Abstract of "Distributed Queuing and Applications" by Srikanta Tirthapura, Ph.D., Brown University, May 2003.

Distributed Queuing is a fundamental distributed coordination problem, arising in a variety of applications, such as ordered multicast and distributed *fetch-and-ϕ* data structures.

In the distributed queuing problem, processors in a network asynchronously and concurrently request to join a distributed queue. A distributed queuing protocol organizes these requests into a queue, and each request learns the identity of its successor in the queue.

In this thesis, we present the first systematic study of distributed queuing and demonstrate its importance as a fundamental building block in building distributed data structures.

We focus our attention on the *Arrow queuing protocol*, which has been observed to perform well in practice. We present the first analysis of the Arrow protocol under *concurrent* access to the queue, improving on previous work which analyzed the protocol under sequential access. Our analysis shows that the Arrow protocol is competitive to the "optimal" in the concurrent case.

While fault tolerant distributed queuing is impossible in the face of process failures, we show how to make the Arrow queuing protocol *self-stabilizing*, so that it can automatically recover from failures. Our self-stabilizing queuing protocol is scalable, and stabilizes quickly into a legal state.

We show a novel application of distributed queuing to *ordered multicast*. There is some evidence that ordered multicast using distributed queuing has lower latency than previously studied approaches to ordered multicast which use distributed counting.

We present a construction of a general purpose parallel and distributed *fetch-and-ϕ* data structure based on distributed queuing. Our algorithm takes a sequential implementation of a data structure and converts it into a parallel and distributed implementation. Our algorithm has significant advantages over its only other competitor we know of, the Combining trees.

Abstract of “Distributed Queuing and Applications” by Srikanta Tirthapura, Ph.D., Brown University, May 2003.

Distributed Queuing is a fundamental distributed coordination problem, arising in a variety of applications, such as ordered multicast and distributed *fetch-and-ϕ* data structures.

In the distributed queuing problem, processors in a network asynchronously and concurrently request to join a distributed queue. A distributed queuing protocol organizes these requests into a queue, and each request learns the identity of its successor in the queue.

In this thesis, we present the first systematic study of distributed queuing and demonstrate its importance as a fundamental building block in building distributed data structures.

We focus our attention on the *Arrow queuing protocol*, which has been observed to perform well in practice. We present the first analysis of the Arrow protocol under *concurrent* access to the queue, improving on previous work which analyzed the protocol under sequential access. Our analysis shows that the Arrow protocol is competitive to the “optimal” in the concurrent case.

While fault tolerant distributed queuing is impossible in the face of process failures, we show how to make the Arrow queuing protocol *self-stabilizing*, so that it can automatically recover from failures. Our self-stabilizing queuing protocol is scalable, and stabilizes quickly into a legal state.

We show a novel application of distributed queuing to *ordered multicast.* There is some evidence that ordered multicast using distributed queuing has lower latency than previously studied approaches to ordered multicast which use distributed counting.

We present a construction of a general purpose parallel and distributed *fetch-and-ϕ* data structure based on distributed queuing. Our algorithm takes a sequential implementation of a data structure and converts it into a parallel and distributed implementation. Our algorithm has significant advantages over its only other competitor we know of, the Combining trees.

Distributed Queuing and Applications

by
Srikanta Tirthapura
B. Tech., Indian Institute of Technology, Madras, 1996
Sc. M., Brown University, 1998

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2003

Vita

Srikanta Tirthapura was born in 1975 in Bangalore, India. He was initially educated in Bangalore in the following wonderful schools and colleges: Sri Aurobindo Memorial School (till 1987), Vijaya High School (1987-1990) and The National College, Jayanagar (1990-1992). In 1992, he entered Indian Institute of Technology (IIT), Madras. He graduated from IIT in 1996 and entered the Ph.D. program in Computer Science at Brown University in fall 1996.

- *Ph.D. in Computer Science, July 2002.*
  Brown University, Providence, RI.
- *Sc.M. in Computer Science, October 1998.*
  Brown University, Providence, RI.
- *B.Tech. in Computer Science, July 1996.*
  Indian Institute of Technology, Madras.

