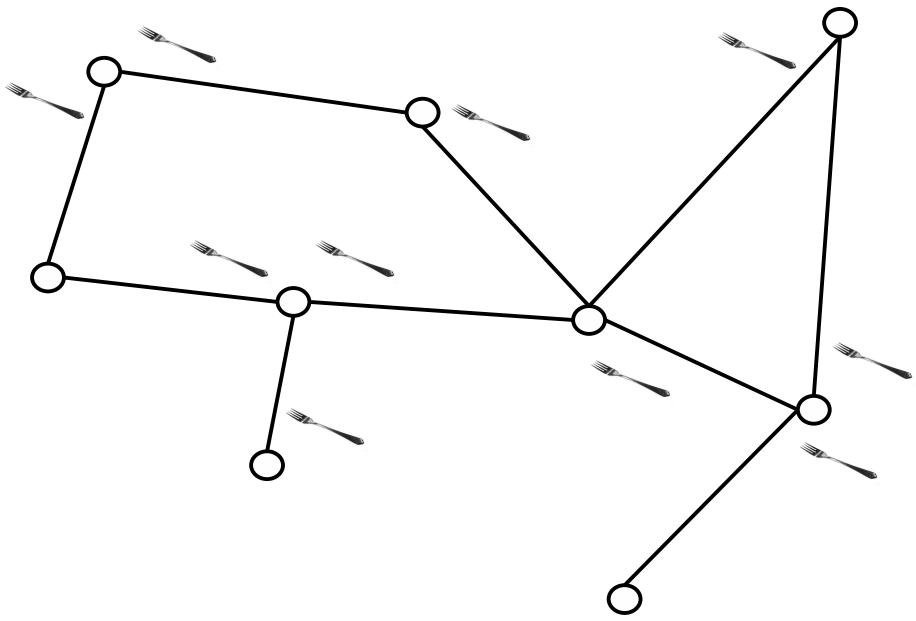

Figure 2.2: An arbitrary conflict graph

to acquire forks and arrange philosophers in such a way to ensure the two criteria. All six algorithms examined in this thesis satisfy these two criteria.

2.2 The Hygienic Solution

The first dining philosophers algorithm presented is called the hygienic solution [1]. In order to ensure progress, the hygienic solution introduces asymmetry by assigning a priority to each edge in the graph. This assignment must be done with care, however, not to introduce a cycle in the priorities of the edges. This imposes a partial order on the undirected graph—that is, while the edges may be undirected, this priority partial order has direction (see Figure 2.3). Thus, a directed edge marks priority. If there is a directed edge between philosopher u and philosopher v, then v is higher in the partial order, and v has higher priority than u. For example, in Figure 2.3, node B has higher priority than A, since there is a directed edge leading from A to B. For nodes that do not have an edge between them, such as A and C, neither has priority over the other. We notationally represent priority with a $<$ symbol; in Figure 2.3 we represent the priority relationship between A and B as $A < B$. There are two requirements for this priority scheme. The first is that a *conflict* for a fork is always resolved in favor of the higher-priority process. By conflict, we mean the situation where two neighboring philosophers are hungry. The second is that after a philosopher gets to eat, it lowers its priority to allow its neighbors to acquire its forks. Priority is only applied in the case where for a philosopher neighbor pair, both philosophers are trying to acquire the same fork. Only in that situation will the priority scheme arbitrate the conflict. If for a pair of philosophers, only the low-priority philosopher

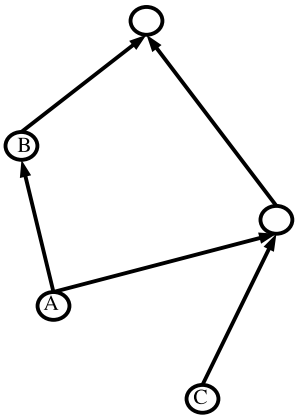

Figure 2.3: An example of a directed partial order.

wishes to acquire its fork, it may certainly obtain it if the fork is not under contention from the other philosopher.

This priority scheme is implemented by assigning a flag to each fork marking it as either *clean* or *dirty*. Priority is defined in terms of this flag. For a pair of two philosophers u and v, $u < v$ if u and v share a fork and v holds the clean shared fork or u holds the dirty shared fork (see Figure 2.4). Formally put,

$$u < v \equiv (fork(u, v) = v \wedge clean(u, v)) \vee (fork(u, v) = u \wedge \neg clean(u, v))$$

Then, in terms of clean and dirty forks, there are four important properties that must hold:

1. Eating philosophers hold all their shared forks, and all their shared forks are dirty. This ensures that when a philosopher finishes eating, it has the lowest priority amongst its neighbors, meaning that it is at the bottom of the partial order.

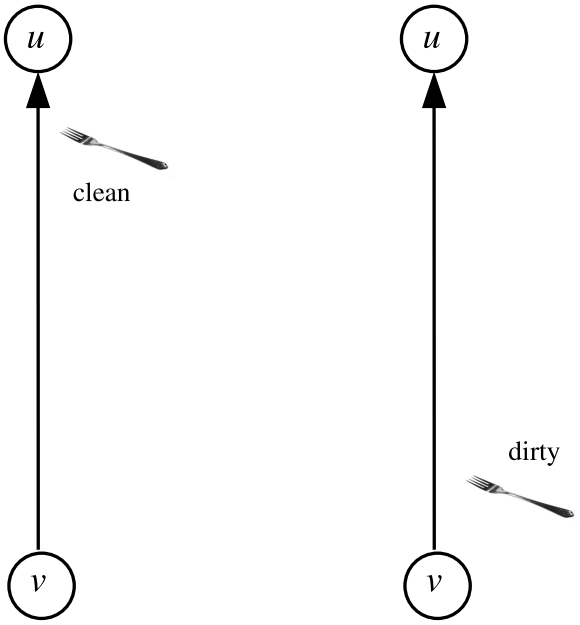

Figure 2.4: Relation of priority to flag on the shared fork

2. Philosophers do not relinquish clean forks—that is, they do not release forks to lower-priority neighbors.

3. Before a philosopher relinquishes a dirty fork, it sets the flag on the fork to "clean", thus ensuring that the direction of the edges in the partial order does not change.

4. Clean forks are only held by hungry philosophers, in order to ensure that if a thinking philosopher receives a request for a fork, it will always relinquish that fork.

Before presenting the algorithm, I discuss the notion of tokens. The hygienic algorithm has two tokens—a fork token (which corresponds to a shared fork) and a request token. These tokens can be thought of as being passed between nodes along

the edges connecting them. The request token is used to request the fork from the particular neighbor with whom the hungry process shares that request token and fork.

Now we present the hygienic solution. Initially, all philosophers are thinking, all forks held are dirty, and the forks are arbitrarily distributed to impose an acyclic partial order on the graph. The algorithm goes as follows: upon becoming hungry, a philosopher sends request tokens to all its neighbors whose shared forks it does not hold. Once it acquires all the forks, and when either all the forks are clean or if the fork is dirty but has not been requested by the neighbor, the philosopher eats and sets all its forks to dirty. Upon receiving a request token, a philosopher relinquishes the corresponding fork if the fork is dirty and if the philosopher is not currently eating. This means that it will always relinquish a fork when it is thinking, never when it is eating, and will only relinquish forks while hungry if the requesting process has higher priority.

Before presenting the formal algorithm, we discuss the notation used. The formalized algorithm has three parts: a section describing the initial configuration of the system, a section defining terms, and the definition of actions to be performed when certain predicates are met. The format for the definitions is as follows: $label\{conditions\ to\ be\ met\}\ action$.

Program	p
initially	$p.state = thinking$
	$clean(p, q) = false$
	Priorities form a partial order
always	$p.t \equiv p.state = thinking$

$$p.h \equiv p.state = hungry$$

$$p.e \equiv p.state = eating$$

assign

H_p $\{p.h \wedge fork(p,q) = q\}$

$req(p,q) := q;$

E_p $\{p.h \wedge fork(p,q) = p \wedge (clean(p,q) \vee req(p,q) = q)\}$

$p.state := eating;$

$clean(p,q) :=$ false;

R_p $\{req(p,q) = p \wedge fork(p,q) = p \wedge \neg clean(p,q) \wedge \neg p.e\}$

$fork(p,q) := q;$

$clean(p,q) := \neg clean(p,q);$

2.3 Algorithm Evaluation

Traditionally, resource allocation algorithms have been evaluated with respect to two criteria: *response time* and *message complexity*. Response time is a measure of how long it takes for a philosopher to acquire all its forks and eat once it becomes hungry, and message complexity is a measure of how many messages are generated as a result of a request for forks. Both of these metrics serve as indicators of an algorithm's efficiency—the lower the values for the metrics, the more efficient the algorithm is. There are several ways of defining these metrics, the most common of which is in terms of asymptotic worst case function (i.e, Big-O notation). For the hygienic algorithm, the worst-case response time is $O(n)$. That is, in the worst case, the response time of this algorithm is a function of the number of philosophers in the

system. The worst-case message complexity for this algorithm is $O(\delta)$—a worst-case function of the connectivity of the node in the conflict graph with the most neighbors.

CHAPTER 3

Failure Locality

3.1 Definition

As mentioned earlier, we have examined the expected robustness of various dining philosophers algorithms under node failure. In the dining philosophers problem, when a philosopher crashes while holding a fork, the fork is irrecoverable; that is, when a neighboring philosopher requests a fork held by the crashed philosopher, the neighboring philosopher will not be able to acquire the fork, and will therefore starve. This effect could cascade and deadlock the entire system. For example, one could have a chain (see Figure 3.1), where the left-most philosopher has crashed; the neighbor to the right of the crashed philosopher holds its right fork, but is waiting for its left fork (which it will never receive); its right neighbor is waiting upon a fork, etc. so that all philosophers in the system are waiting upon the crashed philosopher, and thus starve.

Figure 3.1: A starvation chain, where the black node has crashed and the gray nodes are starving

Traditionally, robustness in distributed resource allocation algorithms has been defined in terms of *worst-case failure locality*—a measure of how far, in the most pessimistic scenario, a starvation chain can extend from a crashed process. In other words, worst-case failure locality is simply the distance between the crashed node and the farthest starving process in the graph for the most pessimistic failure scenario. This metric, however, does not work for the purpose of evaluating average or expected behavior of these algorithms.

We want to examine expected failure locality, to see how exactly these algorithms perform in reality. The worst-case measurement may not reflect the actual performance of an algorithm. For example, the worst-case failure locality of an algorithm may be a function of the cardinality of the graph (i.e, in the worst case, all philosophers may starve), but in practice, the conditions that would enable global starvation may in fact only occur very rarely. By examining the average performance of an algorithm, we can get a more realistic evaluation of how these algorithms perform.

Examining expected robustness required us to consider new metrics for quantifying this behavior. Our first thought was to use the average length of starvation chains. However, when considering expected case behavior, this metric overlooks some important elements. For example, suppose one has a graph, where a node with many neighbors has crashed. Suppose one of those neighboring philosophers (which has several neighbors itself) starves, one of its neighbors starve, and so forth. Then, a long starvation chain would ensue, thus giving the impression that the algorithm used was not very robust. However, in this scenario, most of the philosophers do not starve: only one philosopher in each successive distance from the failed node starves. The

length of the very long starvation chain would suggest that the performance is very poor. We needed a metric which somehow measures how many philosophers were effected by the crash.

One of the metrics we considered was *failure cardinality*—that is, to evaluate the robustness of the system by counting all the processes which will starve because of a failure. The main drawback of this is that it is this metric depends on the total number of philosophers in the graph. For example, if one algorithm has failure cardinality 100 on one graph topology with 10,000 nodes, and another algorithm has failure cardinality 20 on a different graph with 20 nodes, then, just by looking at the numbers, one would assume that the second algorithm performed better, even though, clearly, all the processes in the system starved.

This leads us to consider another metric, *normalized failure cardinality.* Normalized failure cardinality is the ratio of starved to total processes in a graph—that is, it is the percentage of starving processes in the graph. This metric clearly does not have the same problem as failure cardinality. Because it is normalized, we can compare the behavior of algorithms across topologies. This metric, however, leaves out some important information, namely, it tells us nothing about the distances of the starving processes to the failed node. Also, it does not allow us to compare expected behavior with worst-case behavior, since worst-case behavior is a measure of distance, whereas this metric is a measure of cardinality. The main problem with normalized failure cardinality is that the normalization does not quite work. If one adds nodes outside the starvation neighborhood (i.e, outside the maximum distance a starvation chain extends from a crashed process), the denominator increases, thus decreasing the value

of the normalized failure cardinality, making the performance of the algorithm appear better.

The next metric we considered was *multidimensional failure locality* (MFL). Before discussing MFL, we need to introduce some terminology. The first piece of terminology we introduce is *ring*. Classically, a ring is the set of nodes in the graph of a certain distance from a given node. We extend this definition to specify a set of nodes in the graph of a certain *minimum* distance from a given node—that is, if a node is both distance 1 and distance 2 away from a given node, it is said that the node is in the 1-ring of the given node. When we discuss rings, they will always be in terms of a failed process. Therefore, when we mention a 2-ring, we refer to the set of all nodes which are 2 "hops" away from the failed node. The other piece of terminology we use is neighborhood. A neighborhood is similar to a ring, except that all processes within a given distance to a failed node are in the neighborhood. For example, a 2-neighborhood would contain all the philosophers in the 1-ring and the 2-ring of the failed node. MFL is a multidimensional metric, which measures the ratio of starved to total processes within successive rings (i.e, the first element is the ratio within the 1-ring, the second element is the ratio within the 2-ring, etc). For example, in Figure 3.2, the MFL would be [0.5, 1, 1, 0, 0].

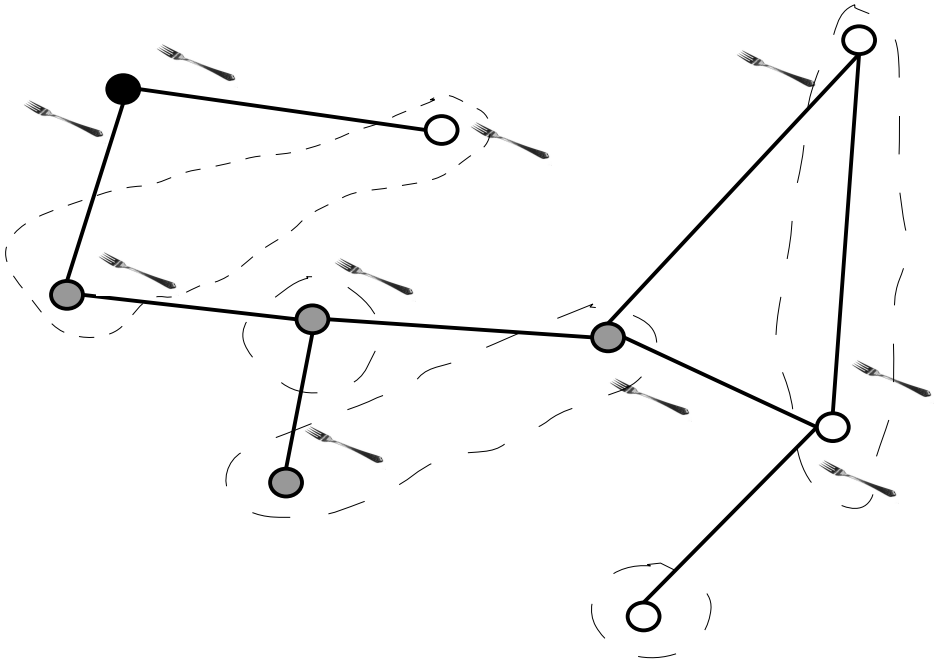

Figure 3.2: An arbitrary graph, where the black vertex is crashed, the gray are starving, and the white are operating normally.

Although this metric better embodies what we're trying to represent, there are several disadvantages, the most severe of which is the fact that this metric is multidimensional, and is therefore difficult to compare with worst-case failure locality. Furthermore, it would be very difficult to compare the performance of algorithms for which the distance of the farthest starving node is topology-dependent, since this metric would be impossible to normalize, since it would require elongating or truncating the length of the vector for comparison.

This leads us to the metric that we have decided upon, *integrated failure locality*. Integrated failure locality is based on multidimensional failure locality; it is simply the sum of the elements in the MFL vector. That is, it is the sum of ratios of starved

to total processes within successive rings from the failed node. In other words, IFL can be calculated by using the following formula (where Δ is the distance of the farthest ring to the crashed node):

$$IFL = \sum_{i=1}^{\Delta} \frac{\#\text{ of starved processes within ring } i}{\text{total }\#\text{ of processes within ring } i} \tag{3.1}$$

So, in Figure 3.2, we would get an IFL of 2.5. The obvious advantage of IFL is that it represents both a measure of the cardinality of starving processes and a measure of distance from the failed node. Furthermore, this allows comparison with worst-case failure locality, since we can represent the worst-case failure locality using IFL. In the worst case, the IFL will be equal to the traditional worst case failure locality, since in the most pessimistic scenario, all the processes within a given distance to the failed node will starve. Furthermore, because this metric has a single dimension, it is normalizable. (As we will see, however, we will never have to normalize an IFL value in the scope of this thesis).

3.2 Robustness of the Hygienic Solution

For the hygienic solution, the worst-case failure locality is O(n), that is, all the philosophers starve. To see why, imagine the following scenario. Suppose one has the priority partial order for a graph in Figure 3.3. Starting here, and for the rest of the thesis, we assume that the topology is an undirected version of the following partial order:

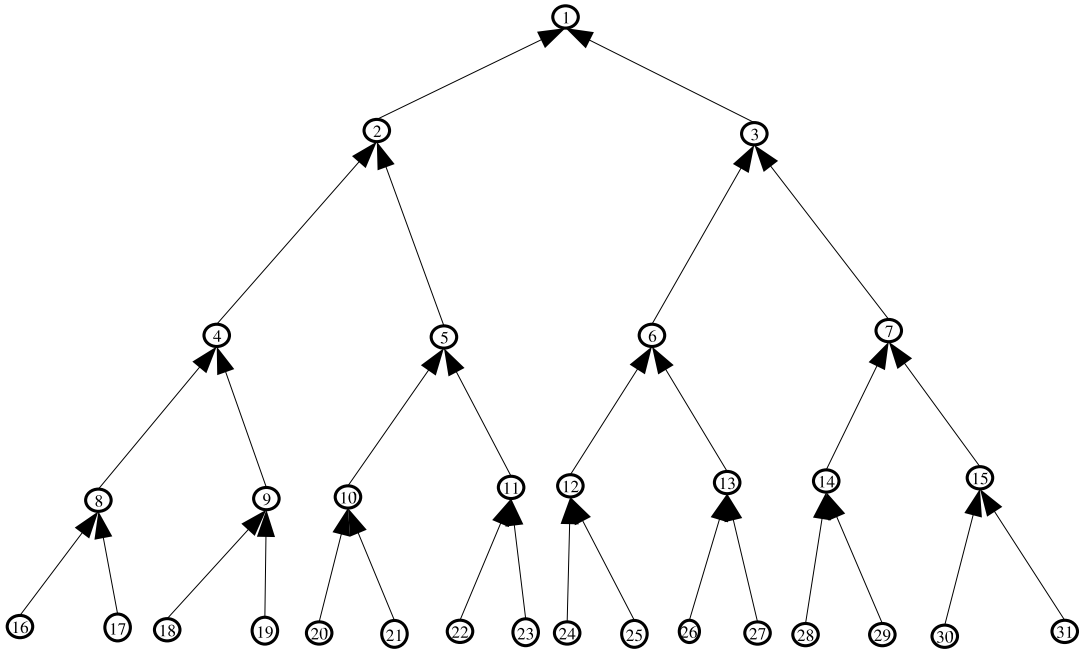

Figure 3.3: The topology where a philosopher crashes while at the top of its partial order

Suppose philosopher 1 fails while eating, and suppose all other philosophers are hungry. Philosophers 2 and 3 request all the forks. They acquire the forks from philosophers 4, 5, 6, and 7, and do not relinquish them (2 and 3 are high priority neighbors to 4, 5, 6, and 7). Philosophers 2 and 3 starve (since they will not be able to acquire the forks from process 1), as will nodes 4, 5, 6, and 7 (since philosophers 2 and 3 will not relinquish their forks). This is also extended to other graph topologies—if a philosopher dies while at the top of the partial order, all its immediate neighbors starve, as do its ring 2 neighbors (after the ring 1 neighbors become hungry and

acquire the forks shared with ring 2 neighbors), as do its ring 3 and 4 neighbors (who starve because the forks are acquired by ring 2 philosophers who do not relinquish the forks to low neighbors). This effect extends from the failed node to the periphery of the graph. As demonstrated, all the philosophers in worst-case scenario starve, and thus have a worst-case failure locality of the diameter of the graph.

3.3 Expected IFL Analysis

In hygienic algorithm, the expected IFL is the same as the worst-case failure locality. Suppose philosophers become hungry infinitely often, meaning that all the neighbors of a failed node holding all its forks will become hungry, request all their forks, and starve. The neighbors of the starving nodes will eventually become hungry. The low-priority neighbors will starve immediately since they will not be able to acquire the forks from their high starving neighbors, and the high-priority nodes will acquire the shared forks, eat, and lower their priority below the starved philosophers. They then will become hungry, and starve, since they are now low-priority neighbors to starving philosophers. Then, the neighbors of the newly starved philosophers will become hungry, and the starvation chain will expand until it fills the entire conflict graph.

CHAPTER 4

Experimental Methods

4.1 Introduction to AnyLogic

For algorithms and topologies as complex as the ones examined here, it is excessively complicated to evaluate expected IFL analytically like we did at the end of chapter 3 for the hygienic solution. A method is needed for evaluating expected (or average) performance quickly under many different varying conditions such as contention upon forks, topologies, failure scenarios (i.e, at what stage during its life-cycle does a philosopher crash), and scheduling of state transitions and events. To that end, a simulation was used to evaluate the performance. For the implementation, we used the AnyLogic 5.0 Java-based simulation toolkit. This toolkit provides a simulation engine to simulate the passage of time and to schedule events, an implementation of a message-passing scheme for implementing philosophers as independent threads, several useful classes for developing the simulations (explained below), as well as an environment for simulation development.