Acknowledgements

I thank my advisor, Maurice Herlihy, for his patience and encouragement, and for the numerous enlightening discussions on research. It has been rejuvenating to see his refreshing approach towards research. I thank Phil Gibbons, for being on my thesis committee, and for the research collaboration. Working with him has been a pleasure, and has helped me in many ways. Thanks to Eli Upfal, for being on my thesis committee, and for teaching me about randomized algorithms. Steve Reiss has been a patient listener of my research questions/ideas, and I'm grateful to him for his advice and help. Thanks to Ugur Cetintemel for advice during my job search. A lot of my thesis work was done in collaboration with Roger Wattenhofer, and I'm grateful for his research companionship. I have enjoyed and benefited from many discussions with Costas Busch and Eric Ruppert, and they have been inspirations to me.

The administrative staff at Brown was very friendly, and I thank them all, especially Dawn Nicholas, for their congeniality and efficiency. The technical staff were superb. And thanks to Tony for the daily Portuguese lessons.

My friends have made my stay at Brown a memorable one, and I thank them all for that. Nari Subramanian welcomed me to Brown University and introduced me to the habit of consuming enormous amounts of coffee (while getting ready to release RG 2003). The unlimited energy and enthusiasm of Nars Ramachandran sometimes rubbed off on me too. Bose, Pramod, Song and Aris were wonderful company during those much needed breaks while working at the CIT. My roommates, Ram, Ashok, Vasanth, Kishore and Mulla have been fun to live with, and best of all, they all cooked well. I'll always remember the good times spent with my friends at Brown and elsewhere, Shishir, Dhume, Shashikiran, Suresh Babu, Nidhiya, Chandra, Niranjan, Ramesh, Balaji, Kedar, Suvarna, Sanjukta, Vaze, Bhakti, Rama, Priya, Santosh, Ashwin, Harsh, Mokshay, Bhaskar, Sowmya, Pooja, Pallavi, Lucia, Nesime, Selim, Swarup, Liz, Olga, Ioannis, Lijuan, Dave, Keith, Stu, Don Carney, Tim Rowley, Robert Velisar, Navin and Yasemin.

My sister Gayathri, brother-in-law Subbu, and Chandrashekhar doddappa have been my well wishers always, and their support and the memory of the fun times together keeps me going. I'm grateful to my brother-in-law, Ajit Agrawal, for timely advice and encouragement during the initial stages of my Ph.D. My music teacher, Warren Senders, has been a constant source of inspiration, and I'm deeply indebted to him for his confidence and his investment in me.

My parents, Sharada and Nagabhushana, have always believed in me and have had to endure the pain of being separated from their children for such long times. Their love and confidence, and the love of my late grandmother, Smt Jayalakshamma, have always been very important to me. Finally, I thank my wife, Kirthi, for her patience, cheerfulness, and the promise of exciting times ahead.

Contents

⋆ Parts of this thesis were previously published as [39, 38, 37]

List of Figures

Chapter 1

Introduction and Motivation

Distributed data structures can be manipulated concurrently by many different processors. Their design and analysis lies at the heart of distributed computing.

The need for distributed data structures arises naturally in a distributed system. For example, the internet needs routing tables, which are distributed data structures maintained and used by all the computers on the network. A peer-to-peer system such as Gnutella [28] requires a distributed search data structure, which lets computers look up locations of files matching a specific search criterion. A multiprocessor system needs load balancing data structures which distribute computational tasks evenly among all the processors.

In this thesis, we study a fundamental distributed data structure, the distributed queue. The interface to the distributed queue is simple; it supports a single method, *queue(·)*. Method *queue(o)* extends the queue by adding operation o to the end of the queue, and informing o's predecessor of its (newly formed) successor. The *queue(·)* method can of course be concurrently invoked at different processors. The queue thus consists of all operations queued by all processors, and each operation in the queue only knows the identity of its successor (possibly an operation queued by another processor). [1]

This is a distributed queue in two senses. First, just like any other distributed data structure, it can be manipulated concurrently by many processors. In addition, the state of the queue is distributed. No single processor needs to know what the whole queue looks like. Each element of the queue knows the identity of its successor (if a successor exists), as in a singly linked list where each element has a pointer to the next one. Sometimes, it might be useful for an element to know its predecessor too. But in general, each element of the queue needs only a local view of the queue.

Distributed queuing lies at the heart of many seemingly diverse distributed coordination problems, such as distributed directories, ordered multicast, and the general distributed *fetch-and-ϕ* data structuring problem. Given an efficient solution to distributed queuing, we show how to solve all these problems efficiently.

[1] When we say that an an operation in the queue "knows" x, we mean that the processor which issued the operation has been informed about x.

This thesis introduces distributed queuing, presents new analyses of (existing and new) distributed queuing algorithms, studies its fault tolerance, and demonstrates the use of distributed queuing in various applications.

Distributed Queuing vs Distributed Counting: To understand distributed queuing better, we contrast it with a related problem, *distributed counting.* Distributed counting is the problem of obtaining a parallel and distributed implementation of a counter. A counter is an object that holds an integer value, and supports the *fetch-and-increment* operation, that increments the counter's value and returns its previous value. When k processors concurrently increment a counter whose current value is c, they are returned successive integral values in the range $[c, c+k-1]$ and the value of the counter is increased to $c+k$. Distributed counting has been researched extensively, and many counting techniques have been invented, such as Counting Networks [8], Combining Trees [30, 31] and Combining Funnels [60].

We can think of distributed counting as follows. Processors issue operations which are arranged into a total order. Each processor gets back *the rank of its operation* in the total order. In distributed queuing, as in counting, processors issue operations which are arranged into a total order. However, each processor gets back *the identity of its successor* in the total order. Thus, in distributed counting, the return value is a number, while in queuing, the return value is an operation identifier. Knowing an operation's rank in the queue tells us little about the identity of its successor, and vice versa. Thus, distributed queuing and counting cannot be trivially reduced to each other. In chapter 5, we show that for underlying topologies with high diameters, such as the list and the mesh, distributed queuing is an inherently easier problem than distributed counting. In chapter 6, we show how distributed counting can be reduced to distributed queuing, with additional overhead.

So far, we have informally described distributed queuing. We now give a precise definition.

Our Definition of Distributed Queuing: First, a few words about our model of a distributed system. This thesis deals with message-passing distributed systems. A message-passing distributed system consists of processors which communicate by sending messages over communication links. Very often, this is modeled by an undirected graph whose nodes are processors and whose edges are first-in-first-out communication links. A distributed data structure encapsulates some data and supports one or more *operations* which can be invoked at any processor and perhaps simultaneously at different processors.

Formally, a distributed queue encapsulates an ordered pair (o, v), which is the operation currently at the tail of the queue; o is the operation identifier and v is the node of origin. The value of the object is initially $(\perp, w)$, where $\perp$ is a special operation identifier, indicating that the queue is empty. The distributed queue supports one operation, *queue.* When a node u issues an operation *queue*(o', v'), and the current value of the object is (o, v),

- The value of the object is changed to (o', v'), reflecting the fact that (o', v') is the new tail of the queue.

- Processor v is informed of (o', v'), which is the successor of (o, v) in the queue.

Concurrent operations are queued in some order $(o_1, v_1), (o_2, v_2), (o_3, v_3) \ldots$ Node v_1 is informed of its successor (o_2, v_2), v_2 is informed of (o_3, v_3) and so on.

1.1 Applications

Solutions to a number of distributed coordination problems can be built on top of distributed queuing.

Ordered Multicast: In multicast, the same set of messages is delivered to a group of recipients. A multicast protocol is *ordered* (or *totally ordered*) if it ensures that the set of messages multicast to a group of recipients is delivered in the same order at each destination, even when those messages are generated concurrently from several sources. One approach to this problem is to solve it in two stages: (1)the ordering stage, where messages are ordered into a total order (2)the multicast stage, where messages are multicast (using unordered but reliable multicast) with the ordering information piggy-backed. Recipients might *receive* them in different orders at different nodes, but *deliver* them to the applications in the same order at all nodes.

The first stage, that of ordering the messages, can be solved using distributed queuing. We will elaborate on ordered multicast in chapter 5.

Distributed Object Management: Consider the problem of managing a mobile object in a network. Nodes might need access to this object, which can reside at only one node at a time. When a node requests the object, the object's current owner is informed of the request and the object is transferred to the requesting node. If there were concurrent requests for the object, then they would be queued in some order, and the object passed from node to successor down the queue. This application has been studied in the context of distributed directories and in distributed mutual exclusion.

The hard part in a solution to managing such an object is queuing concurrent requests and knowing who's next in the queue, so that the object can be passed down the queue. This is exactly solved by distributed queuing.

The object that we pass down the queue could be a privilege, in which case this is a solution to the *distributed mutual exclusion* problem.

Concurrent and Distributed Data Structures: We show how to use distributed queuing as a building block in constructing general purpose parallel and distributed data structures (i.e. implement an arbitrary *fetch-and-ϕ* object). For example, distributed counting, additions, multiplications, are all special cases of the *fetch-and-ϕ* operation.