All AnyLogic components inherit from the `ActiveObject` class - an extension of the standard Java `Object` class registered with the AnyLogic simulation engine. All

the components we have implemented, as well as built-in AnyLogic components such as ports, statecharts, and timers are extensions of the `ActiveObject` class. We have used these three classes extensively in implementing the experimental framework, and now discuss their use.

The *`Port`* object is the mechanism that enables message passing between the simulation components. Communicating components are connected via these ports, thus creating channels along which data packets can be sent. Ports are bidirectional—they are used for both the sending and receiving of messages. Messages are defined by the AnyLogic *`message`* class: a data packet passed between `ActiveObject`s. These messages model forks and tokens, as well as meta-information needed by the philosophers and by the other simulation components described below.

The *`statechart`* object is an extensively used mechanism that triggers events in the simulation. Statecharts are implementations of state transition diagrams. They consist of two parts: states and transitions between states. Upon creation, the active state is set to a special initial state. Transitions are triggered by either a boolean formula evaluating to true or by a certain amount of time passing. When a transition triggers, user-specified code on the transition is executed, and a new state becomes the active state in the state chart. Upon entering a state, user-specified code in the state is executed.

Finally, a *`Timer`* is a mechanism that schedules a future event. A timer is initialized with a time value. After the given time has passed, the timer expires, triggering an event to occur.

4.2 Starvation Detection

There were several theoretical issues to deal with in implementing these simulations, the foremost being how to detect starvation. Determining which processes in a system will definitely starve is a computationally unsolvable problem in the general case, where one does not know how the algorithm effects which processes starve. It is impossible to design an algorithm to decide, given a failure in a system, which hungry processes will never be able to transition to eating. To that end, we had to develop a method to intelligently "guess" which philosophers are starved. This "guess" is an estimated time value; if a hungry philosopher fails to acquire its forks within the estimated time period, then it probably will never manage to acquire all its forks, and thus starve. In order to find this time value, the simulation has a training period at the beginning of each run, during which the simulation collects the response times of the philosophers. When the training period expires, the simulation sets a variable called `starvetime` with the estimated time value. The `starvetime` is defined as a constant multiple of the longest response time recorded by the simulation during the training period. After the simulation component responsible for the data collection (the `Collector` object) computes `starvetime`, it informs all philosophers of this variable. After a philosopher in the system fails, all philosophers start timing the length of time it takes to eat once they become hungry. If that time exceeds the `starvetime`, a philosopher postulates it is starving, and sets its state to "starved." In essence, the simulation tries to "guess" which philosophers are starving, by examining which philosophers are taking an unexpectedly long time to transition into the eating state. This method, however, can generate false positives (i.e, it can identify some philosophers as starving when they may just be taking an exceptionally long time to

collect forks and eat). Therefore, we allow potentially starving philosophers to transition into eating if they manage to acquire all their forks. In that case, the philosopher sets its state to "eating", and the simulation increases the `starvetime`. However, this does not happen often. We have only observed this once under a single algorithm in a very high contention scenario, leading us to conclude that false positives happen quite infrequently. This method also allows for false negatives. To combat this effect, when a philosopher crashes, the simulation calculates a time period which is a multiple of the `starvetime`. When this time period expires, the simulation ends. During this period, philosophers will have the opportunity to become hungry and eat several times, since it is a multiple of a multiple of the longest response time. The danger for false negatives occurs in algorithms where response time dramatically increases as a result of a philosopher crashing. None of the algorithms we have studied has this problem.

4.3 The Implementation

We used the AnyLogic 5.0 Java-based toolkit to implement these simulations. We divided the simulations into two packages: one package for the algorithm-independent components, and one for the algorithm-specific implementation. A philosopher is implemented in two components: a client and a proxy. The *Client* contains all the aspects common to philosophers under all algorithms. The *Proxy* contains the algorithm-specific details. The three other components in the simulation are the *Assassin*, a component that injects a failure into the system (i.e, "tells" a philosopher to fail); a *networkDelay* object that introduces a delay on the channels between philosophers through which they exchange forks and tokens; and a *Collector* object

that collects statistics such as response time, message complexity, and failure locality statistics.

4.3.1 The General Architecture

All five components above come together to create the entire simulation. The architecture is represented in Figure 4.1. The large gray boxes represent whole simulation components, and the small boxes on the borders of the larger boxes represent ports, and the lines between ports represent connections. So, for example, all clients are connected via a port to the collector—which therefore means that they communicate via this connection.

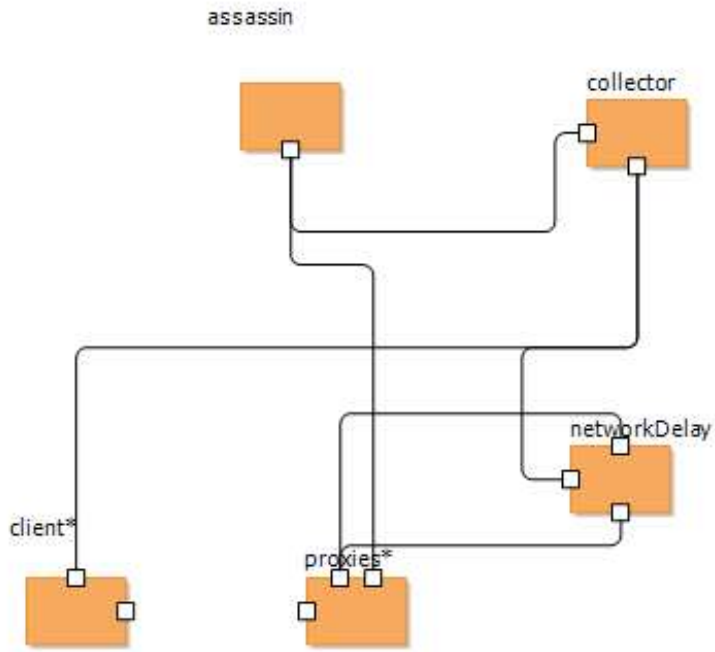

Figure 4.1: The simulation architecture

The `client*` and `proxies*` above represent an array of clients and proxies whose length is the cardinality of the vertex set of the conflict graph. In Figure 4.1, there is no connection between the ports of the clients and the proxies, because they must be programmatically defined at runtime, in order to ensure connectivity in a 1-to-1 fashion (see Figure 4.2). Furthermore, the cardinality is determined dynamically, at run time, so the size of the arrays containing the client and proxy components cannot be determined a priori.

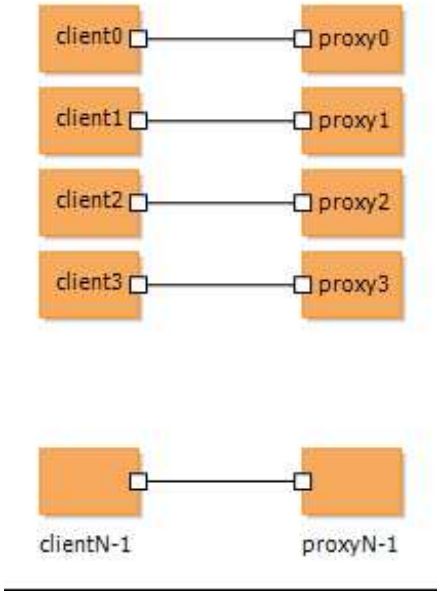

Figure 4.2: The 1-to-1 mapping of proxies to clients

4.3.2 The `Proxy` Component

The `proxy` component contains all the algorithm-specific implementation details, although there are similarities between all the implementations of the proxy object.

All proxies have three ports: the *clientPort* port, to connect them to a corresponding client to pass messages containing state information; the *mainPort* port, to connect proxies to each other (via the network delay object) for passing forks and tokens; and the *Death* port, to connect the philosophers to the `Assassin` object, through which a philosopher receives instruction to fail.

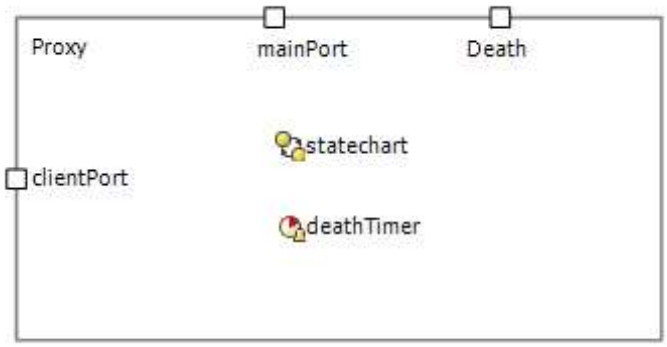

Figure 4.3: The proxy component

The `mainPort` is the component in which most of the algorithm is implemented. Most of the work of an algorithm is implemented in the "on receive" code of the `mainPort`, because these algorithms are event-driven (where the arrival of a request or fork token is an event). The `mainPort` of every proxy is connected to the `networkDelay` object. When a proxy has to send a fork or token to another proxy, it sends a message containing the destination address to the `networkDelay` object. The `networkDelay` object holds the message for a period of time, then broadcasts the message to all the proxies. Only the proxy to whom the message is addressed will

act upon the message. The message that it sends contains the following fields (this is constant across all algorithms, although not all the algorithms use all the fields):

```
String type          // Either "fork" or "token"
int source_id        // the source address
int destination_id   // the destination address
boolean clean        // a boolean which marks the message
                     // as either clean or dirty if the
                     // message is a fork.  This is only
                     // used in hygienic, thresholds,
                     // biserial, and strict biserial
                     // algorithms
double time          // represents a time value used to seed
                     // a timer. This is used when calculating
                     // starvetime
int priority         // used for marking dynamic priority
                     // in the double-doorway and bounded
                     // doorway algorithms
```

The other important port is the `clientPort`. All communications between a client and its proxy passes through the `clientPort`. All the important events which do not occur at the reception of a message in the `mainPort` occur in the `clientPort`. For example, when a philosopher becomes hungry (a state transition which is implemented in the client object), it sends a message to the proxy via the `clientPort`. When a proxy receives such a message from the client, it can request forks.

Some proxies also contain statecharts, which are used for algorithm-specific bookkeeping - for example, in the implementation of the thresholds algorithm, the statechart marks the transition between the so called "threshold point" and the non-threshold point (this is further elucidated in Chapter 5).

4.3.3 The `Client` Component

The other half of a philosopher is called the client. As previously stated, the client contains all the implementation-independent code of a philosopher.

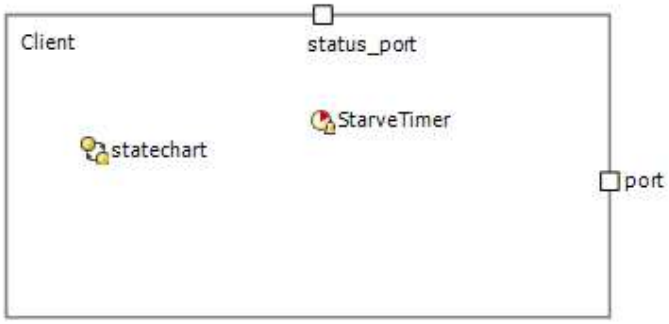

Figure 4.4: The client component

The most important component of a client is the statechart, which controls the transitions between the three states of a philosopher's life cycle. It also has two extra states: *`DEAD`* and *`END`*. If a proxy receives an instruction to fail from the assassin, the proxy informs the client, and the client transitions into the `DEAD` state. If a proxy receives an instruction from the collector object that the simulation is over, it transitions into the `END` state. The reason for this is that when a philosopher is in the `END` state, it will no longer send any forks or tokens—which means that after a few

iterations, there are no more events scheduled, which causes the AnyLogic simulation engine to terminate the simulation. A description of the statechart is represented in Figure 4.5:

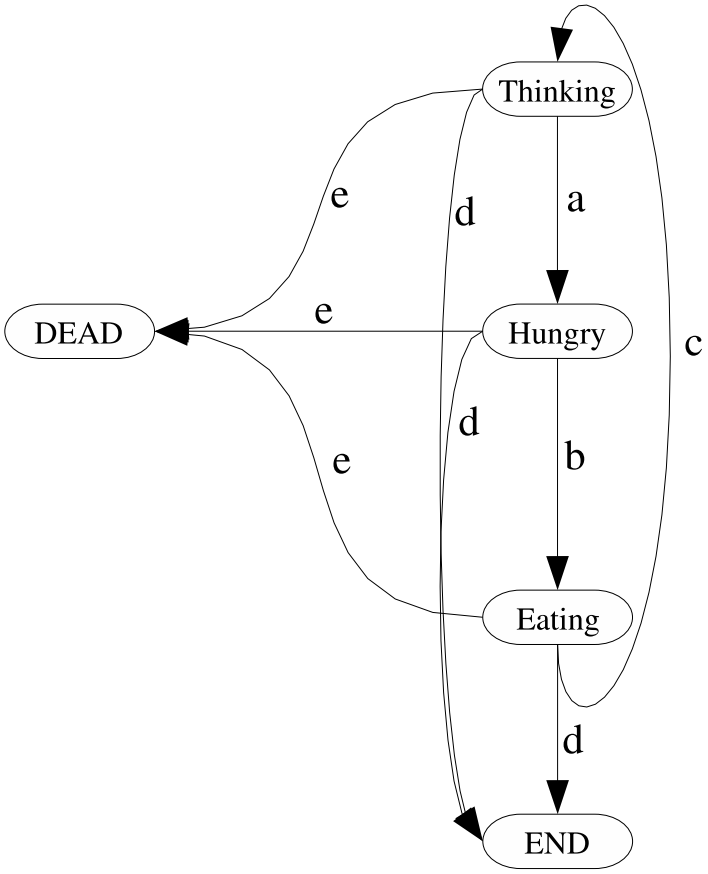

Figure 4.5: A representation of a `client`'s statechart

Table 4.1: State transitions and their actions

Transition	**Trigger**	**Action on transition into new state**
a	A finite period of time, determined by the output of a negative exponential distribution.	Send a message to the proxy informing it that the philosopher is hungry. If the training period is in effect, record of the current simulation time. If the training period is over, start the `StarveTimer` (which is seeded with the `starvetime`).
b	The reception of a message from the proxy, informing the client that it has collected all needed forks and may therefore eat.	If the training period is in effect, record of the simulation time, and take the difference between this time and the previous value on transition into the hungry state. This is the response time, which the client sends to the Collector object. If the training period is over, reset the `StarveTimer`
c	A finite period of time, determined by the output of a negative exponential distribution.	Send a message to the proxy informing it that the philosopher is now thinking
d	The reception of a message from the collector object informing the client that the simulation is over	None

Transition	Trigger	Action on transition into new state
e	Reception of a message from the proxy that the philosopher has been marked to die. The proxy sends this message when two conditions occur. The first is that it receives a message from the assassin that the philosopher telling the philosopher to crash and the second is when a condition set in a parameter has been met. This condition is a failure scenario - that is, the condition specifies a state which the philosopher should be in when it dies. For example, this condition can be "eating" (specifying that the philosopher should die while eating), or can be "no_forks" (specifying that the philosopher should die while not holding on to any forks).	None

Finally, the `Client` contains a timer, whose expiration implies that the philosopher is starving. Upon expiry, the `StarveTimer` sets a variable marking the client as "starving."

4.3.4 The `Assassin` Component

The `Assassin` component's purpose is fairly straightforward. At the beginning of the simulation, it starts a timer seeded with a value specified by the user. If the user specified a certain philosopher to fail (i.e, choose a specific philosopher to crash) when configuring the simulation run, the assassin sends a message to that philosopher instructing it to die when the timer runs out. If a philosopher is not specified to die, then the assassin randomly chooses a philosopher and sends it instruction to die when a certain condition is met. This condition is a function of the philosopher's state. For

example, if the condition is set as "hungry", the philosopher will crash when it next becomes hungry. There is one failure condition that requires slightly different behavior in the `Assassin`. `hungry_time` is a failure condition specifying that a philosopher should crash at some point while hungry; not necessarily when it first becomes hungry. When the failure parameter is `hungry_time`, the assassin first sends a message to the `Collector` (see 4.3.6 on page 36). The `Collector` then calculates the average response time in the system, and computes a new random sample from the negative exponential distribution with the average as the sample, and sends that number to the `Assassin`, which then sends the "die" message to the proxies. The proxy then receives this message, and when the condition in the instruction is satisfied, it sets a boolean marking it as dead. If that condition is a state, such as "thinking", "hungry", or "eating", the proxy will wait until it transitions into the state, after which it ceases all sending of forks and tokens, and sends a message to its client informing of its death. If the condition is "hungry_time", the proxy does the following: as soon as the philosopher becomes hungry, it starts a timer seeded with the time value in the "die" message. When the timer expires, it checks to see that it is still hungry. If it is, then it dies. If not, it sends a message to the collector to abort the simulation run.

4.3.5 The `networkDelay` Component

The `networkDelay` component is also a fairly simple component. All proxies are connected to the `networkDelay` object, meaning that all exchanges of forks and tokens between philosophers pass through this component. This component adds a time delay to all messages exchanged between philosophers, by creating a timer each time it receives a message. After the timer associated with a message expires,

it broadcasts that message to all philosophers. Only the philosopher to whom the message is addressed will then act upon it.

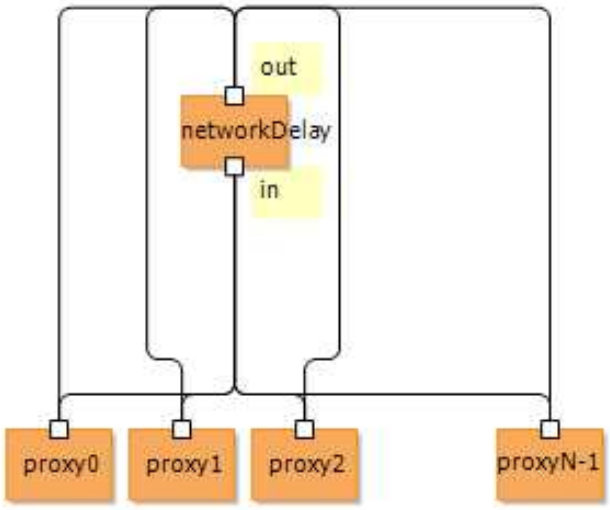

Figure 4.6: The channel connections between the `proxies` through the `networkDelay` object.

4.3.6 The `Collector` Component

The `Collector` component serves many purposes. Its primary purpose is to collect information from the other components in the simulation for use in evaluating performance. It contains both a representation of the graph topology in the form of an adjacency matrix, and a representation of the relative distances of the philosophers from one another in a matrix called the *neighborhood map*. The neighborhood map has the same dimensions as the adjacency matrix. Each entry in the map is an integer representing the shortest distance between the philosopher represented by the row and the philosopher represented by the column of the entry. Throughout the

simulation, philosophers send data about response time and message complexity to the collector. When the simulation is scheduled to terminate, the collector queries all the philosophers to discern who is starving and who is not - and then calculates the integrated failure locality metric. The algorithm for calculating integrated failure locality is fairly simple. First, the collector isolates the line in the neighborhood map corresponding to the dead philosopher. It then finds the distance of the farthest node in the graph, and creates an array of that size to store the number of philosophers within successive rings to the failed node. It then builds a similar array to store how many starved processes exist within these successive rings, and finally sums up the ratios of starved to total processes within the rings.

4.3.7 Simulation Initialization

The configuration of the experiment is parameterized in the command-line arguments specified when executing a simulation run. The parameters are described below:

Table 4.3: Input parameters to the simulation

type	**name**	**function**
`int`	`N`	Represents the cardinality of the vertex set of the conflict graph (i.e, the number of philosophers in the system).
`String`	`Filename`	The name of the file containing the representation of the graph topology in the form of a boolean adjacency matrix.
`int`	`failure_node`	The ID of the node to fail. If it is set to 0, then the simulation will fail a random process.

Continued on next page

type	name	function
String	failure_pattern	A string representing the condition in which a philosopher should fail. This can take the values "thinking" "hungry" "eating" "no_forks" and "half_forks." A philosopher will then fail when the condition is met.
double	failure_timeout	A time value since the beginning of the simulation after which the assassin must inject a failure into the system.
double	collector_timeout	A time value since the death of a philosopher after which the simulation ends. If "mode" is set to "training", this value will be overridden with the value obtained during the training period.
double	think_time	the average thinking time.
double	eat_time	the average eating time.
double	starve_time	Set the time for a philosopher to identify itself as starving after. If "mode" is set to "training," this value is overridden with the value obtained during the training period.
String	mode	If set to "training," the time before a failure is injected into the system will be designated as a training period in order to set the starvation time.
double	network_delay	the average delay on channels.
String	output_filename	The name of the file to which the simulation outputs its results.

The simulation is then initialized with these values. Most of the parameter values are simply copied over to the components. Information in files (for example, the filename containing the adjacency matrix representation of the topology) is then processed and loaded into the proper data structures. Finally, the simulation is seeded with the initial configurations specified by the algorithms (for example, initial distribution of forks).

CHAPTER 5

The Thresholds Algorithm

5.1 Algorithm Description

The thresholds algorithm [6] is based on Chandy and Misra's hygienic algorithm [1]. Like the hygienic algorithm, the forks have a flag associated with them marking them as clean or dirty, thus marking a dynamic priority on edges between philosophers. In the thresholds algorithm, a philosopher's set of neighbors is partitioned into two sets—high-priority neighbors (called the threshold set), and low-priority neighbors. Note that the membership of these two sets is dynamic, since the relative priorities among a set of philosophers change after a philosopher eats. When a philosopher becomes hungry, it requests forks only from all processes it believes to be high priority neighbors (we call these forks *high forks*). Once it acquires all its high forks, it then requests the rest of its forks from low priority neighbors (we call these forks *low forks*). Once it acquires all its forks, a philosopher eats. To take advantage of the partitioning of neighbor sets, the hungry state has been divided into two states: hungry and not at threshold point and hungry and at threshold point. A philosopher that holds all its high forks is said to be at its threshold point, while a philosopher that does not hold all forks shared with high priority neighbors is not. If a philosopher

is not at its threshold point and receives a request for a clean fork (i.e, a low fork), it immediately relinquishes the fork. If, in the same conditions, it receives a request for a dirty fork (i.e, a high fork), it relinquishes the fork and immediately re-requests it. If a philosopher is at its threshold point and receives a request for a clean fork, it defers the request; and if it receives a request for a dirty fork, then it relinquishes the fork, and re-requests it. This means that the philosopher will no longer be at its threshold point, so it will also relinquish all deferred requests from low neighbors. (See Figure 5.1 for a graphical explanation.)