This is perhaps the first such construction which is both distributed and parallel at the same time, and has non-trivial theoretical properties. This construction is elaborated in chapter 6.

1.2 Contributions of Thesis

- Our first contribution lies in identifying queuing as a fundamental distributed coordination problem and formulating its definition. The problem has been implicitly studied earlier, especially in the context of *distributed mutual exclusion* [55, 51, 64, 56]. Queuing can be used to solve distributed mutual exclusion, but we show that it is of much wider applicability.

- **Analysis of the Arrow queuing protocol:**

  We focus mainly on one protocol for distributed queuing, the *Arrow protocol.* The Arrow protocol was invented by Kerry Raymond [55] in the context of distributed mutual exclusion, and is based on path reversal on a network spanning tree. The protocol is simple and lightweight, and has been observed to perform well in practice, especially under high contention to the queue. To date, however, there has been no systematic analysis of this protocol's concurrent performance or its inherent scalability. We present a competitive analysis of the Arrow protocol, showing that its performance is never far from the idealized "optimal" protocol. The highlight is our analysis of the performance of the protocol under concurrency, when a number of operations are being queued at the same time. This analysis yields a surprising connection to the nearest-neighbor traveling salesperson heuristic on a tree metric. Our analysis improves on the analyses by Demmer and Herlihy [21], and by Peleg and Reshef [52] who study the sequential case only.

  We analyze other properties of the protocol such as fairness and linearizability. We present experiments analyzing the behavior of the Arrow protocol, which show that it outperforms conventional centralized queuing protocols. The study of the Arrow protocol is the subject of chapter 3.

- **New queuing protocols:**

  We introduce a new distributed queuing protocol, based on combining trees [29, 71], that we call the combining queuing protocol. We present a competitive analysis of the combining queuing protocol.

- **Ordered multicast using queuing:**

  We present a new approach to *ordered multicast* using distributed queuing. Our queuing based solution to ordered multicast seems more efficient than previous solutions based on distributed counting.

  Ordered multicast (sometimes called totally ordered multicast) is the problem of ensuring that messages multicast to a group of nodes are delivered in the same order everywhere, even when these messages are generated concurrently at several sources. This need for ordering arises in many applications. For example, consider *push-based* cache coherence: a distributed object is cached at multiple nodes in a network. Updates are sent to these cache copies using multicast. If several nodes issue concurrent updates, all the copies must apply those updates in the same order. More generally, ordered multicast is useful in any kind of middleware where the order of events originating at different nodes in a distributed system must be reflected consistently among the nodes "listening" to such events. Examples include publish-subscribe systems, which have been the subject of much recent interest.

We believe that any truly scalable multicast protocol must be *anonymous*, in the sense that a node performing a multicast need not be aware of the identities of the recipients. IP multicast [19], SRM [26], and RMTP [46] are all anonymous in this sense. If all nodes (or even some nodes) must know every group member, then entering or leaving the group requires a global reconfiguration, which is clearly not scalable beyond local area networks. Our approach to ordered multicast thus differs in fundamental ways from that employed by systems based on notions of *virtual synchrony* [4, 15, 49, 65], in which each node knows the exact group membership. In these systems, each node entering or leaving the group provokes a global reconfiguration (called a "view change"). These systems trade fault-tolerance for scalability, while we do the opposite.

We believe that the queuing-based solution to ordered multicast, coupled with appropriate fault tolerance mechanisms, can lead to truly scalable and anonymous ordered multicast, which has not been achieved so far. Ordered multicast, and related problems such as multiple group ordering, are discussed in detail in chapter 5.

- **Fault tolerance:**

  In a distributed system, there is an additional dimension to every problem: fault-tolerance. How does the distributed data structure react when a fault occurs? For example, what happens when a message between processors is lost, or a processor crashes?

  Many distributed coordination problems, such as the well known *consensus problem*, have been known to be impossible to solve in the presence of processor failures in a purely asynchronous distributed system, where there are no upper bounds on message latencies and on the time between two consecutive steps taken by a processor (Fischer, Lynch and Paterson [25]). We show that similarly, there cannot be a distributed queuing data structure which can tolerate processor failures. Thus, instead of fault *tolerance*, we focus mainly on fault *recovery*. How does the system recover from a failure to a legal state? How long does it take for the system to do so?