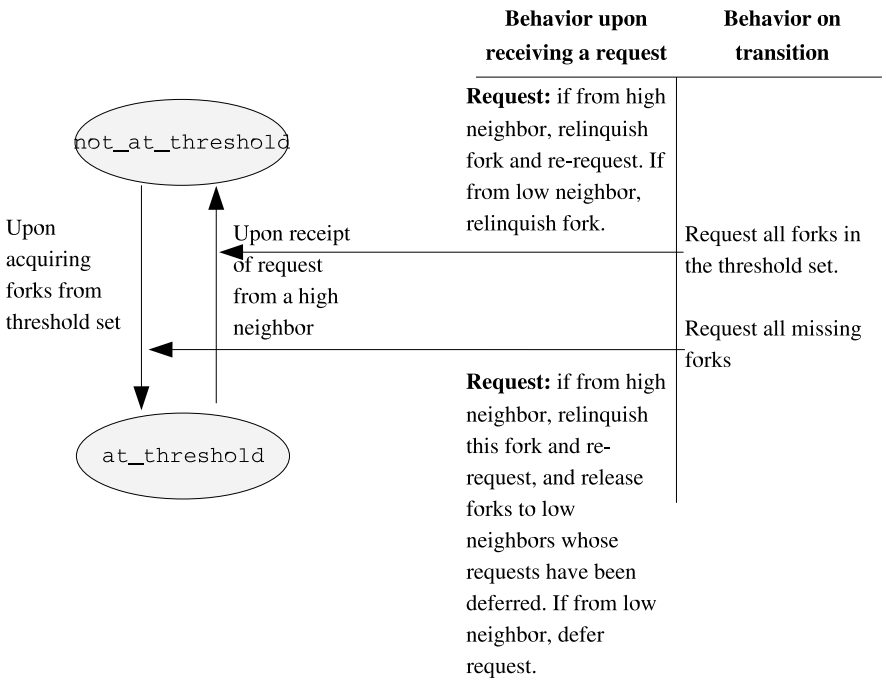

Figure 5.1: Threshold algorithm behavior on transitions and receipt of requests

The formal algorithm follows:

Program p

initially $(\forall p :: p.state = thinking)$

$(\forall p, q :: clean(p, q) = false)$

$(\forall p, q :: p < q : fork(p, q), req(p, q) = p, q)$

Priorities form a partial order

always $p.tp \equiv (\forall q : p < q : fork(p, q) = p)$

$p.t \equiv p.state = thinking$

$p.h \equiv p.state = hungry$

$p.e \equiv p.state = eating$

assign

$H_p : \{p.h \wedge \neg p.tp\}$

$(\forall q : N(p, q) \wedge fork(p, q) = q \wedge clean(p, q) : req(p, q) := q;)$

$P_p : \{req(p, q) = p \wedge fork(p, q) = p \wedge \neg clean(p, q)\}$

$fork(p, q) := q;$

$clean(p, q) :=$ **true**;

$req(p, q) := q;$

E_p $\{p.h \wedge (\forall q : N(p, q) : fork(p, q) = p \wedge (clean(p, q) \vee req(p, q) = q))\}$

$p.state := eating;$

$(\forall q : N(p, q) : clean(p, q) :=$ **false**);

R_p $\{req(p, q) = p \wedge fork(p, q) = p \wedge \neg p.tp\}$

$fork(p, q) := q;$

$clean(p, q) := \neg clean(p, q);$

5.2 Worst-case Analysis

In the worst case, the thresholds algorithm has failure locality 2. When a philosopher crashes while eating, all its immediate neighbors starve, and all the neighbors of its high neighbors starve. The chain of events is as follows. A philosopher crashes (node A in Figure 5.2). When its low priority neighbors (e.g. node B in Figure 5.2) become hungry, the low neighbors request their high forks, which they will never receive. Consequently, B will never reach its threshold point because it cannot acquire its high fork from A. Since B never reaches its threshold point, if any of B's neighbors become hungry and request forks, they will receive them. When B's high neighbors (other than the crashed philosopher) finish eating, they will return the fork to B. However, if and when they become hungry again, they will be low neighbors—which means that when they request forks again, they'll be able to keep the forks indefinitely.

When A's high priority neighbors (e.g, node C in Figure 5.2) become hungry, they first request their high forks. Once they acquire them, they request their low forks. Because C can reach its threshold point, it will only relinquish forks to high neighbors. Assuming that philosophers become hungry infinitely often, all of C's neighbors will eventually be low neighbors, at which point C will be at its threshold, request all its low forks, and never relinquish any—which means that all its neighbors (e.g. node D) will starve. D's immediate neighbors will not starve, however. Because eventually all of C's neighbors become low neighbors, none of them will be at their threshold points, which means that they will all relinquish forks until they receive a fork from C. Therefore, the farthest starving node will be distance 2 from the crashed philosopher.

In the worst-case, A only has high neighbors, so everything within radius 2 of A will starve.

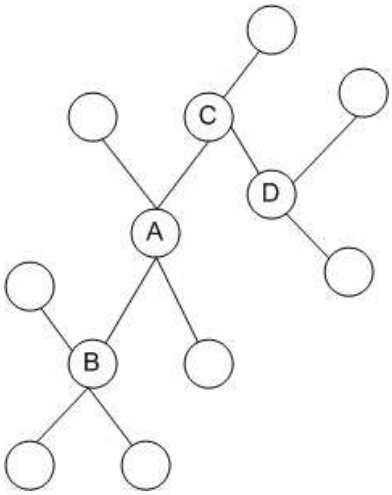

Figure 5.2: Worst case analysis for the Dynamic Thresholds Algorithm

5.3 Expected IFL Data

We ran the simulation, evaluating performance for failures under three scenarios – failure while eating, failure upon becoming hungry, and failure after an arbitrary time period while hungry, across three different topologies – a randomly generated topology (to get an idea of how topological "noise", that is, random connectivity impacts the IFL), a simple k-tree (called a *starburst*) where each node had four neighbors (save for the leaves, which had one), and a modified k-tree where some of the leaves were interconnected. The reason for using the modified starburst topology is to make the response times more uniform among the nodes, since variations in the cardinality of

each process' neighbor set result in variations in the relative response times between neighboring philosophers.

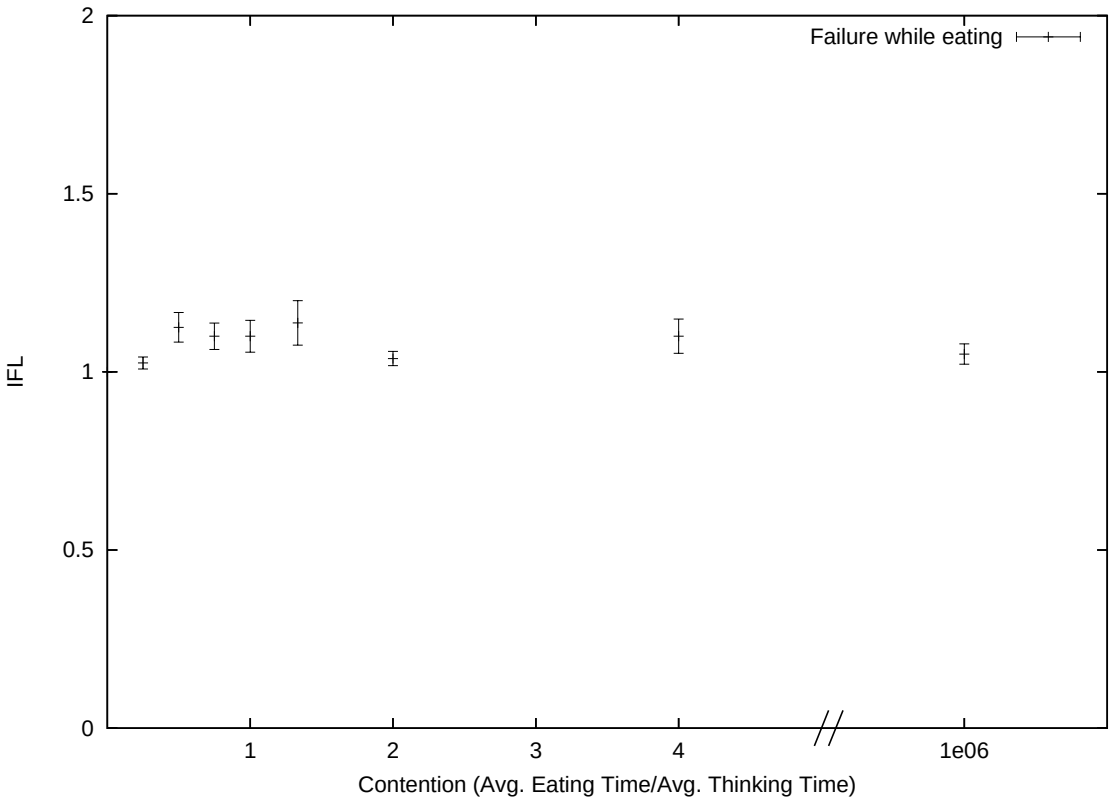

Figure 5.3: IFL for the Thresholds algorithm on a starburst topology, where a philosopher crashed while eating

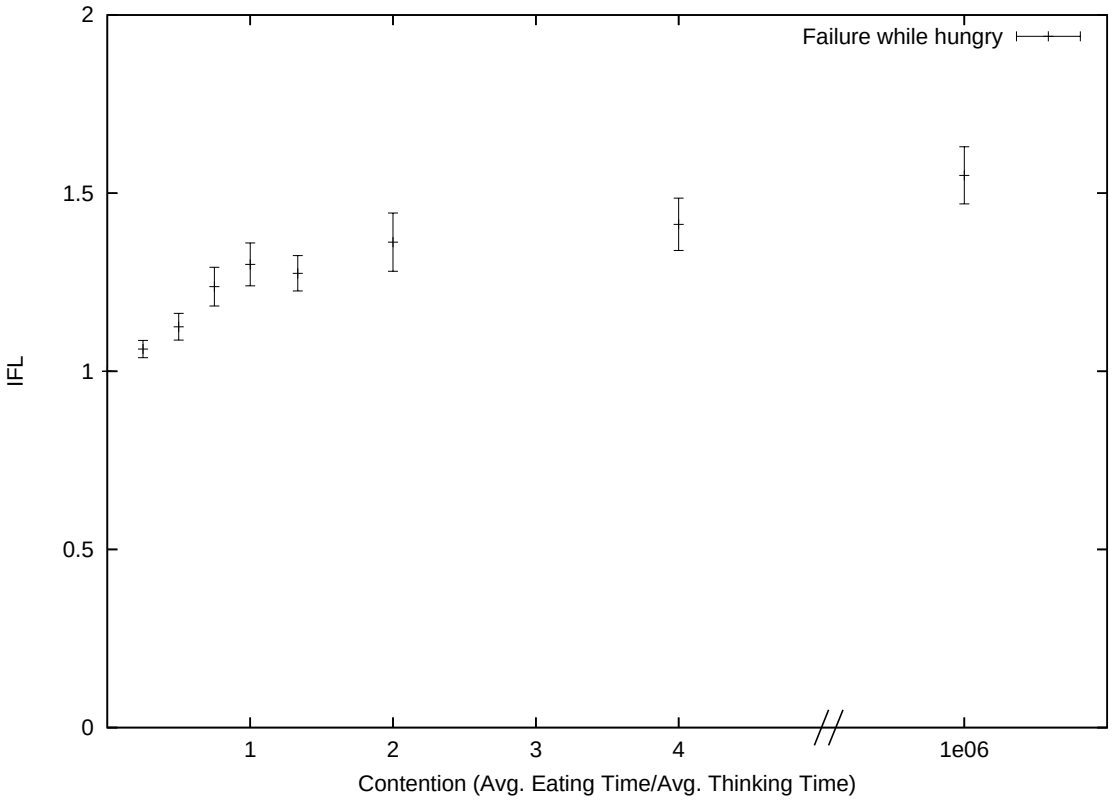

Figure 5.4: IFL for the Thresholds algorithm on a starburst topology, where a philosopher crashed upon becoming hungry

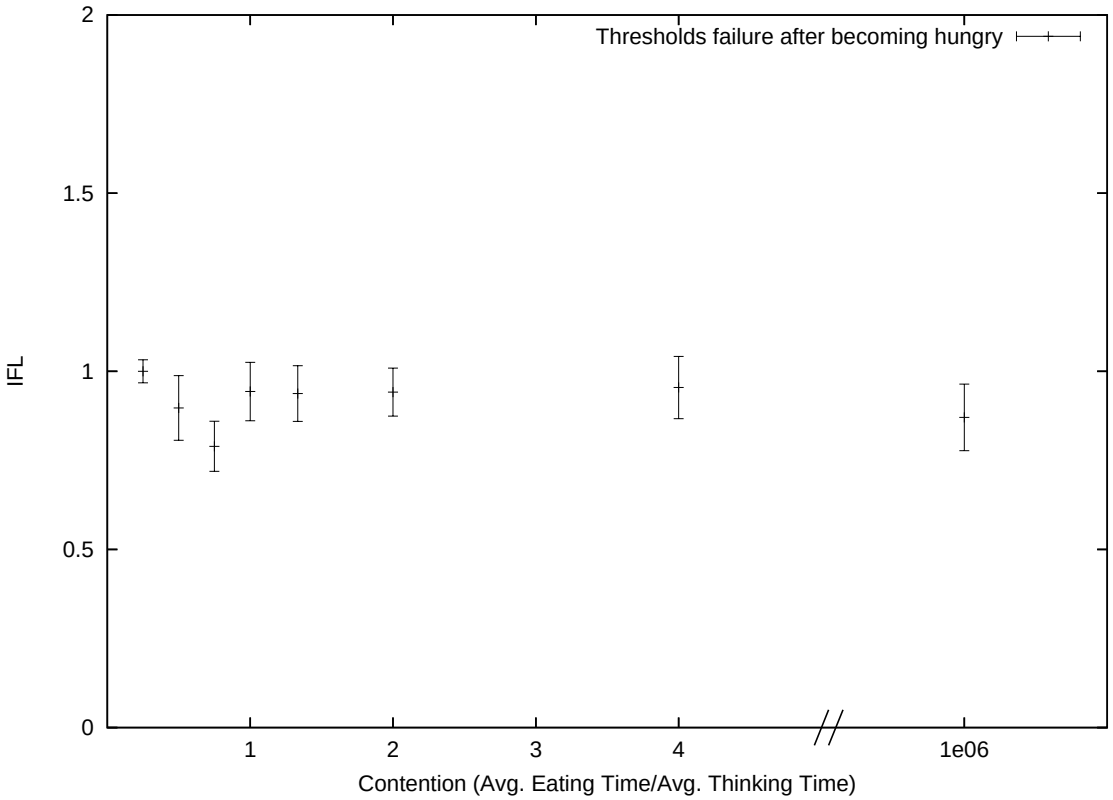

Figure 5.5: IFL for the Thresholds algorithm on a starburst topology, where a philosopher crashed some time after becoming hungry, before eating

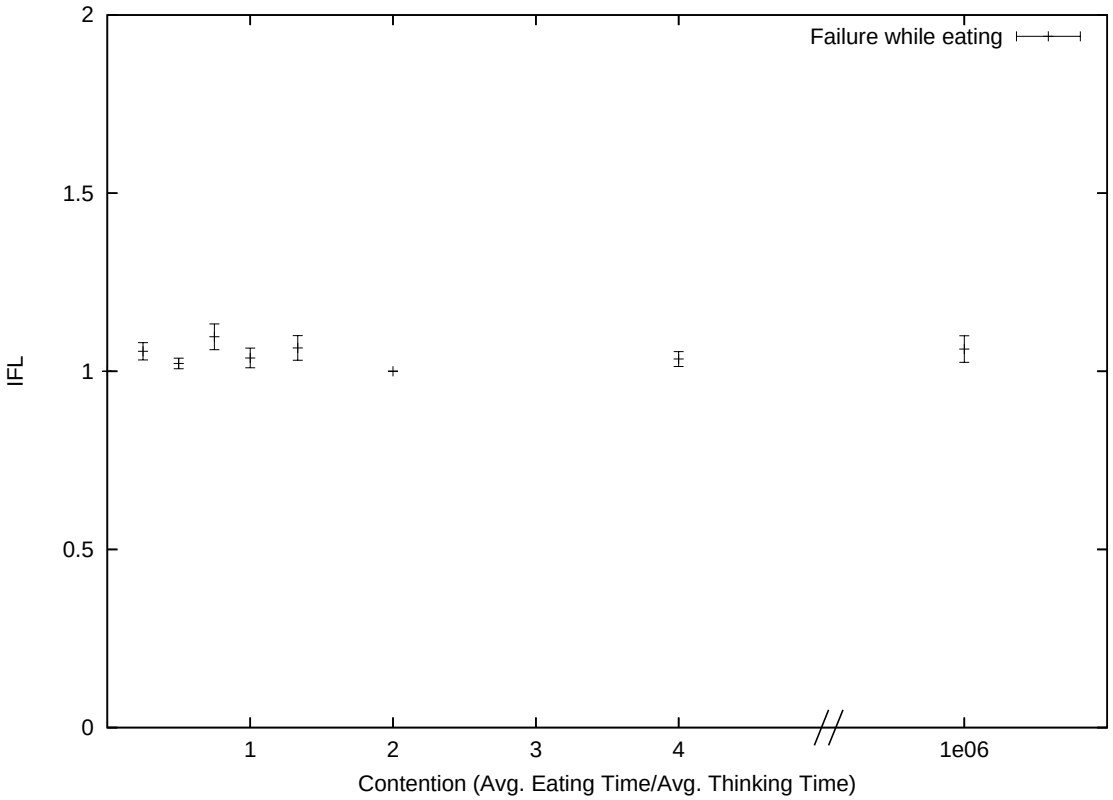

Figure 5.6: IFL for the Thresholds algorithm on a random topology, where a philosopher crashed while eating

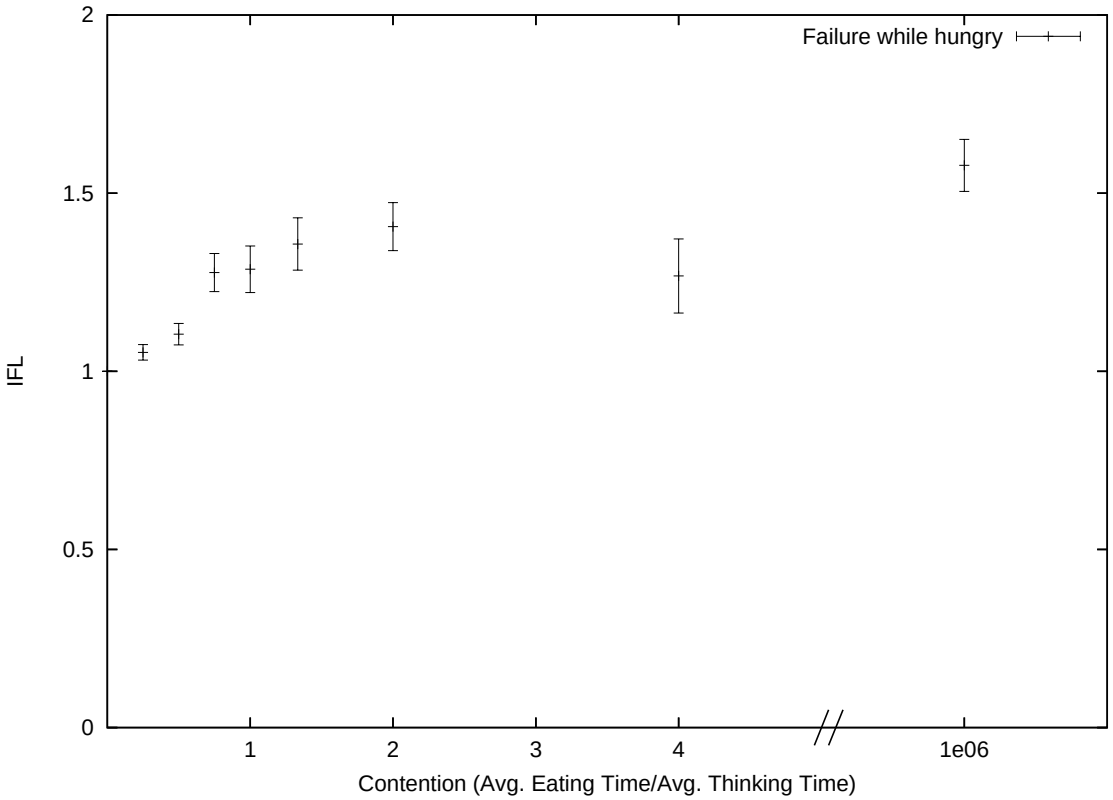

Figure 5.7: IFL for the Thresholds algorithm on a random topology, where a philosopher crashed upon becoming hungry

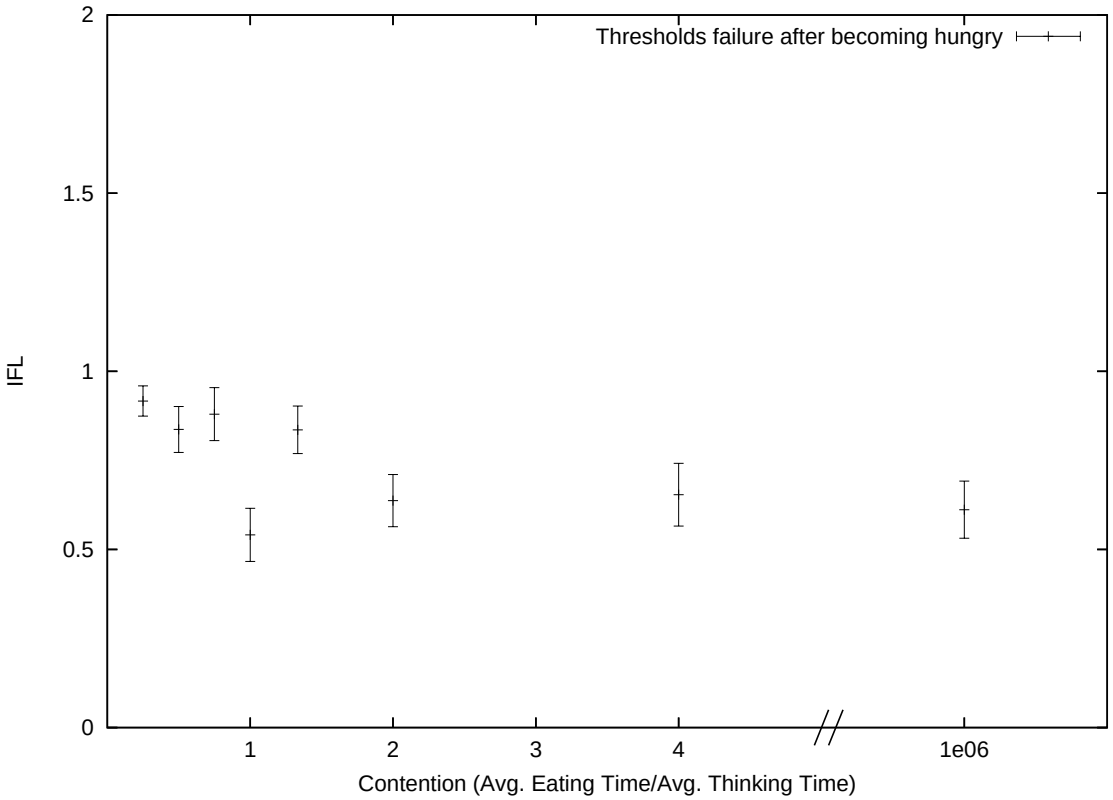

Figure 5.8: IFL for the Thresholds algorithm on a random topology, where a philosopher crashed some time after becoming hungry, before eating

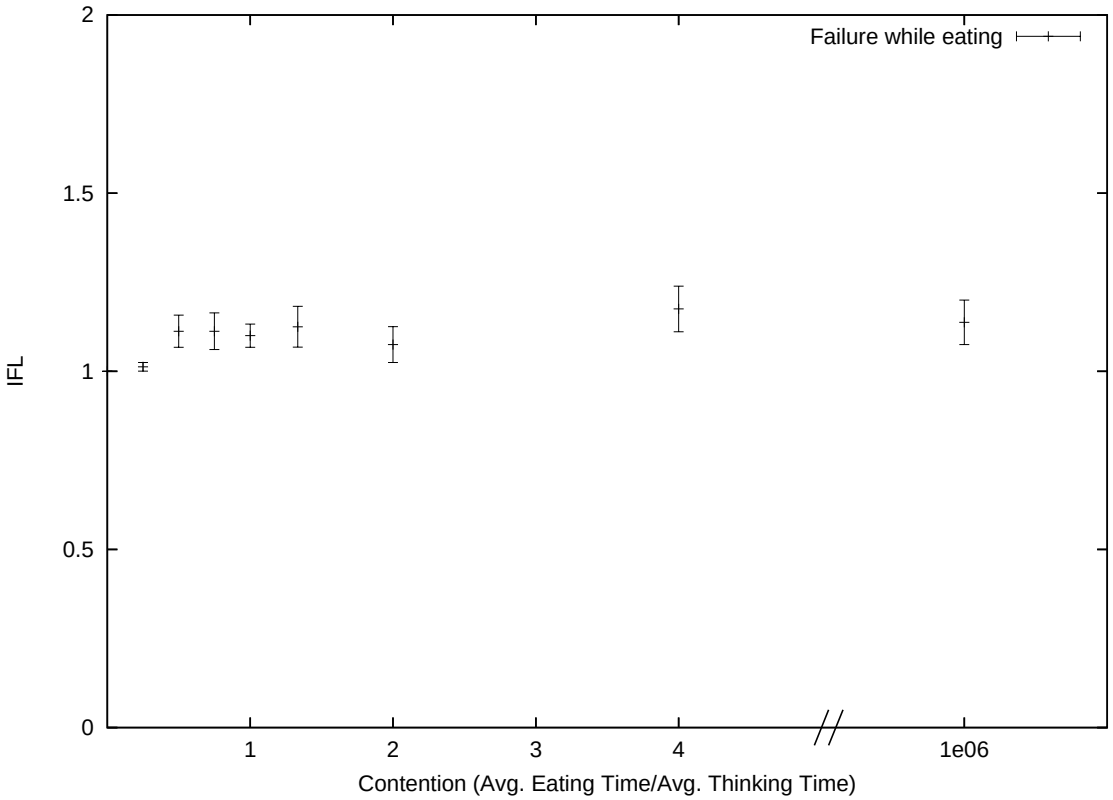

Figure 5.9: IFL for the Thresholds algorithm on a modified starburst topology, where a philosopher crashed while eating

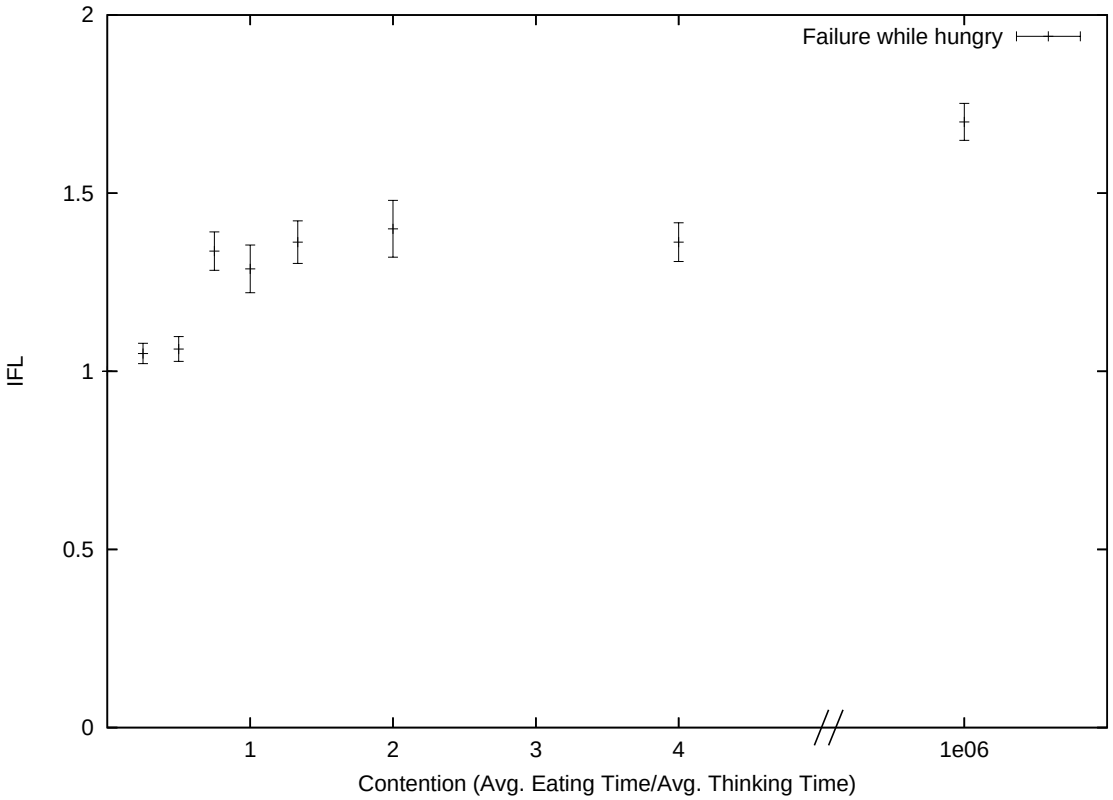

Figure 5.10: IFL for the Thresholds algorithm on a modified starburst topology, where a philosopher crashed upon becoming hungry

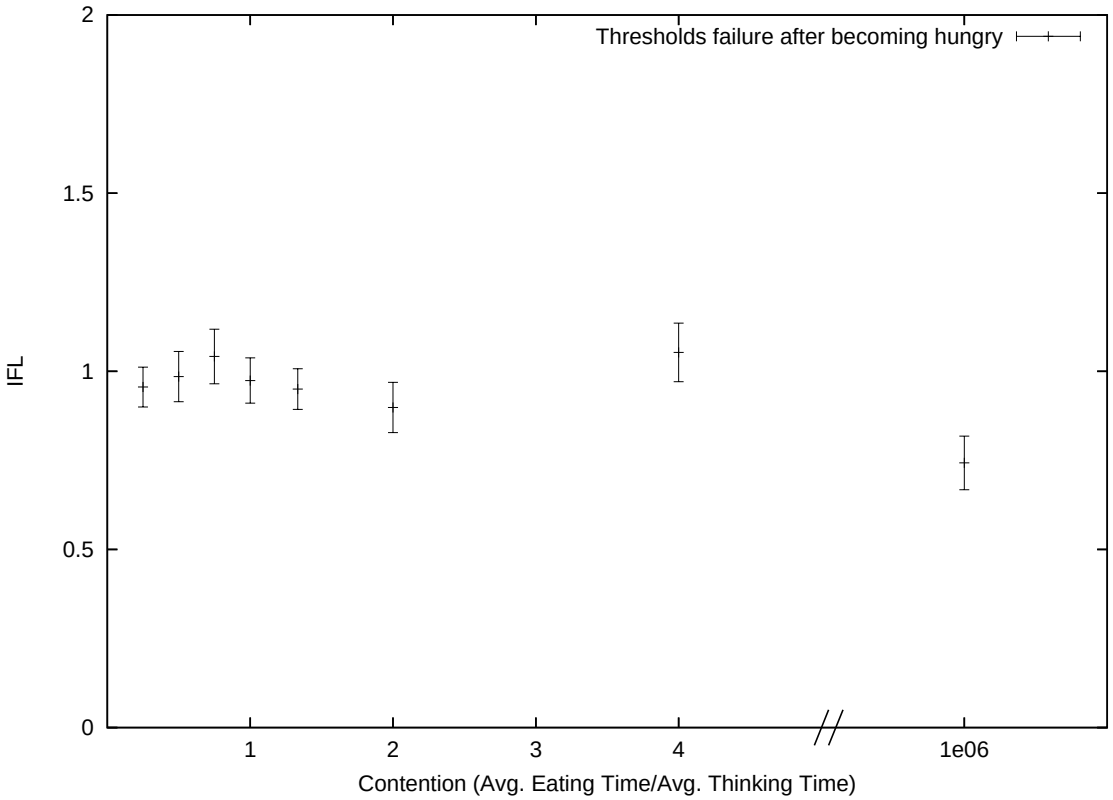

Figure 5.11: IFL for the Thresholds algorithm on a modified starburst topology, where a philosopher crashed some time after becoming hungry, before eating

As one can see, the actual performance of these algorithms is much better than the worst-case failure locality of 2.

5.4 Analysis

In our initial expected failure locality analysis, we hypothesized that we would get an average IFL of 1.5. We figured, based on the worst-case analysis, that on average, when a philosopher crashed, half its neighbors would be high and half would be low. Our initial analysis, however, was oversimplified. There are several phenomena at play which lead to lower IFL values.

5.4.1 The Stale Knowledge Effect

The first oversimplification we made in our initial analysis was that philosophers always know their place and their neighbors' places in the priority partial order. This is demonstrably false. A philosopher receives information about a neighbor's priority only upon receiving a fork from the neighbor. Therefore, if a philosopher eats and lowers its priority, none of its neighbors will know that it has lowered its priority until they become hungry and request (and subsequently receive) a fork from the philosopher. This is the most dominant effect in explaining the lower-than-expected average IFL. In the very low contention scenario under failure while holding all forks, where we observe an average IFL of 1.0, the following happens.

A philosopher crashes while holding on to all its forks. If all its neighbors have eaten before it has since it has last eaten (i.e, of all its neighbors, it is the last one to have become hungry and to have eaten), then it will only have low neighbors, giving an IFL of 1.0. If it has some high neighbors and some low neighbors (that is, it has eaten at least once before its high neighbors have eaten, and its low neighbors ate before it has), the following takes place: the philosopher's high neighbors believe themselves to have last eaten with respect to their neighbors (i.e, a philosopher in the high neighbor set believes itself to have been the last one to eat among its neighbors), since they would not have received any forks from the crashed philosopher. Therefore, they see the crashed philosopher as a high neighbor, (even though the crashed philosopher is actually a low neighbor). Thus, the high neighbors of the crashed philosophers believe they are waiting upon a fork from a high neighbor, and therefore will let their neighbors acquire their shared forks. This effect is most dramatic in very low

contention scenarios, but still heavily impacts the performance under all contention scenarios.

The other manifestation of the stale knowledge effect is in scenarios in which philosophers fail while thinking or immediately after a philosopher becomes hungry. In that case, the stale knowledge effect actually makes the performance of the algorithm worse. When a philosopher becomes hungry, it perceives all its neighbors to be high neighbors, since, when it last ate, it marked all its forks as dirty, and while it was thinking, it never received any forks. So according to its local state information, it is at the bottom of the partial order. Therefore, when a philosopher crashes immediately after having become hungry and after it had sent its fork requests, all of its immediate neighbors will eventually starve, since the low neighbors who are not eating will automatically relinquish the forks (the eating ones will relinquish once they have finished eating), and the high neighbors will immediately relinquish their forks if they are not at their threshold points. If they are, then once they eat (or once they are preempted by a high neighbor), they will relinquish their forks.

The stale knowledge effect is mitigated to an extent when a philosopher fails some time interval after becoming hungry, especially in high contention scenarios. As stated earlier, information about a process' position in the partial order is only communicated by the receipt of a fork. When a philosopher crashes some time after it has become hungry (but before it eats), there is a slightly greater probability that more forks will have been exchanged (especially in high-contention scenarios), which means that it is more likely that the crashed philosopher has accurate knowledge of its position in the partial order. Therefore, failure after the receipt of a fork followed by

relinquishing it leads to lower IFL, since it will not hold all its forks or have all outstanding request when it fails.

5.4.2 Relative Response Times

The relative response times of pairs of philosophers also impacts average IFL. The lower the average response time of a philosopher compared to the average response time of its neighbors, the more often it will have a chance to become hungry, since on average it will spend less time hungry and will be able to acquire forks more readily than its neighbors. This, therefore, means that a philosopher with lower average response times is more likely to be a low neighbor to a philosopher with higher average response times. Thus, in higher-contention scenarios, it cannot be expected that when a philosopher crashes, on average half its neighbors will be high and half will be low. It is more likely that a crashed philosopher will have more low-priority neighbors than high-priority neighbors, thus leading to a lower-than-expected IFL. This effect is most apparent in the high-contention scenarios, where there is high variance in response times. The connectivity of the underlying conflict graph also impacts the IFL, since it affects relative response times. Suppose there are two neighboring philosophers, u and v, where u's only neighbor is v, and v has six neighbors. On average, v's response time will be much higher than u's, since v has to collect six forks, where as u has to collect just one.

5.4.3 Connectivity Among 1-ring Neighbors

The topology of the subgraph of high neighbors to a crashed node also impacts failure locality. Certain scheduling scenarios lead to some philosophers protecting other philosophers within the same ring from starvation. For example, in the graph

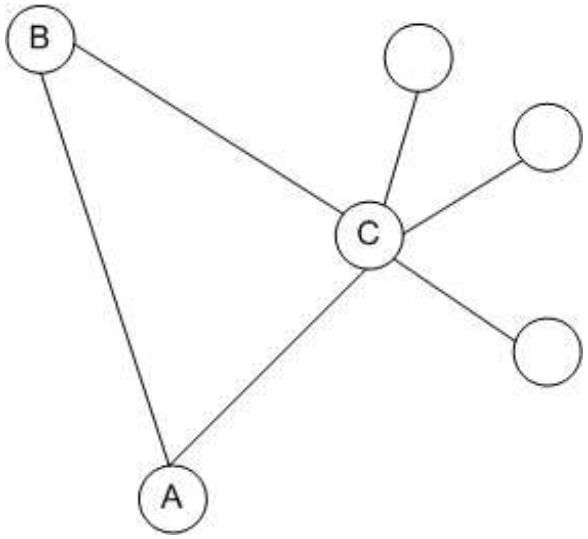

Figure 5.12: Connectivity analysis

in 5.12, philosopher A crashes. Philosopher B then becomes hungry. It reaches its threshold point, and requests its low forks, and acquires a fork from C. Philosopher C starves; it never reaches its threshold point, since B will not relinquish the shared fork. Therefore, Philosopher C protects all its immediate neighbors (except for B) from starvation - that is, it isolates part of the 2-ring from starvation. Even if C becomes hungry before B, the same situation will occur: B will reach its threshold point, C will relinquish its fork to B, and none of C's neighbors (other than B) will starve. This effect helps lower the IFL under all failure conditions.

CHAPTER 6

The Biserial and Strict Biserial Algorithms

The biserial and strict biserial algorithms are variants of the dynamic thresholds algorithm[6]. In the dynamic thresholds algorithm, when a philosopher is preempted by a higher-priority neighbor's request for a fork while hungry, it immediately re-requested the fork after relinquishing it. In the biserial algorithm, this does not happen. Instead, a philosopher sends out its fork requests in rounds. When a philosopher first becomes hungry, it will send out requests for its missing high forks. If it receives any requests for high forks while it is waiting upon requested forks, it does not relinquish and immediately re-request; it relinquishes the fork, and then waits until it has received the current batch of outstanding requested forks before sending out a request for the fork. When it has attained its threshold point, it will form a batch of low requests and request all missing low forks. If, while at its threshold point, a philosopher gets preempted by a high neighbor, it will relinquish the requested fork, but not immediately re-request it; it will wait for its outstanding requests to be fulfilled, after which request all missing high forks. A formal description of the algorithm follows:

Program p

var S : A set to add outstanding requests to

initially $(\forall p :: p.state = thinking)$

$(\forall p, q :: clean(p, q) = false)$

$(\forall p, q :: p < q : fork(p, q), req(p, q) = p, q)$

Priorities form a partial order

always $p.tp \equiv (\forall q : p < q : fork(p, q) = p)$

$p.t \equiv p.state = thinking$

$p.h \equiv p.state = hungry$

$p.e \equiv p.state = eating$

assign

$H_p : \{p.h \wedge \neg p.tp\}$

$(\forall q : N(p, q) \wedge fork(p, q) = q \wedge clean(p, q) : req(p, q) := q;$

$S := S \cup \{q\};)$

$P_p : \{req(p, q) = p \wedge fork(p, q) = p \wedge \neg clean(p, q)\}$

$fork(p, q) := q;$

$clean(p, q) :=$ **true**;

E_p $\{p.h \wedge (\forall q : N(p, q) : fork(p, q) = p \wedge (clean(p, q) \vee req(p, q) = q))\}$

$p.state := eating;$

$(\forall q : N(p, q) : clean(p, q) := false);$

$R_p : \{req(p, q) = p \wedge fork(p, q) = p \wedge \neg p.tp\}$

$fork(p, q) := q;$

$clean(p, q) := \neg clean(p, q);$

$F_p : \{fork(p, q) = p \wedge \{q\} \in S\}$

$S := S - \{q\};$

$RR_p : \{S = \{\} \wedge (p.h \vee p.tp)\}$

$$(\forall q : N(p,q) \wedge fork(p,q) = q \wedge clean(p,q) :$$
$$req(p,q) := q;$$
$$S := S \cup \{q\};)$$

The strict biserial algorithm is a further refinement on the biserial algorithm, in which each request in a batch is sent out one-by-one – that is, when a philosopher becomes hungry, it adds the request tokens for all outstanding high forks to a queue (thus forming a batch). Then, the philosopher sends out the first request in the queue. Once it has received the corresponding fork, it sends out the next request in the batch. Thus, the strict biserial algorithm behaves like the biserial algorithm, except for the finer granularity in sending batches of fork requests.

6.1 Data

Here, we show the performance of the biserial and strict biserial algorithms, and showing how their IFL values compare to those of the thresholds algorithm.