  We describe how to make the Arrow protocol *self-stabilizing*. Informally, a protocol is self-stabilizing if it can recover from an arbitrary fault "on its own". That is, starting from an arbitrary (possibly faulty) initial global state, the protocol executes actions which bring it to a "legal" protocol state, if it is not already in one. Such a guarantee would be useful for applications which can tolerate transient inconsistencies when the protocol is stabilizing into a legal state. Significantly, our self-stabilizing Arrow protocol is *scalable*, and each node only needs to interact with its immediate neighbors.

We prove the correctness of our self-stabilizing protocol and show that the protocol stabilizes quickly. Fault tolerance is the subject of chapter 4.

- **Building general distributed data structures using queuing:**

  We show how to use distributed queuing as a building block in constructing general purpose parallel and distributed data structures (i.e. implement a distributed *fetch-and-ϕ* object). Given a sequential implementation of a data structure, we show how to construct a parallel and distributed implementation of the same. This is perhaps the first such construction that is truly distributed and parallel at the same time.

  A competing technique is *Combining trees* (Goodman, Vernon and Woest [29], Tew, Tzeng and Lawrie [71])). However, Combining trees lack the important property of *locality.* For example, if the same processor issued multiple operations one after the other, and no other processor was operating on the data structure at the same time, each operation might still incur long operation latencies, and generate unnecessary message traffic.

  Our algorithm, based on the Arrow queuing protocol, exhibits locality. If the same (active) processor operated on the object repeatedly, and no other processor issued an operation in the meanwhile, the object would automatically move to the active processor, and no more message traffic would be generated after that. At the same time, if there are a number of processor issuing operations concurrently, all these operations proceed in parallel.

  Our algorithm builds on the combination of two ideas: distributed queuing and pointer jumping. Distributed queuing ensures that there is no central bottleneck in the algorithm (i.e. that it is distributed). Pointer jumping brings in the additional parallelism required for making concurrent operations proceed in parallel. Queuing based distributed *fetch-and-ϕ* is described in detail in chapter 6.

- **Complexities of concurrent queuing and concurrent counting:**

  In Chapter 5, we show that on high diameter graphs, such as the list and mesh, concurrent queuing is an inherently easier problem than concurrent counting. We would like to emphasize that our result *is about the problems* of queuing and counting and *is not about specific solutions* to them.

  This is the first result that we know of which studies these problems from a latency and message complexity viewpoint rather than from a fault tolerance viewpoint.

In summary, this thesis highlights the problem of distributed queuing, and presents the first systematic study of the problem.

Roadmap of this thesis: Chapter 2 contains a description of a few queuing protocols: the centralized protocol, the Arrow protocol, the Combining protocol and a protocol based on Counting Networks. Chapter 3 contains analyses of the Arrow protocol and the Combining protocol. Chapter 4 contains a discussion of the fault tolerance of distributed queuing in general, and of the Arrow protocol in particular. Chapter 5 describes the application of queuing to ordered multicast, related problems, and contains the comparisons of the concurrent complexities of queuing and counting. Chapter 6 contains a description of the new *fetch-and-ϕ* distributed data structure based on distributed queuing. We conclude with Chapter 7, which contains a a recap and a list of open research problems.

Chapter 2

Distributed Queuing Protocols

In this chapter, we describe a few distributed queuing protocols. We will start with a simple protocol, the Centralized queuing protocol and then go on to describe the Arrow protocol, the Combining Tree Protocol and the Queuing protocol using counting networks.

Our system is a network of processors, which communicate by sending messages to each other. We make no assumptions on the topology of the underlying network. We model the network as a graph $G = (V, E)$. Each vertex $v \in V$ is a processor. Each edge $(u, v) \in E$ is a bi-directional FIFO communication link between processors u and v. Thus, a message between two processors might have to travel a number of edges. We assume that we are provided with the ability to route messages from one processor to another via the shortest path between them on the graph. The algorithms that we describe can be readily adapted to the shared memory model.

2.1 Centralized Queuing

Let us examine what is necessary for implementing queuing. An operation has to join the queue and inform its predecessor about itself. In the following discussion, when we speak of the *tail* of the queue, we mean the node which issued the last operation in the current state of the queue. For any queuing protocol, the important piece of information required to join the queue is the location of the tail of the queue.

Perhaps the simplest implementation would be to use a central node to maintain the location of the current tail. This central node c is known to all the nodes in the network. Node c has one variable t, which is the location of the current tail of the queue.