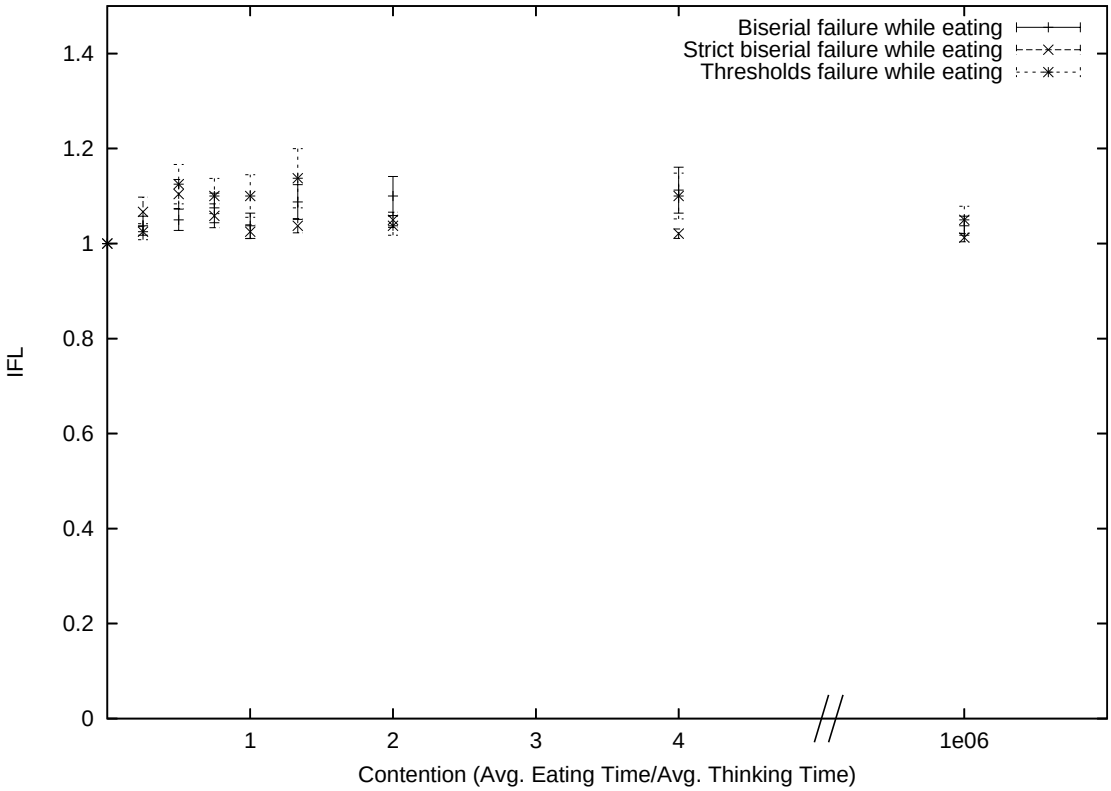

Figure 6.1: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a starburst topology, where a philosopher crashed while eating

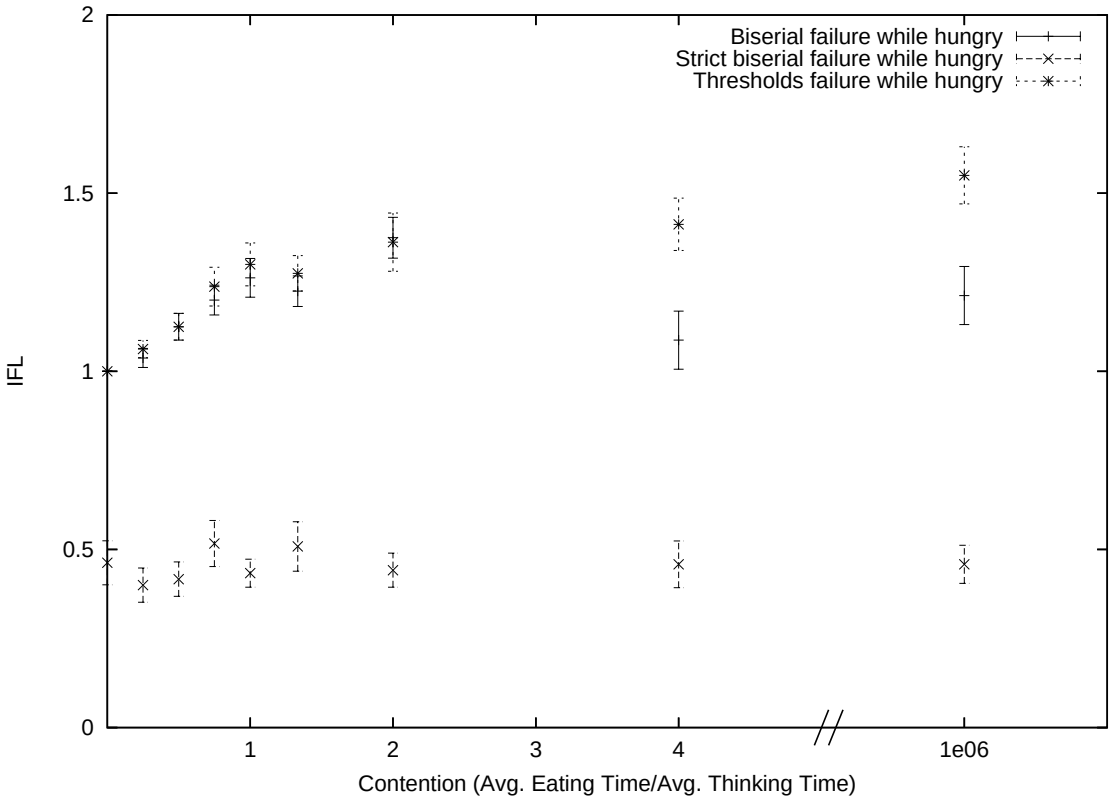

Figure 6.2: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a starburst topology, where a philosopher crashed upon becoming hungry

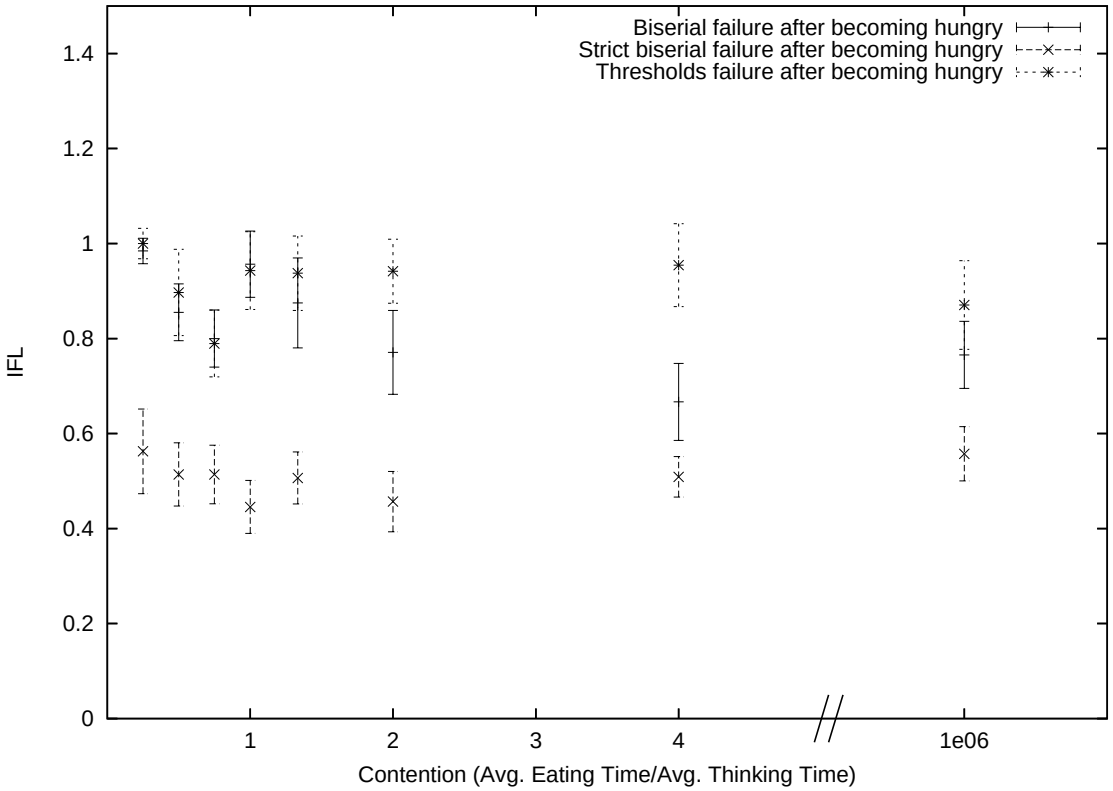

Figure 6.3: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a starburst topology, where a philosopher crashed some time after becoming hungry, before eating

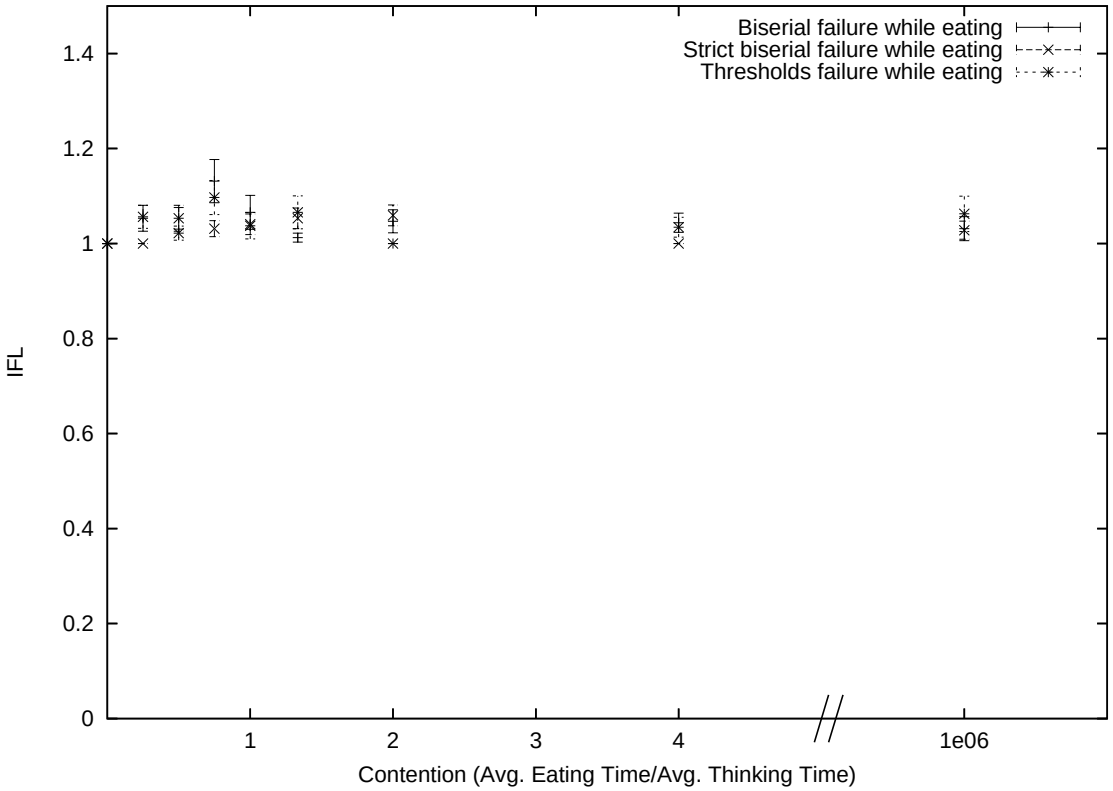

Figure 6.4: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a random topology, where a philosopher crashed while eating

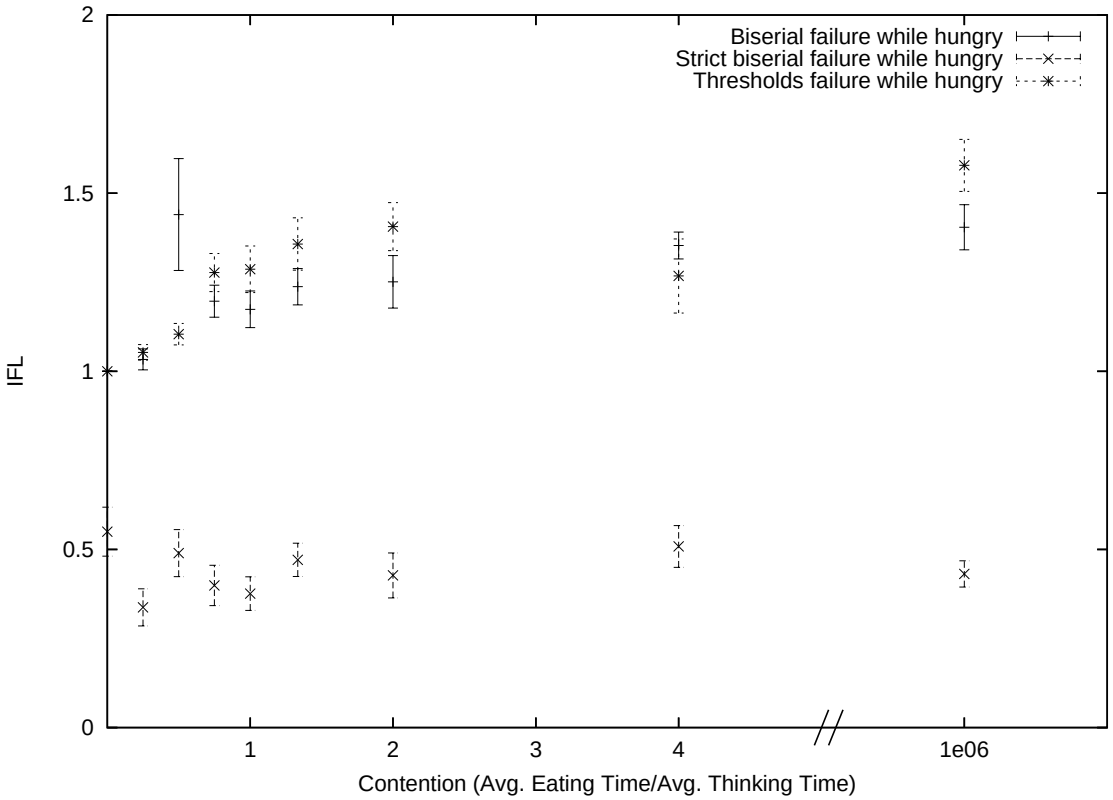

Figure 6.5: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a random topology, where a philosopher crashed upon becoming hungry

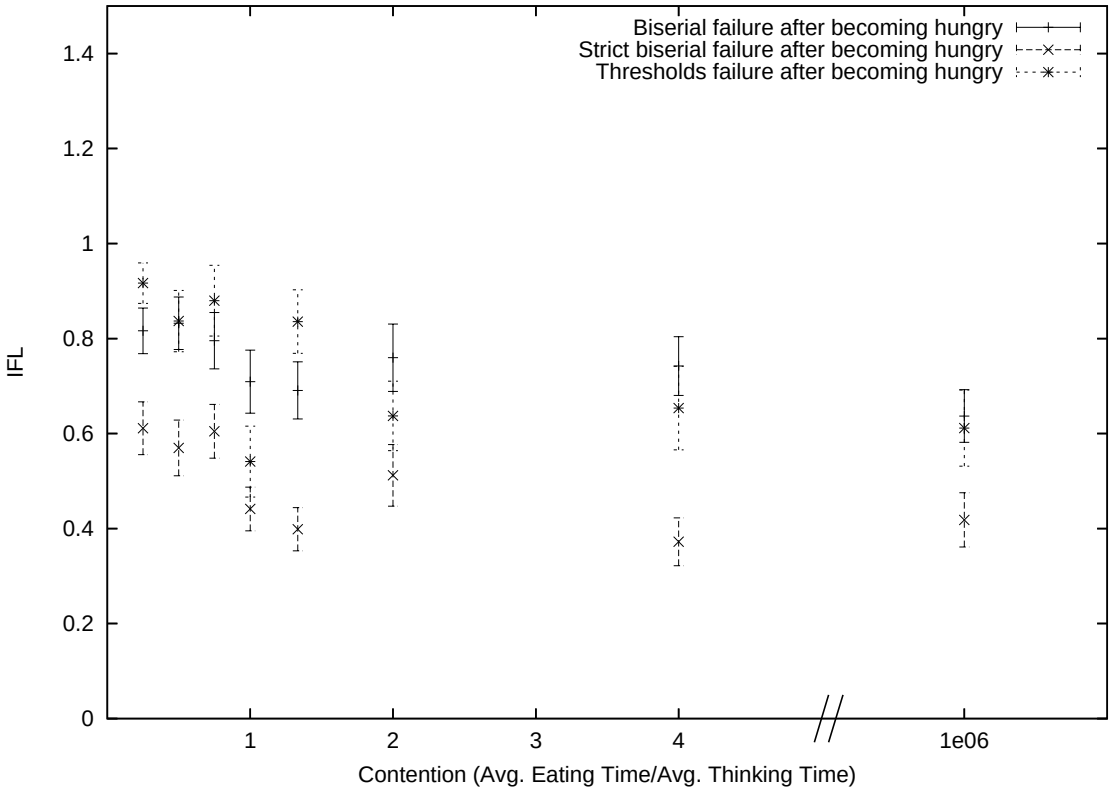

Figure 6.6: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a random topology, where a philosopher crashed some time after becoming hungry, before eating

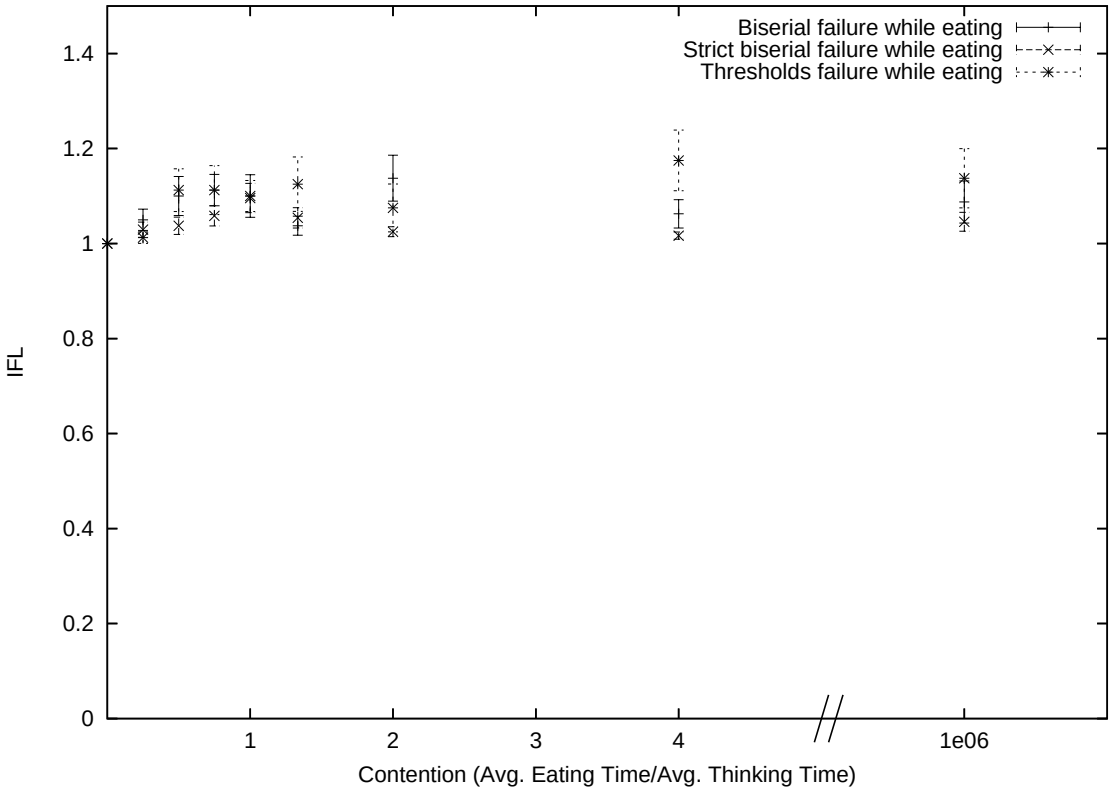

Figure 6.7: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a modified starburst topology, where a philosopher crashed while eating

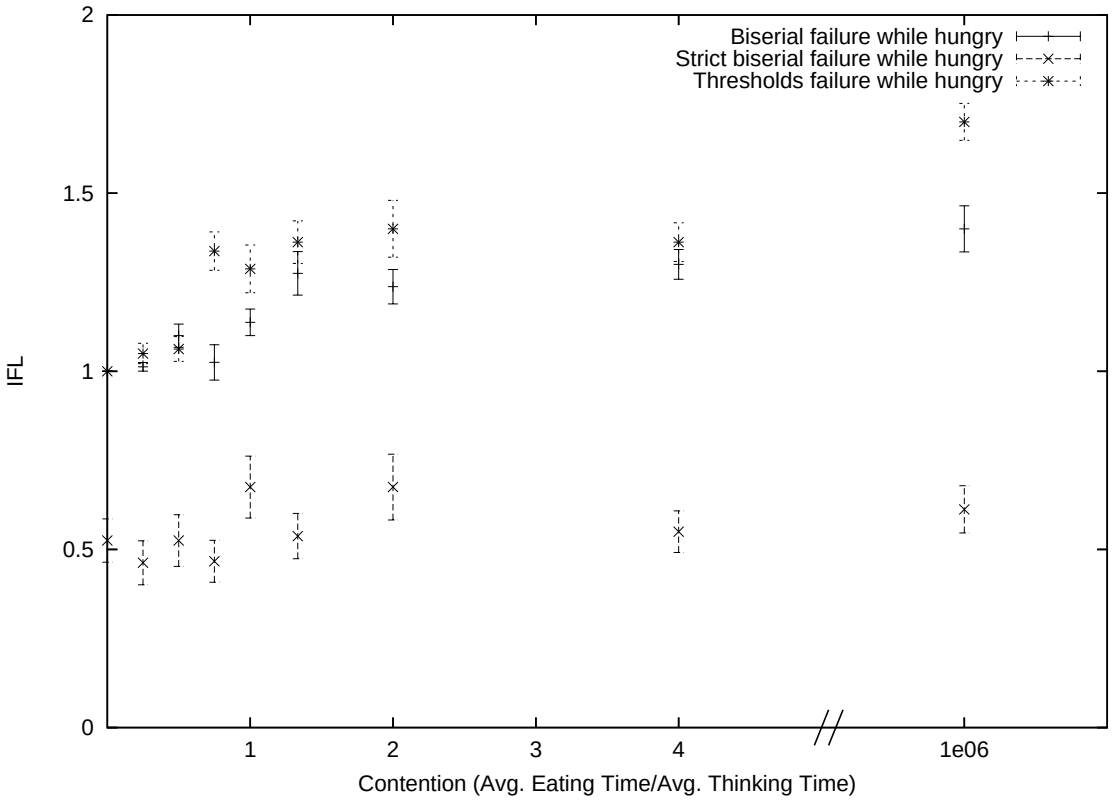

Figure 6.8: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a modified starburst topology, where a philosopher crashed upon becoming hungry

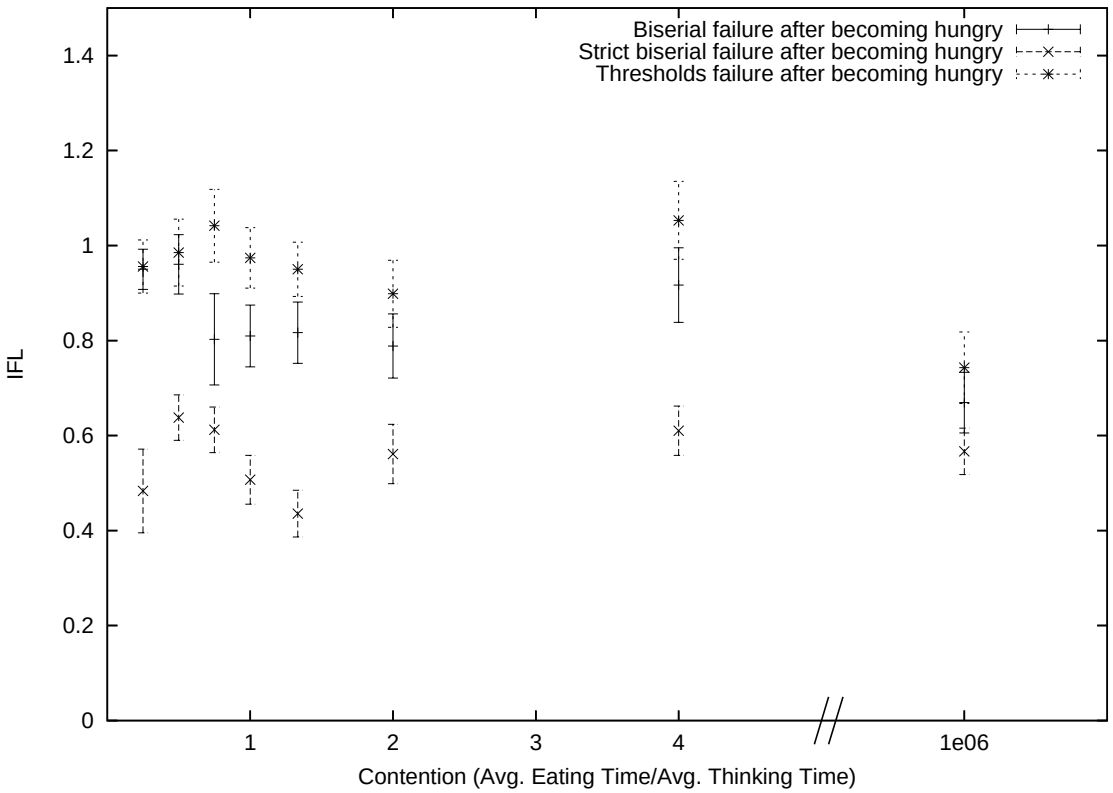

Figure 6.9: IFL for the Biserial, Strict Biserial, and Threshold algorithms on a modified starburst topology, where a philosopher crashed some time after becoming hungry, before eating

6.2 Analysis

The analysis of these algorithms is very similar to the analysis for the dynamic thresholds algorithm. All the phenomena that affect IFL in the dynamic thresholds algorithm similarly affect the IFL for these two algorithms. No new phenomena affect the biserial and strict biserial algorithms which do not affect the dynamic thresholds algorithm.

6.2.1 Failure While Eating

Under failure while eating, both algorithms perform similarly to the dynamic thresholds algorithm, although they both perform slightly better. When a philosopher crashes while eating, all its immediate neighbors starve. As in the dynamic thresholds algorithm, all neighbors of high neighbors starve, since they eventually become low neighbors to the high neighbors, and will never acquire their forks from their high neighbors. This analysis also applies for the biserial algorithm, *except* for the case where a high neighbor to the crashed node is at its threshold point and is preempted by a high neighbor. In that case, none of its neighbors will starve, since the philosopher will relinquish all forks until it receives all low forks, which will never happen since the crashed process is a low neighbor. The advantage of the biserial algorithm is most evident under scenarios of high contention – since there is a greater chance that a process may be preempted. In the strict biserial algorithm, neighbors of high neighbors are affected since the forks are collected one at a time. All neighbors of high neighbors will eventually become low neighbors - however, they may not all necessarily starve. If a high neighbors first requests the fork it shares with the dead philosopher, none of its other neighbors will starve - since it will not request any forks until it receives the fork it shares with the dead node. However, if it requests the dead fork last, then all its neighbors will starve, since it will not relinquish any forks to low neighbors.

The algorithms do not affect the neighbors of low neighbors to a crashed node – none of a low neighbor's neighbors will starve, since low neighbors cannot reach their threshold points.

6.2.2 Failure While Hungry

There are two scenarios to consider when discussing failure while hungry. There's the scenario where a philosopher crashes immediately after sending out it's initial requests, and the scenario where a philosopher fails some time after it becomes hungry but before it eats. First we discuss the former scenario.

As one can see in the previous section, the biserial algorithm behaves much like the thresholds algorithm, but performing slightly better. Recall from Chapter 5 that the reason that the IFL was higher than expected under failure while hungry was because of the stale knowledge effect. This effect is still prevalent under the biserial algorithm, although the IFL is lower than the IFL in the thresholds algorithm. As with the scenario where a process would fail while eating, the most noticeable difference is seen in high contention. The reason for this difference is, again, that in high contention scenarios processes are more likely to be preempted while hungry – which protects a high neighbor's neighbors. The analysis for the strict biserial algorithm in this scenario is not particularly interesting, since the only processes which will starve are those neighbors to the crashed process which did not hold their forks shared with the process, and the one neighboring process whose fork was requested right before the process crashed. For those immediate starving processes which are high neighbors to the crashed node, some of their neighbors may starve – it depends on the order with which they try to acquire forks.

CHAPTER 7

The Double Doorway Algorithm

7.1 The algorithm

The next algorithm we examined was Choy and Singh's Double-Doorway algorithm [2]. Their algorithm relies on static priorities on nodes (called *coloring*) to resolve conflicts on resources, and on a mechanism called a *doorway* to ensure that processes are not successively preempted infinitely often.

7.1.1 Doorway Mechanisms

A doorway [5] is a piece of code such that if a process p finishes executing the doorway code, all neighboring processes are blocked until p finishes eating. A doorway has two parts – entry code and exit code (see Figure 7.1). If a philosopher has not yet executed the entry code, it is said to be outside the doorway. If it has finished executing the entry code, it is said to have crossed the doorway and is now past the doorway. When it finishes the color-based conflict resolution code, it executes the doorway exit code, after which it is outside the doorway again. Choy and Singh define three types of doorways: *asynchronous doorways*, *synchronous doorways*, and *double doorways*.

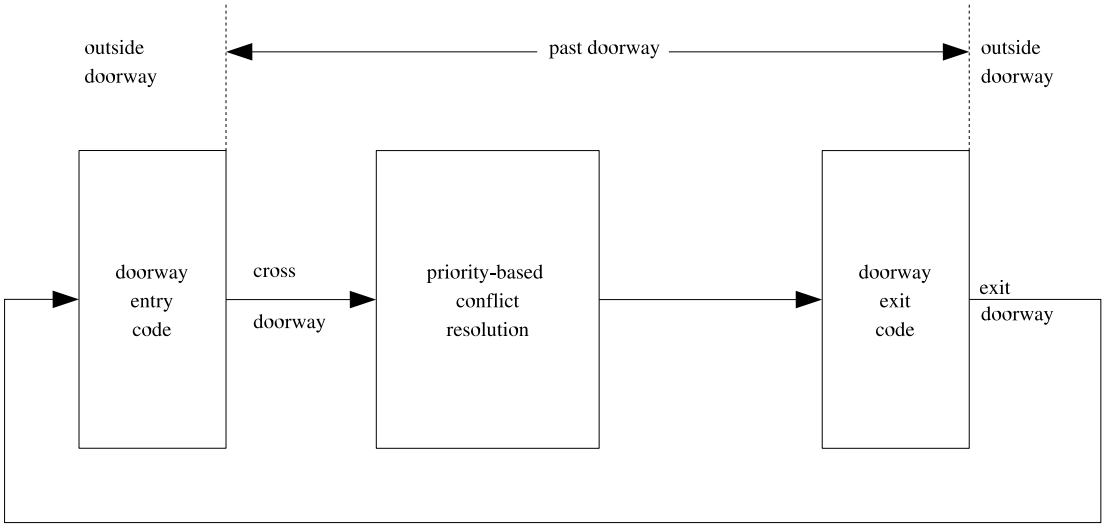

Figure 7.1: The doorway mechanism

Asynchronous Doorways

A simple implementation of a doorway, fulfilling the requirement that a process outside a doorway should be prohibited from crossing the doorway if one if its neighbors is past, is for a process to check whether any of its neighbors are past the doorway. If they are, then it waits until those processes have exited the doorway, after which it enters. This means that if there are two processes waiting upon another process past the doorway, when the process exits, both waiting processes will cross the doorway. This implementation, however, has an exponential response time [2].

The implementation of an asynchronous doorway is as follows: when a philosopher P_i crosses the doorway, it sends a message m_1 to all its neighbors, and broadcasts a different message when it exits the doorway. Processes also keep an array L_i to store the last message received from each neighbor. A process does not enter the doorway until it observes all entries in the array to either currently differ from m_1 or have

been observed to differ from m_1 in the past while the process was trying to execute the doorway code.

Synchronous Doorways

In order to avoid an exponential response time, Choy and Singh then consider a synchronous doorway. A synchronous doorway works as follows: when a philosopher lines up to enter the doorway, it waits at the doorway until it sees that all of its neighbors are outside the doorway simultaneously – that is, it waits to make sure that when it crosses the doorway, it will be the only philosopher past the doorway. This type of doorway still allows for starvation however – it is still possible for a philosopher blocked at the doorway to be preempted infinitely often, thus causing it to remain indefinitely blocked at the doorway. However, a synchronous doorway mechanism properly combined with an asynchronous doorway mechanism removes the possibility of starvation.

The implementation of a synchronous doorway is very similar to the implementation of an asynchronous doorway: when a philosopher P_i crosses the doorway, it broadcasts a message m_2 to all its neighbors. A neighbor of P_i will then be blocked at the doorway until it observes that none of its neighbors are past the doorway – that is, until each of the entries in the array L_i is different from m_2.

Double Doorways

A double doorway is a hybrid of the synchronous and asynchronous doorways. By enclosing the synchronous doorway entry code within an asynchronous doorway (see

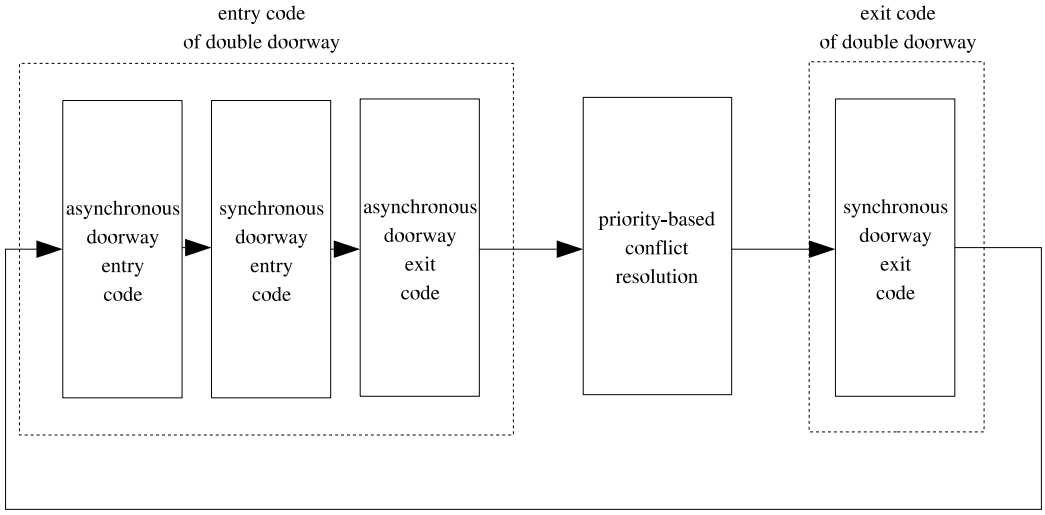

Figure 7.2: An illustration of a double doorway

Figure 7.2), one can avoid both the exponential response time of the asynchronous doorway, and the starvation of the synchronous doorway. (For the proof, see [2]).

The implementation of the double doorway is as follows: when a philosopher P_i becomes hungry, it transitions into a state `wait2` and executes the asynchronous doorway entry code. First, it waits until it observes that none of its low-priority neighbors are past the asynchronous doorway, by verifying that all entries in its L_i array are different from m_1, or if they have been observed to have been different from m_1 while P_i was executing the doorway code. Once it observes these conditions, it broadcasts message m_1 to all its high neighbors, and transitions into the `wait2` state. Then, it executes the synchronous doorway entry code. First, it waits to see that none of its high neighbors has last sent P_i message m_2 – that is, it checks to see that the entries for high neighbors in array L_i are not m_2. Once it has observed all its

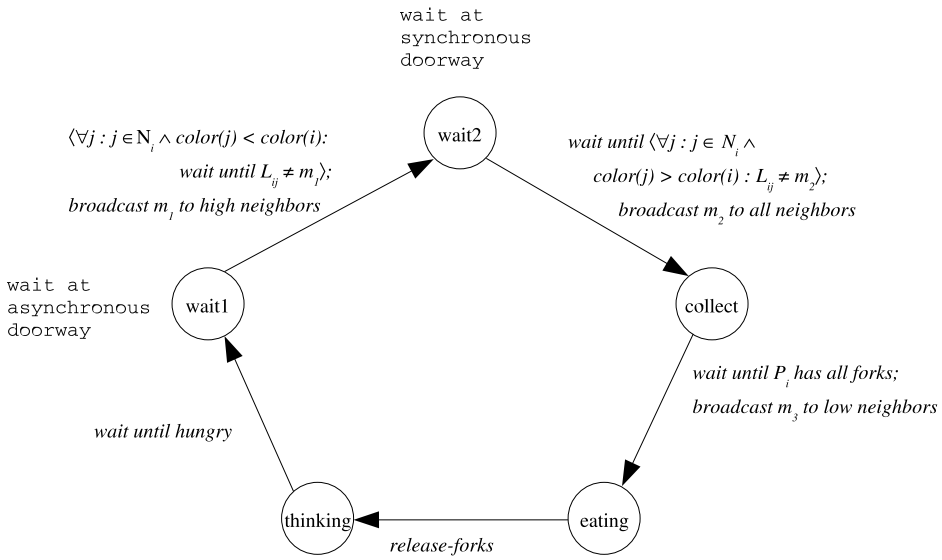

Figure 7.3: The statechart detailing the entry and exit code for the double doorway scheme for a process P_i

high neighbors to be outside the doorway, it broadcasts m_2 to all its neighbors, and transitions into the `collect` state, and sends out its requests. Once it has received all its forks, it broadcasts message m_3 to its low neighbors, and transitions into the `eating` state. It is crucial to note that this mechanism *requires* FIFO channels – that is, messages must arrive in the order that they are sent. If process P_i successively sends messages m_2 and m_3 to process P_j, P_j must receive m_2 first followed by m_3. This is detailed in Figure 7.3.

7.1.2 Forks

In the double doorway algorithm, forks have flags associated with them. These flags are used in case a philosopher relinquishing a fork needs to re-request it. This

flag is set to `true`, if the philosopher sending the fork is past the doorway, and if the fork is being sent to a low-colored high-priority neighbor. Otherwise, when sending a fork, the flag is set to `false`.

7.1.3 The Algorithm

The Double-Doorways algorithm works as follows: initially, all processes are hungry, and, for each pair of neighboring processes in the conflict graph, the forks are distributed to the lower-colored process. When a philosopher becomes hungry, it executes the double-doorway entry code (see Figure 7.3). Once it is in the "collect" state, it checks to see if it holds all forks shared with low-colored (high-priority) processes. If it does, it requests all forks from high-colored (low-priority) neighbors. If it does not, it requests all forks from low-colored processes, after which (once all outstanding requests are satisfied), it requests all forks from high-colored processes. Once it acquires all forks, it eats. When it is finished eating, it exits the double doorway, sets its state to thinking, and relinquishes all requested forks.

When a philosopher receives a request from a high-colored neighbor, it checks to see that it is either outside the doorway or that it does not hold all shared forks with low-colored philosophers. If one of these two conditions is satisfied, it relinquishes the fork. Otherwise, it defers the request. When a philosopher receives a request from a low-priority neighbor, it checks to see that it is either outside the doorway or that it does not hold all its forks. If one of these two conditions is met, it relinquishes the fork and satisfies all deferred requests (which will be to high-colored philosophers). Otherwise, it defers the request.

Upon receiving a fork, a philosopher first checks to see if, upon receipt of the fork, it has received all forks from low-colored neighbors. If it did, and the flag on the fork is set to `true`, it defers the request, and requests all forks shared with high-colored neighbors. Otherwise, if it does not hold all forks shared with low-colored neighbors, it sends back the fork, marking the flag on the fork to `true`.

7.2 Data

Interestingly, there is very little variance in the IFL values of the Double Doorway Algorithm. The IFL values seem to be independent of the contention on the system (unlike the previous three algorithms considered). Instead, the IFL depends entirely upon the forks held when a process crashes, the position of the process in the graph, and the state of each process in the graph (within distance 4 of the crashed node). Note that since the robustness of the Double Doorway Algorithm is independent of the relative response times of processes within the graph (since they are independent of resource contention), it was unnecessary to perform experiments on starburst topology modified to make the difference between the relative response times more uniform.

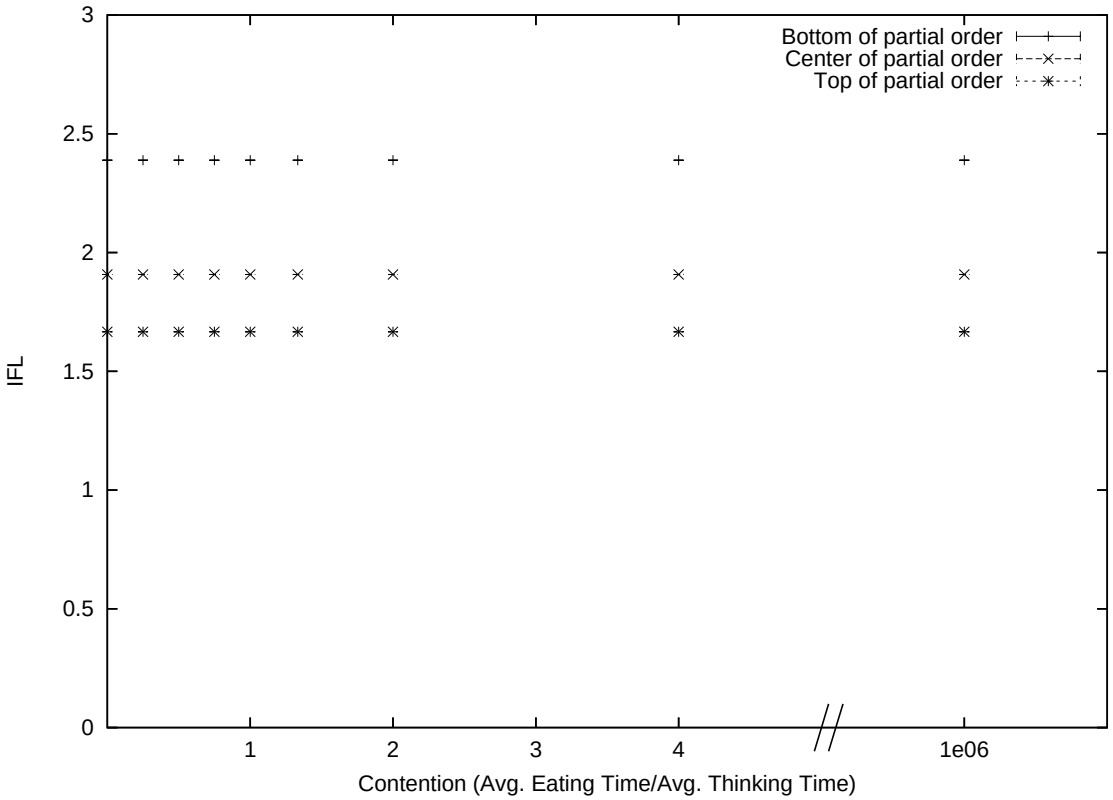

Figure 7.4: IFLs for the Double Doorway algorithm on a starburst topology where the process crashed upon transitioning into the eating state, where each data set represents a different position of the failed node in the static partial order

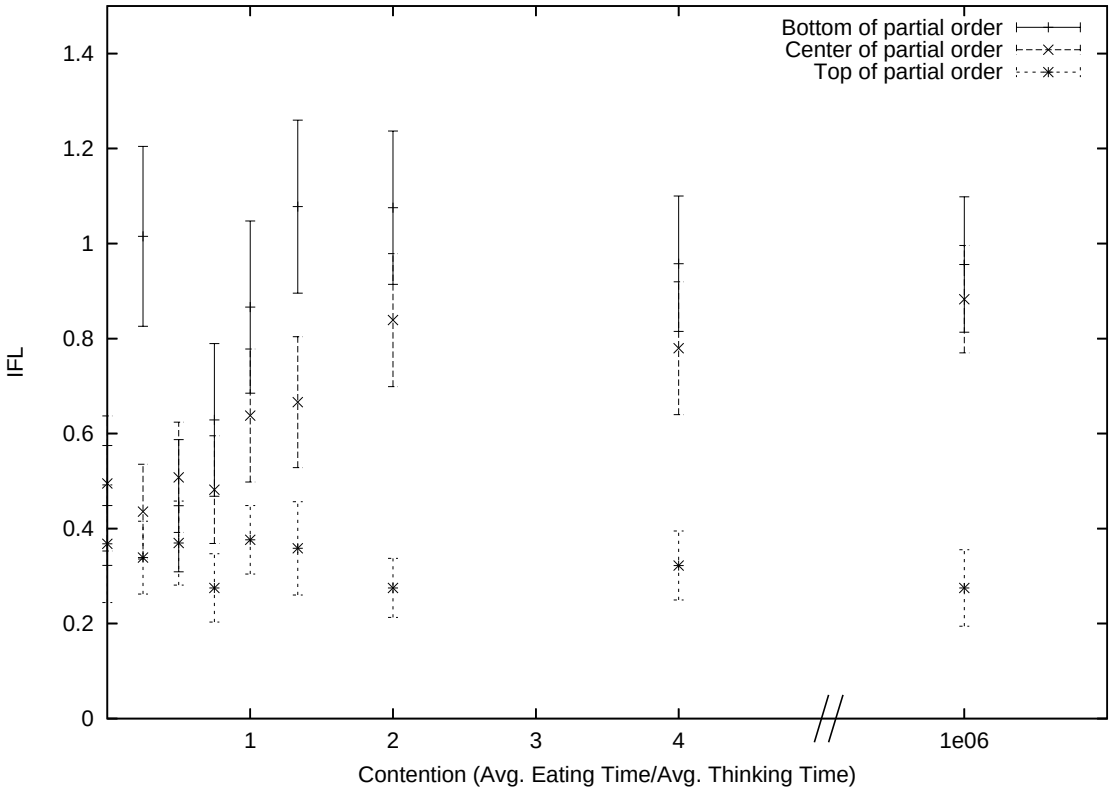

Figure 7.5: IFLs for the Double Doorway algorithm on a starburst topology where the process crashed upon transitioning into the hungry state (i.e. upon lining up at the doorway), where each data set represents a different position of the failed node in the static partial order