When a node v wants join the queue, it sends a message to c. Upon receiving this message, c sends a message to t (the location of the current tail of the queue), indicating that v is next in the queue. At the same time, c assigns v to t, thereby noting that the tail of the queue is now at v. At this point, v's operation has been queued and v is the current tail. A new operation arriving at c will be queued behind v's latest operation. Concurrent operations arriving at c are executed in an arbitrary order. This is a very simple protocol, yet might be effective in many cases. But this protocol has two drawbacks.

- **Centralized bottleneck:** The central node has to handle every queuing operation, and this might overwhelm the central node if there are many concurrent operations, and lead to long latencies.
- **No locality:** Suppose the same node v issued a number of operations in succession, and no other node issued an operation to join the queue in the meanwhile. Each of v's operations would travel to the central node and back, though its predecessor operation is at v itself! This is clearly inefficient, and the ideal protocol would not produce any network traffic in such a case.

2.2 The Arrow Protocol

We now describe the Arrow protocol, which overcomes both the drawbacks (locality and centralized bottleneck) of the centralized protocol mentioned above. The Arrow Protocol was invented by Kerry Raymond in 1989 [55] in the context of distributed mutual exclusion. In 1998, Demmer and Herlihy [21], applied the protocol to the problem of distributed directories.

2.2.1 Description

We first give an informal description of the protocol. The basic idea in the algorithm is *path reversal* on a spanning tree of the network.

The key issue, as in the centralized protocol, is to find the tail of the queue in the network. Initially, some node in the system contains the tail of the queue. The Arrow protocol assigns a direction to every edge of the spanning tree. [1] The directions are assigned such that following these directed edges on the spanning tree will lead us to the node which has the tail. Thus, each node in the network knows the direction along the tree in which the tail lies, but not the location of tail itself. When a node v issues a queuing operation, the operation follows the directed edges on the tree to the current tail, flipping the edges on the way. Once v's operation reaches the tail, it informs the node that its successor is at v, and its queuing is complete. The directions on the edges have now been modified in such a way that following the directed edges from any node leads to the new tail. A new queuing operation from any node will be queued behind v's operation. Thus, by changing the directions of edges on the path from v to the old tail, we have changed the global state to reflect the fact that v contains the new tail of the queue.

A formal description follows. Recall that we model the network as a graph $G = (V, E)$. We are given a spanning tree T of G. Figure 2.1 shows an example graph G and a spanning tree T of G. Each node v in T has a "pointer", denoted by $link(v)$. For each v, $link(v)$ is either a neighboring node in the spanning tree, or v itself. Each node v also has an attribute $id(v)$, which is the (unique)

[1]This is not entirely accurate, but helps the intuition. See the formal description for the actual protocol.

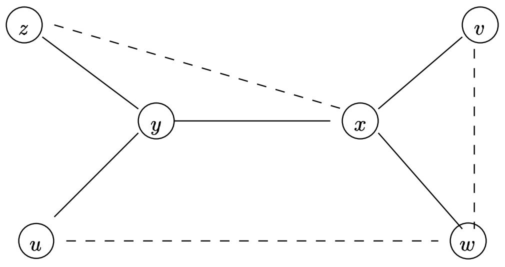


Figure 2.1: Arrow Queuing
The underlying network graph G, and a spanning tree T. The solid edges belong to the spanning tree.

identifier of the previous queuing operation issued by v. If v has not issued any queuing operations so far, then $id(v)$ is $\perp$.

A node v is a *sink* if $link(v) = v$. The *link* pointers are initialized so that following the pointers from any node leads us to a unique sink. Informally, except for the unique sink node, a node knows only in which "direction" the sink lies. Figure 2.2 shows the initial state of the arrows.

When a node v initiates a queuing operation whose *id* is a, it executes the following sequence of steps atomically (i.e no other operation at v overlaps in time with this sequence of steps):

- sets $id(v)$ to a
- sends a $queue(a)$ message to $u_1 = link(v)$ and
- sets $link(v)$ to v

When node u_i receives a $queue(a)$ message from node u_{i-1}, it executes a *path reversal*, elaborated in the following atomic sequence of steps. Let $u_{i+1} = link(u_i)$.

- It "flips" u_i's link i.e., it sets $link(u_i)$ to u_{i-1}.
- If $u_{i+1} \neq u_i$, then u_i forwards the message $queue(a)$ to u_{i+1}.
- If $u_{i+1} = u_i$, then operation a's queuing operation is complete, and a is queued behind $id(u_i)$.

If there is a need to know the predecessor of an operation too, then node v, the originator of a, can be sent a message (which can travel directly on the shortest path on G) saying that a has been queued behind $id(u_i)$.

Thus far, our explanation might convince the reader that the protocol works correctly if operations are issued sequentially (the next queuing operation is issued only after the previous one has been queued). But the interesting thing about the protocol is that it works just as well even in the case of concurrent queuing operations. An example execution with two concurrent queuing operations is illustrated in Figures 2.2 through 2.6.

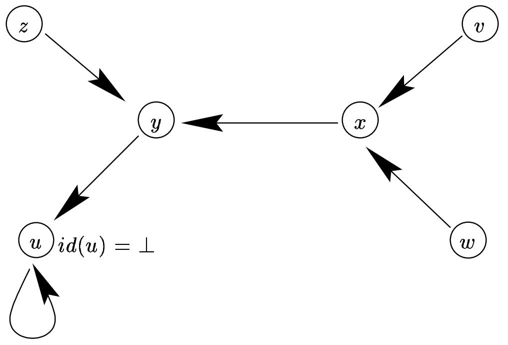


Figure 2.2: Arrow Queuing: Initial System State.
The arrows point to neighbors in the spanning tree, leading to a unique “sink”.

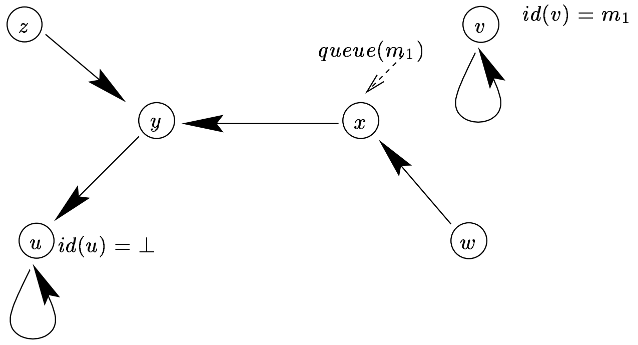


Figure 2.3: Arrow Queuing Step 1
Node v sends message m_1, on its way to x.

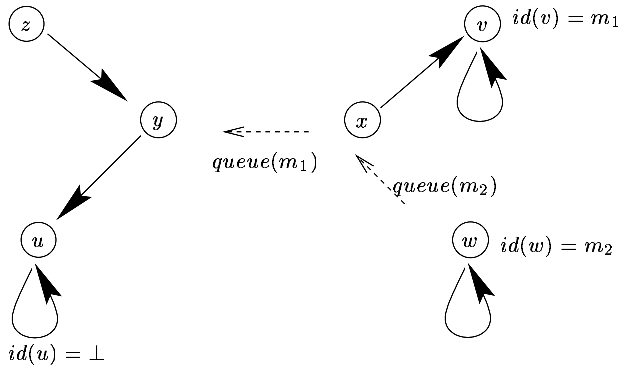


Figure 2.4: Arrow Queuing Step 2
Node w sends m_2, now on its way to x.

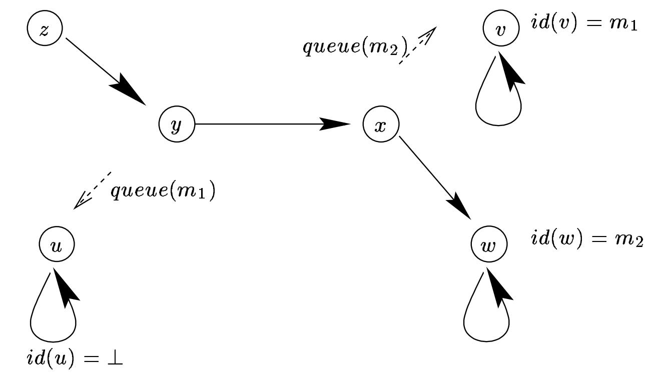


Figure 2.5: Arrow Queuing Step 3
m_1 and m_2 follow the arrows, flipping their directions along the way. Note that m_2 has been "deflected" towards v.