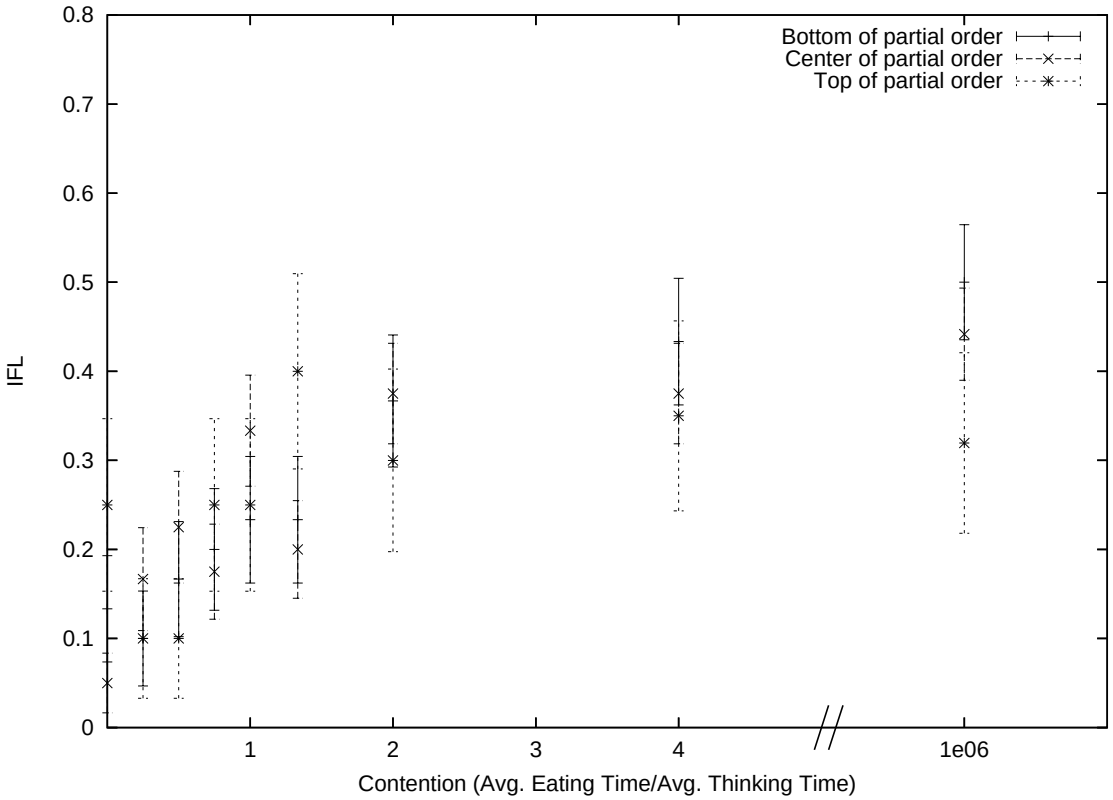

Figure 7.6: IFLs for the Double Doorway algorithm on a starburst topology where the process crashed while not holding any forks, where each data set represents a different position of the failed node in the static partial order

7.3 Analysis

The analysis of the Double Doorway Algorithm relies entirely upon two things: the position and states of processes within the partial order. Suppose a philosopher (A) crashes while eating (see Figure 7.7). All high-priority neighbors will have recorded `m3` as the last message received (meaning that A is in the eating or thinking states), and all low-priority neighbors will have recorded `m2` as the last message received.

Its high-priority neighbors, Processes B eventually become hungry. They will cycle through their states, until they reach the `collect` state – they will not be left

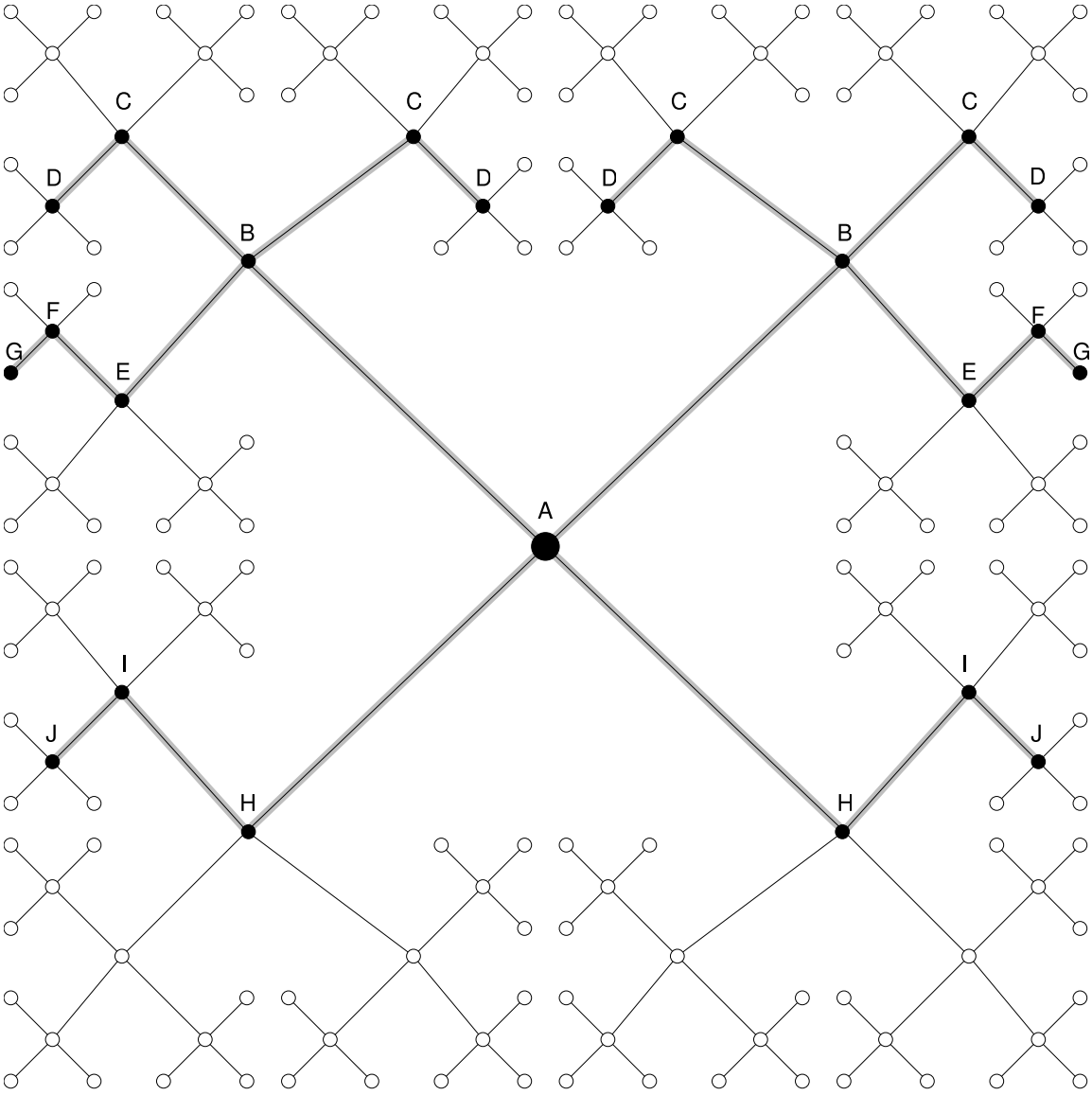

Figure 7.7: Starburst topology where the center node (A) has crashed. The black nodes indicate starving processes, and the white nodes indicate non-starving processes

waiting at `wait1`, since there is no reason their high-priority neighbors will remain inside the doorway, and they will not be waiting at `wait2`, since there is no reason their high neighbors will be caught inside the doorway either. Processes B will remain in the `collect` state however, since they will be waiting indefinitely for forks from A, which they will never receive. Thus, both the low-priority and high-priority neighbors of B will record `m2` as the last message received from B.

After B becomes starving, processes C will eventually become hungry. They will pass through `wait1`, since none of their high-priority neighbors will be stuck inside the doorway. However, they will both stop at `wait2`, since they will both observe their low-priority neighbor as having last sent message `m2`. Therefore, according to the state transition cycle in figure 7.3, their high-priority neighbors will continue to operate normally, since they will have observed `m3` as the last message received from C, and will not get stuck anywhere in the state transition cycle.

Process C's low-priority neighbors will starve, however. The last message they will have received from C will have been `m1`. Because they are high-priority neighbors, processes D will starve, waiting at `wait1`.

Process B's low-priority neighbor, Process E, will become hungry sometime after B has starved. It will transition through `wait1`, since eventually all its high-priority neighbors will send it a message other than `m1`. It will also transition through `wait2`, since its low-priority neighbors will at some point be observed to have last sent a message other than `m2`. It will then transition into the `collect` state, where it will get stuck, because its high-priority neighbor, B, will not relinquish its fork (since it is

also stuck in the `collect` state. Thus, E will starve, and all its neighbors will have observed its last message to be `m2` (since it transitioned out of the `wait2` state.)

Process E's low-priority neighbors are protected from starvation. However, its high-priority neighbor, F, will starve, as it will get stuck at `wait2` – the last message it will receive from E will be `m2`, and it will never receive `m3`. All of F's high-priority neighbors will be protected from starvation, since the last message they will have received is `m3`. However, process G, a low-priority neighbor of F, will starve, since the last message it will have received from F is `m1`. Therefore, G will be waiting indefinitely outside the doorway. This means, however, that none of G's neighbors (other than F) will starve, since they will never observe G inside or past the doorway.

The crashed process' (A) low-priority neighbors, Processes H, will also starve. Upon becoming hungry they will all eventually transition into the `collect` state, where they will remain indefinitely (since Process A will not relinquish any forks.) All of Process H's low-priority neighbors will be safe from starvation, since they will not observe the last message from H to be `m1`.

Process H's high-priority neighbors, Processes I, will starve however. Upon becoming hungry, they will transition through `wait1`, but will be stuck at `wait2`, since the last message it will observe from its low-priority neighbor H will be `m2`.

The only other process affected by Process I's starvation will be its low-priority neighbor, Process J. Process J will starve – it will be waiting indefinitely at `wait1`, because it will observe the last message received from Process I to be `m1`, and will

never observe any other message sent. However, none of Process J's neighbors will starve, since they will all indefinitely observe J to be waiting outside the doorway.

It can be generalized that for all starving processes caught in `wait1`, none of their neighbors (save for their high-priority neighbor(s) waiting indefinitely at `wait2`) will starve.

The question now is how the data presented in the previous section can be explained by this analysis. The analysis suggests that the IFL depends on two factors: the number of forks held by a process upon crashing and the position in the doorway of the crashed process' neighbors and their priorities relative to that of the crashed process. Therefore, when a process crashes while holding on to all its forks, for example, the analysis for which processes starve is similar to the one presented above, adjusted only for the relative distribution of priorities among processes.

Upon crashing when transitioning into the hungry state, the extent of the starvation chain depends on which forks the process holds when it crashes, and the position of its neighbors within its doorway.

Notice, too, that philosophers may also starve when a philosopher crashes while not holding on to any forks. If a philosopher crashes while it's in the doorway, its neighbors may starve since they may be caught waiting indefinitely outside the doorway for the crashed philosopher to cross the doorway.

CHAPTER 8

The Bounded Doorway Algorithm

8.1 The Algorithm

The last algorithm we studied was Choy and Singh's Bounded Doorway algorithm [3]. The Bounded Doorway algorithm is similar to the Hygienic class of algorithms – it ensures a dynamic partial order on the priorities of nodes. Unlike the Hygienic class of algorithms, however, it does this by assigning priority to the philosophers themselves rather than the forks. Choy and Singh define the priorities by stating that, for each pair of neighboring philosophers, the one with the lower ID has higher priority than the one with the higher ID. This means that any conflict over the fork two processes share will always be resolved in favor of the process with the lower ID. The dynamicisim is introduced by requiring that after a philosopher eats, it exchanges its ID with a high neighbor, thereby raising its neighbor's priority above itself. The restriction on the exchange is that a philosopher cannot exchange IDs with the same neighbor indefinitely. For each neighbor A ... N a philosopher may have, it must iterate through them all upon successively eating. In order to do this, each philosopher has a queue of its neighbors' IDs. Upon eating, it selects the first element of the queue which has a higher ID than itself, and tries to exchange IDs with that neighbor. This mechanism prevents local cycles of priority exchanges which may

lock out other processes from eating. Like the Thresholds Algorithm, a philosophers' set of neighbors is partitioned into two sets – a high-priority set and a low-priority set. Upon becoming hungry, a philosopher requests all the forks shared with high-priority neighbors. After it has acquired all forks shared with high priority neighbors, it requests all forks from low-priority neighbors. Upon acquiring all forks, it eats. As in the Thresholds algorithm, if a philosopher is hungry and it receives a request for a fork from a high-priority neighbor, the philosopher relinquishes the fork and re-requests it. If it receives a request for a fork from a low-priority neighbor, it defers the request unless it has not yet acquired all its high-priority forks.

8.1.1 The Doorway mechanism

The bounded doorway mechanism exchanges the IDs of neighboring philosophers. We give the formal algorithm for the ID exchange mechanism in Algorithm 1. It is important to remember that high neighbors have lower priority. The implementation of the doorway assumes that message delivery is FIFO (but it turns out the algorithm does not depend on this for correctness), and that the execution of the procedure is atomic.

Algorithm 1 Implementation of Bounded Doorway

Initially: $ID_i = i \wedge \langle \forall j : j \in N_i : ID_{ij} = ID_j \rangle \wedge ack_i L_i$ contains all $j \in N_i$;

On *receiving message* $\langle exchg, NEWID \rangle$ *from j:*
  if $(ack_i \wedge ID_i > NEWID)$ **then**
    ExchgId(NEWID, j)
  else
    send $\langle exchg - no \rangle$ to j
  end if

On *receiving message* $\langle$ *exchg-yes, NEWID* $\rangle$ *from j:*
  $ID_{ji} := ID_i$;
  $ID_i := NEWID$;
  $ack_i := true$;
  for each $(k \in N_i)$ **do**
    send $\langle$ *newid, NEWID* $\rangle$ to k;
  end for

On *receiving message* $\langle$ *exchg-no* $\rangle$ *from j:*
  $ack_i := true$;

On *receiving message* $\langle$ *newid, NEWID* $\rangle$ *from j:*
  $ID_{ji} := NEWID$;

procedure RaiseId
  $j :=$ GetFirstHigh(L_i);
  if $(j \neq null)$ **then**
    MoveToEnd(j, L_i);
    $ack_i := false$;
    send $\langle exchg, ID_i \rangle$ to j;
    wait until ack_i;
  end if
end procedure

procedure ExchgId($NEWID, j$)
  $ID_{ji} := ID_i$;
  send $\langle$ *exchg-yes,* $ID_i \rangle$ to j;
    for each $(k \in N_i)$ **do**
    send $\langle$ *newid, NEWID* $\rangle$ to k;
  end for
  $ID_i := NEWID$;
end procedure

8.2 Data

Like the thresholds algorithm, the bounded doorway algorithm has failure locality 2. However, the average-case performance of this algorithm is considerably worse than the thresholds algorithm, when the philosopher crashes while eating. Interestingly, however, we get much better IFL in the bounded doorways algorithm when it crashes upon becoming hungry. In order to measure its failure locality, we used the same starburst topology we used for the thresholds algorithm, as well as the same real-world topology. The results of the simulations follow:

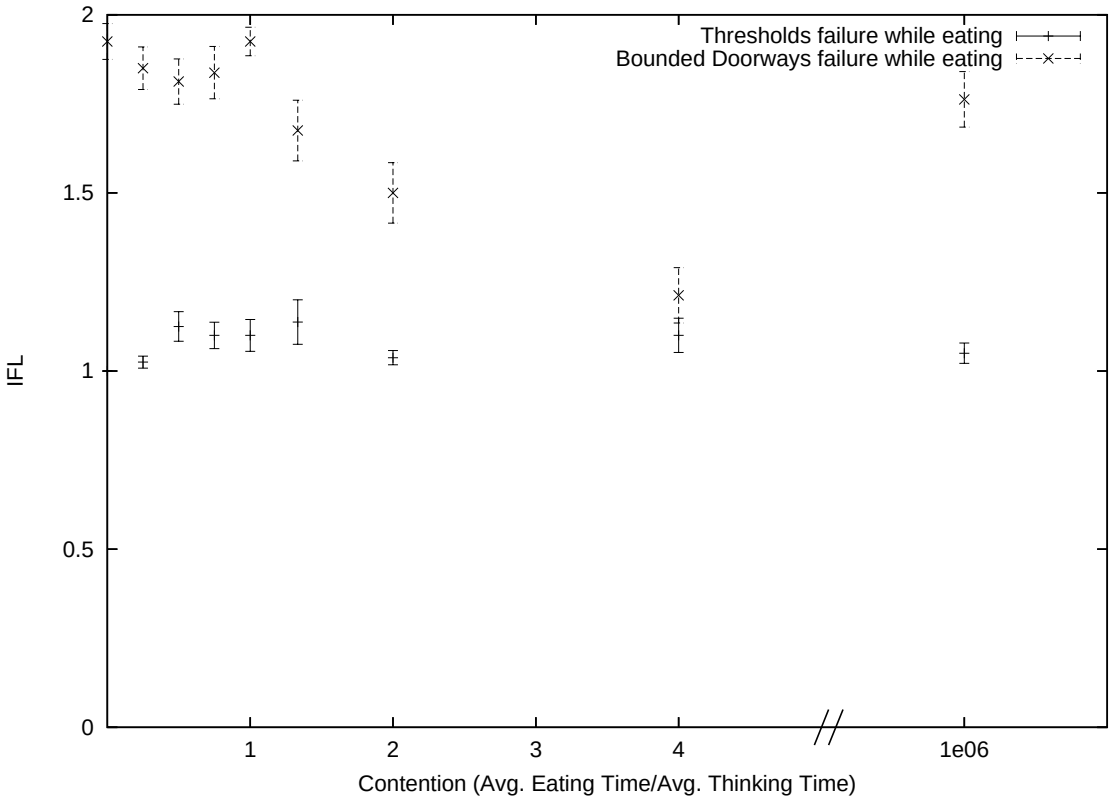

Figure 8.1: IFLs for the Bounded Doorway algorithm and the Thresholds algorithm on a starburst topology where the process crashed upon transitioning into the eating state

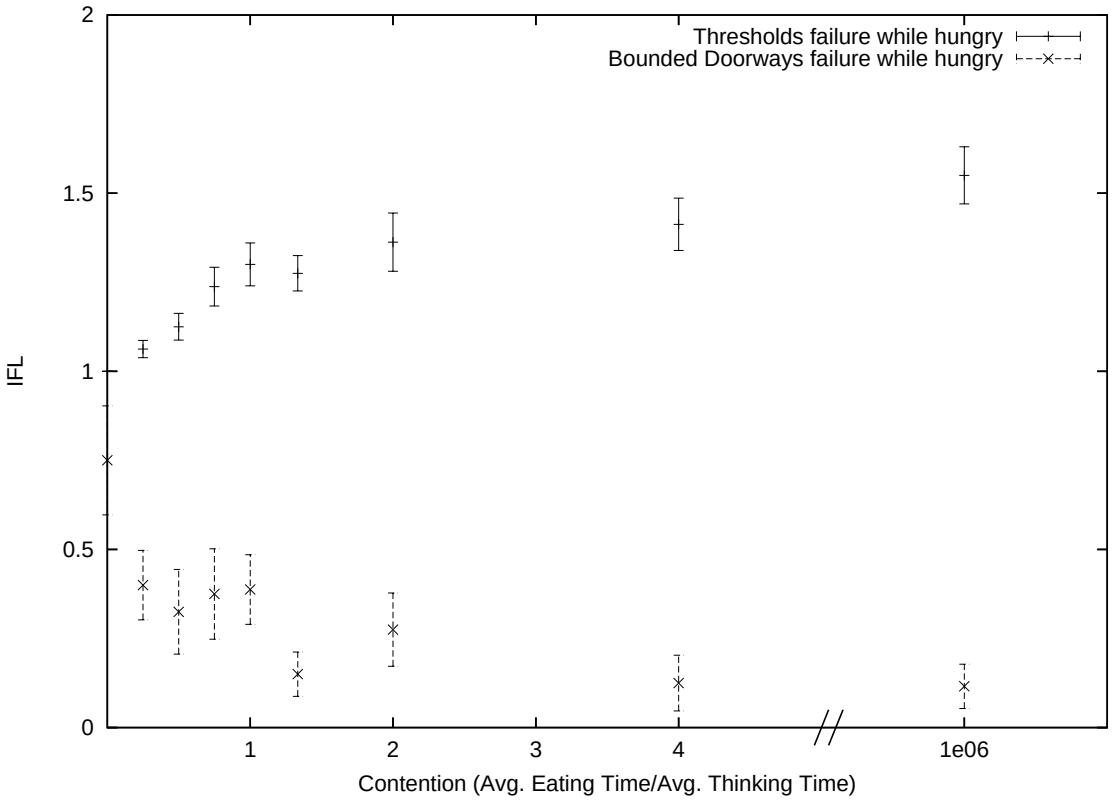

Figure 8.2: IFLs for the Bounded Doorway algorithm and the Thresholds algorithm on a starburst topology where the process crashed upon transitioning into the hungry state

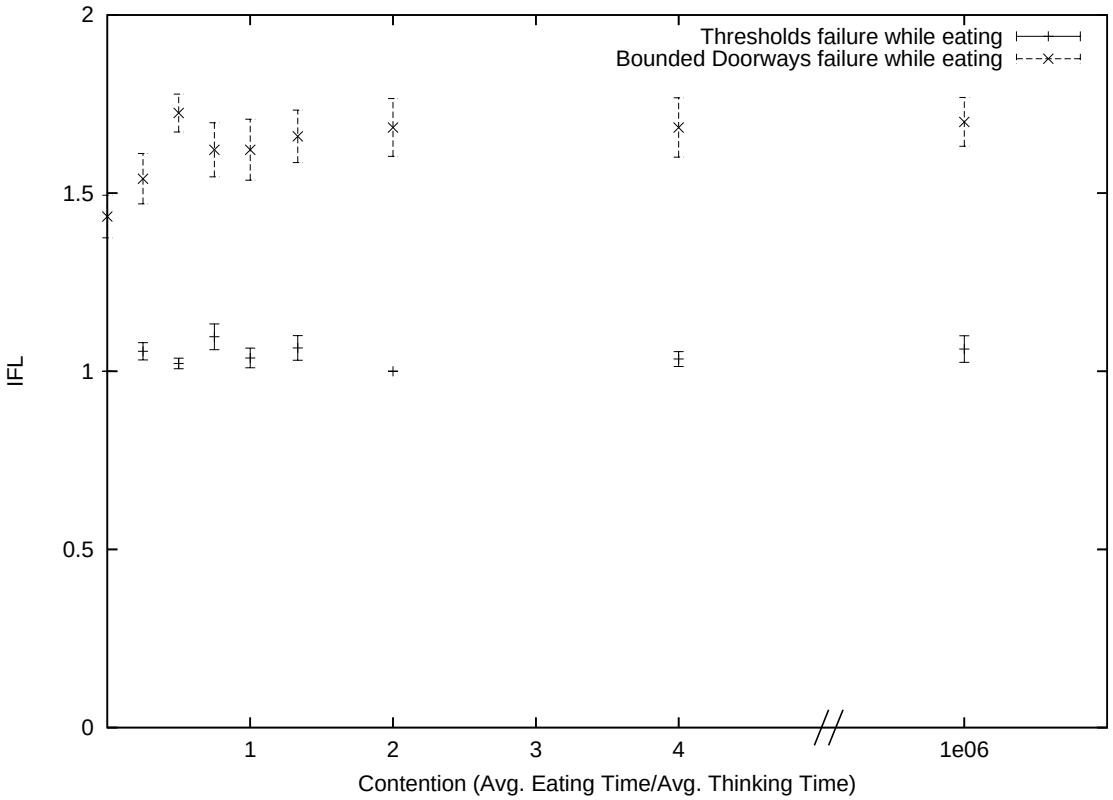

Figure 8.3: IFLs for the Bounded Doorway algorithm and the Thresholds algorithm on a random topology where the process crashed upon transitioning into the eating state

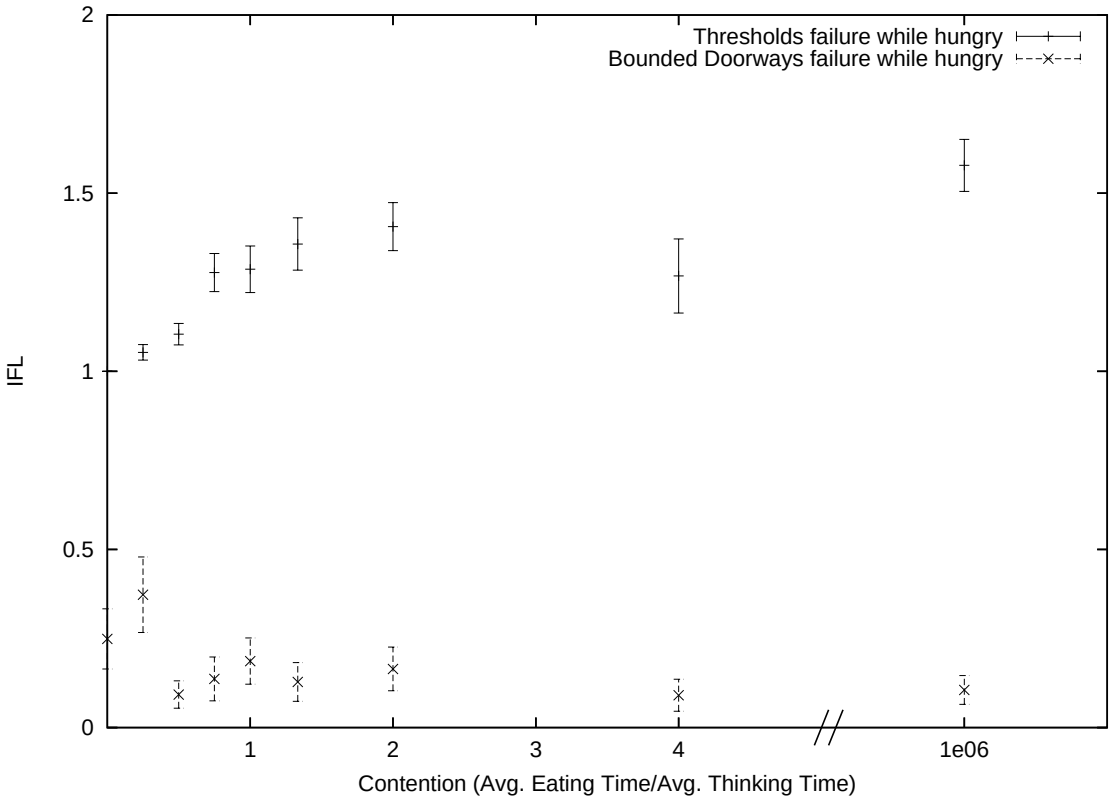

Figure 8.4: IFLs for the Bounded Doorway algorithm and the Thresholds algorithm on a random topology where the process crashed upon transitioning into the hungry state

8.3 Analysis

8.3.1 Failure while eating

The similarity of the bounded doorway algorithm to the thresholds algorithm discussed in chapter 5, particularly with respect to the way forks are allocated, would lead one to believe that the two algorithms would have similar average integrated failure localities. The data indicates otherwise. Despite the fact that both have worst-case failure locality 2, the average behavior of the bounded doorways algorithm is much worse when a philosopher crashes holding all its forks. The analysis for the

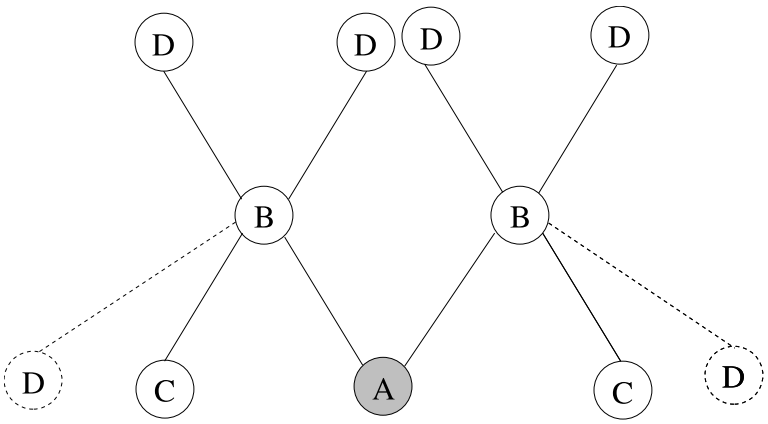

Figure 8.5: Scenario where a philosopher crashes while holding onto forks shared with high-priority neighbors

bounded doorways algorithm is similar to that for the thresholds algorithm. When a philosopher crashes holding all its forks, all its immediate neighbors starve. Furthermore, all neighbors of its high-priority neighbors will also starve. The reason for this is the following: suppose, as in Figure 8.5, process A crashes while holding onto all its forks. Its immediate neighbors B will starve once they become hungry. B's high-priority neighbors, C will starve, since B will not relinquish forks to high-ID neighbors once they acquire their high-priority forks. However, B will relinquish forks to low neighbors D. Once D eat, however, they will raise their ID (thereby lowering their priority). The algorithm ensures that eventually, D's ID will be higher than B's – meaning that, once lowered, the next time D becomes hungry, it will starve since B will not relinquish forks to high neighbors.

Unlike the Thresholds algorithm, however, not all neighbors of low-priority neighbors are immune from starvation. In fact, more often than not, all neighbors of

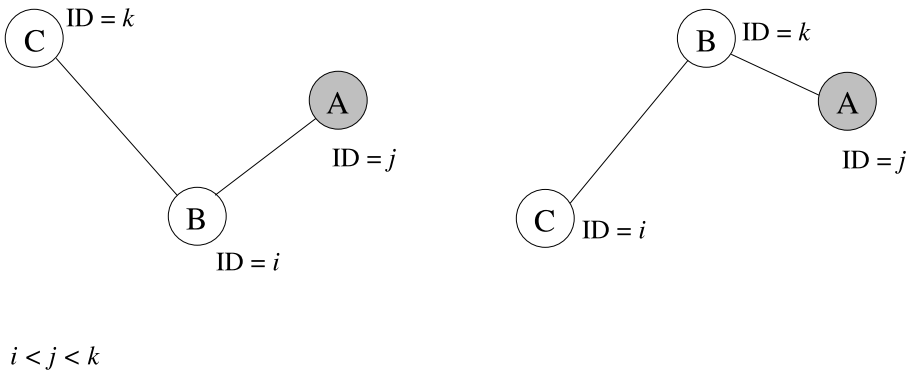

Figure 8.6: Scenario where a philosopher crashes while holding onto forks shared with high-priority neighbors

low-priority neighbors will also starve. This is a result of a very interesting property of the bounded doorways algorithm, namely that it's not only the relative priority ordering between philosophers that's important, but also the relative height (i.e. the differences in priority between a philosopher with ID 1 and a neighbor with priority, say, 99). To illustrate, please refer to Figure 8.6. It turns out that it is very likely that a crashed philosopher's immediate low-priority neighbors will eventually become high-priority neighbors, leading to IFL of 2.0. The reason is as follows. Suppose that we have the topology and the ID distribution illustrated in Figure 8.6. Process A crashes while holding onto its shared fork with B, B eventually becomes hungry and will starve. Then, process C becomes hungry, eats, and exchanges its ID with B. However, notice that C has a higher ID than A, which means that when C swaps IDs with B, B becomes a high-priority neighbor to A, which, as explained above, leads to a starvation chain of length 2.

This analysis can be generalized. Note that this is predicated on the fact that in the starburst topology, when a philosopher crashes it partitions the graph into four subgraphs, where the crashed node forms a barrier for ID exchanges. That is, IDs cannot be exchanged between the subgraphs across the crashed process. Suppose we have the topology and ID distribution as illustrated in Figure 8.7. A is crashed, and B is a low-priority neighbor to A, and none of B's neighbors have a higher ID than A. While it may initially seem that we would get a starvation chain of length 1, it turns out that if some process in the subgraph $\mathcal{X}$ has an ID higher than A, B will probably eventually acquire it, simply through the combination of random ID exchanges. This will lead to a situation where B becomes a high-priority neighbor of A, leading to a starvation chain of length 2. In order to avoid this situation, the crashed process must have an ID higher than any other philosopher which can be reached through a low-priority neighbor (i.e. in the subgraphs $\mathcal{X}$ and $\mathcal{Y}$, with no edges between any of the philosophers with other philosophers which can be reached through a high-priority neighbor (see Figure 8.7). So, assuming that, as in Figure 8.7, we have n subgraphs with m processes in each, there is a $\frac{1}{mn}$ chance that A has the highest priorities, ensuring an IFL of 1.0. If, say, only $\mathcal{X}$ does not contain an ID which is lower than A's, but the other three subgraphs do contain an ID which is lower than A's, we would get an IFL of 1.75 for the topology in Figure 8.7.

This analysis completely fits the data. When a philosopher crashes under minimal contention, on average half its neighbors will have high priority and the other half will have low priority. As stated above, all high priority neighbors and all neighbors of high priority neighbors will starve. All low-priority neighbors will starve, and all their

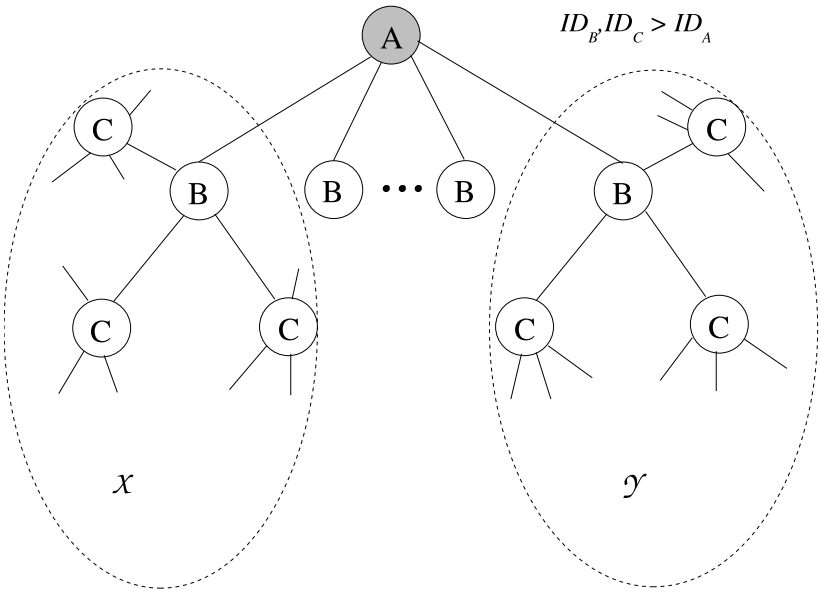

Figure 8.7:

neighbors are likely to starve. That likelihood is the probability that, upon failing, the crashed node has an ID which is lower than all philosophers which are lower in the priority partial order. (Note again that this is only in the case of a starburst topology). Generally speaking, the higher the contention in the system, the greater chance there is that a philosopher must be at the top of the partial order to eat. This means that it is likely to have more subgraphs which are lower than itself in the partial order (as in Figure 8.7). This consequently means that it's more likely that at least one of the subgraphs does *not* contain an ID which is lower than the crashed node, which would protect the distance 2 neighbors within that subgraph from the crashed node.

8.3.2 Failure while hungry

Recall from Chapter 5 that the Thresholds algorithm will always have an IFL greater than 1 when it fails upon becoming hungry, due to the stale knowledge effect. The stale knowledge effect plays no role in the bounded doorways algorithm, since whenever a philosopher's priority changes due to it receiving a new ID, all its neighbors are notified. Therefore, even with channel delays, it is guaranteed that eventually each philosopher's knowledge about its place in the partial order will be accurate. Therefore, the only neighbors which starve will be neighbors whose forks the crashed philosopher holds upon crashing, and possibly their immediate neighbors, according to the analysis presented above.

CHAPTER 9

Conclusion

In this chapter, we summarize the data and analyses of the dining philosophers algorithms we have examined in our research.

9.1 The Hygienic Solution

In Chapter 2, we introduced the dining philosophers problem and the hygienic solution as a simple solution, ensuring that in the absence of process failure, the requirements of both mutual exclusion and progress are achieved. In Chapter 3, we have analytically shown that in the presence of a failed process, the requirement of progress is violated for every process in the graph. The worst-case failure locality of the hygienic solution is the diameter of the conflict graph, since, when a philosopher crashes at the top of the partial order while holding onto all its forks, it does not relinquish any of its forks, which means that its neighbors will not relinquish any of their forks, etc. We have furthermore shown that the integrated failure locality is also the diameter of the graph, because of the property of the hygienic algorithm that eventually, the priority partial order stabilizes (i.e., becomes static), with a starving or crashed process at the top of the partial order, meaning that it will not relinquish forks to low neighbors. Those low neighbors will not relinquish forks to their low

neighbors, and so forth, until the starvation chain expands throughout the graph. It turns out that for the hygienic solution, when a philosopher crashes all philosophers in the connected graph always starve. Thus, the worst-case failure locality is equal to the best-case failure locality, which is equal to the IFL.

9.2 The Thresholds Algorithm

In Chapter 5 we discussed the thresholds algorithm. We have shown that, while its worst-case failure locality is 2 (that is, at most the starvation chain extends two nodes away from the crashed process), its expected IFL is much lower. Its worst-case failure locality occurs when the following conditions occur:

1. each philosopher's knowledge of its position in the partial order (i.e., its knowledge of the state of the cleanliness of all its forks, whether or not it is holding the fork) is accurate

2. the crashed philosopher is at the bottom of its partial order

3. the crashed philosopher was holding all its forks at the time of failure.

The expected IFL of the thresholds algorithm depends upon its state when it crashed. When a philosopher crashed while eating, the average IFL ranged between 1.0 and 1.25. However, when a philosopher crashed while hungry, the average IFL ranged between 1.0 and 1.7 (depending on the topology). The primary phenomenon responsible for the expected IFL values observed is the stale-knowledge effect, which we describe in Section 5.4.1. The stale knowledge effect leads to a lower-than-expected IFL for scenarios where a philosopher crashes while eating, and to a higher-than expected IFL for scenarios where a philosopher crashes while hungry and not holding

all its forks. The principle behind the stale knowledge effect is that philosophers have different notions of their positions in the priority partial order. That is, one philosopher may think that it is a low-priority neighbor to another philosopher, while the other philosopher may know (by holding the fork), that its neighbor is a high-priority neighbor. This contradiction in belief in the local state between philosophers impacts the order in which philosophers request forks from their neighbors and the conditions under which they release the forks, which leads to a lower-than-expected or higher-than-expected (depending on the failure conditions) IFL.

Another effect which impacts the IFL is the relative response times between philosophers, which is a function of local connectivity, as explained in Section 5.4.2. The lower the average response time of a philosopher, the more often it will have a chance of becoming hungry. This means that a philosopher with a lower average response time is more likely to be a low neighbor to a philosopher with a higher average response time, which means that, if the process more likely to be a low neighbor crashes, the IFL is more likely to be closer to 2.0 (assuming we do not factor in the stale-knowledge effect); and if the process more likely to be a high neighbor crashes, the IFL is more likely to be closer to 1.0.

Finally, the local topology of 1-ring neighbors also has an impact on the average IFL, since in certain configurations, 1-ring philosophers may be able to isolate the starvation chain. (See Figure 5.9 and Section 5.4.3).

9.3 The Biserial and Strict Biserial Algorithms

In Chapter 6, we introduce the Biserial and Strict Biserial algorithms, which are variants of the thresholds algorithm discussed in Chapter 5. We showed that as variants, they are subject to the same phenomena which account for the IFL values observed in the thresholds algorithm. We also demonstrated that since, in the variants, there is a finer degree of granularity when it comes to the way the processes request and relinquish forks (especially that, in the strict biserial, philosophers request forks one-by-one), we get slightly lower IFL values for the biserial algorithm and dramatically lower IFL values for the strict biserial algorithm when a philosopher crashes while hungry. Note, however, that the dramatically lower IFL values in the strict biserial algorithm comes at the expense of a dramatically increased response time.

9.4 The Double Doorway Algorithm

In Chapter 7, we discuss a very different algorithm based on the concept of a doorway, which is a piece of code to ensure local progress (i.e. progress between sets of neighboring philosophers, which translates into global progress. [2]). This algorithm is quite different from the others we have examined, as it has a static priority on edges. That is, the priority ordering between philosophers does not change. This is important to note, since it appears that the IFL of a system depends on only two parameters: the number of forks a philosopher held when it crashed, and its permenant position in the priority ordering. The average IFL of a philosopher when it crashed while eating is fairly high compared to the previous algorithms considered, exempting the hygienic solution. This is due to the fact that, as we explain in the analysis in Section

7.3, the worst-case failure locality is 4.0. However, when a philosopher crashes while hungry, we get fairly low expected IFLs, since the IFL relies only on a philosopher's position within the partial order and the number of forks it holds when it fails. The most interesting result we get from this is that the expected IFL when a philosopher crashes while not holding onto any forks is not 0.0, but is between 0.0 and 0.5. The reason for this is that even when it's not holding onto any forks, its neighbors still need to negotiate with it in order to eat, since they need to get past the doorway, and they cannot when any of their neighbors are unresponsive.

9.5 The Bounded Doorway Algorithm

Finally, the last algorithm we studied was the bounded doorway algorithm in Chapter 8. Despite its similarity to the thresholds algorithm, it displayed quite different behavior under process failure. Both the bounded doorway and the thresholds algorithms have worst-case failure locality 2. However, upon failure while eating, the bounded doorways algorithm displays an IFL quite close to 2, which decreases towards 1.0 as the contention is increased. The reason for this is the mechanism for ensuring a dynamic priority ordering, namely, the swapping of IDs. As in the thresholds algorithm, all neighbors of high-priority neighbors starve, and no neighbors of low-priority neighbors starve. However, unlike the thresholds algorithm, unless the crashed philosopher has the lowest ID among processes in the low-priority subgraphs (see analysis in Section 8.3), those low-priority neighbors will become high-priority neighbors, causing their neighbors to starve, leading to an IFL of 2.0. However, as one increases the contention, one increases the probability that a philosopher must be at the top of the partial order in order to eat, and the greater the chance that it

will have an ID low enough so that it does not get any high-priority neighbors after failing.

9.6 Comparison of Algorithm Robustness

The most robust algorithms under failure while eating are the Biserial, Strict Biserial, and Threshold algorithms, due primarily to the stale knowledge effect. The stale knowledge effect causes all three algorithms to behave similarly, leading to very close IFL values (within each others' margins of error).

Under failure while hungry, the same effect which leads to the robustness of the three algorithms above causes a worse measure of robustness for those algorithms. The algorithm with the best performance under failure while hungry is the Bounded Doorways algorithm. The lack of stale knowledge allows us to achieve an integrated failure locality which decreases as the contention on resources increases.

BIBLIOGRAPHY

[1] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646, 1984.

[2] Manhoi Choy and Ambuj K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems*, 17:535–559, 1995.

[3] Manhoi Choy and Ambuj K. Singh. Localizing failures in distributed synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 7:705–716, July 1996.

[4] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[5] Leslie Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1(2):77–85, 1986.

[6] Paolo A.G. Sivilotti, Scott M. Pike, and Nigamanth Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2:524–529, 2000.