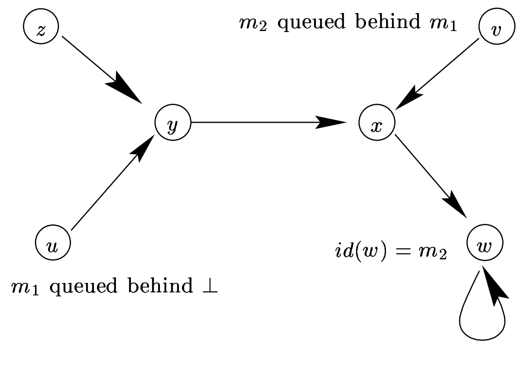


Figure 2.6: Arrow Queuing Step 4
Both m_1 and m_2 find their predecessors and are queued concurrently.

2.2.2 Correctness

The correctness of the protocol has been discussed by Herlihy and Demmer [21]. They show that every $queue(\cdot)$ message eventually finds a sink, and that the maximal number of links traversed by such a message is at most the diameter of the tree T.

2.3 The Combining Tree

Combining trees, described by Gottlieb et. al. [30, 31] are a well known technique for implementing distributed data structures. The idea is simple: the processors are organized in a rooted tree, where the root holds the current value of the data structure. Operations travel up the tree towards the root. If two operations meet en-route, then they are combined into a single operation, which proceeds further up the tree. Combining reduces the traffic at the root and the nodes close to the root. An experimental analysis of combining trees and other distributed data structures for shared memory systems is presented by Herlihy, Lim and Shavit in [36].

Let's first look at how combining trees can be used for *distributed counting* [2]. When two operations combine at a node, a single "aggregated" operation goes upwards, and the node is locked. Once this operation is performed at the root, the result comes down the tree and is split among the two operations waiting at the node. Thus, the latency of an operation is the distance to the root and back. More operations can be combined similarly. For further details, we refer the reader to [30, 31].

However, queuing using combining trees can be performed with lower latency than counting using combining trees. We now explain how. Recall that in queuing, an operation need only find the identity of its predecessor, and inform it. Thus, when two operations combine into a single operation, it has already been decided that they are going to be performed in consecutive order at the root. The later of the two knows its predecessor right away and can travel to its predecessor on the shortest path on the graph. The earlier operation proceeds up the tree, and might combine with other operations on the way, before it reaches the root. The later operation did not have to wait for the round trip to the root, and we can think of this as "short circuiting" the general combining tree algorithm. A detailed example is shown in Figures 2.7 to 2.10.

Let T be the (rooted) spanning tree chosen for combining. To execute the operation $queue(a)$, a node v sends a $queue(a, a)$ message to its parent in T. If two messages $queue(a, b)$ and $queue(c, d)$ meet at a non-root node y, b is designated to be the result of c's queuing operation, and the node which issued the operation b is informed of its successor. The two messages are combined into a single message $queue(a, d)$, and the combined message is sent to y's parent (more than two messages can be combined in a similar way). When a $queue(a, b)$ message arrives at the root, the root's value $val(r)$ is designated to be the result of a's queuing operation, and $val(r)$ is set to b.

[2] In distributed counting, each operation gets back a sequence number, as opposed to distributed queuing, where the return value is the identity of its predecessor.

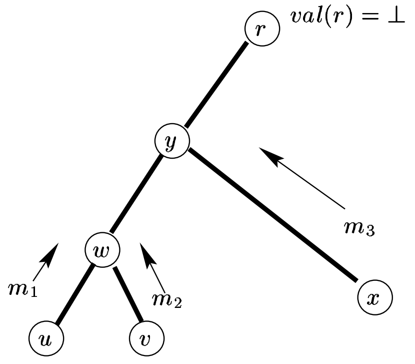


Figure 2.7: Combining Queuing Step 1
Messages m_1, m_2 and m_3 start their journey towards the root.

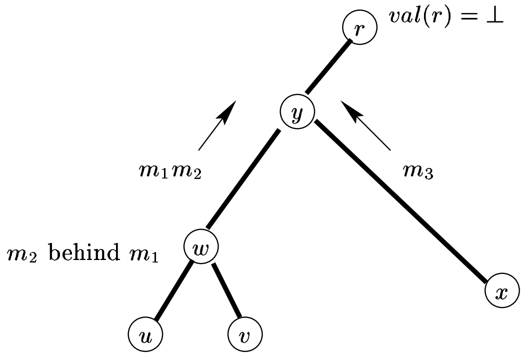


Figure 2.8: Combining Queuing Step 2
Messages m_1 and m_2 combine. m_2 is designated as m_1's successor, and u is informed about m_2.

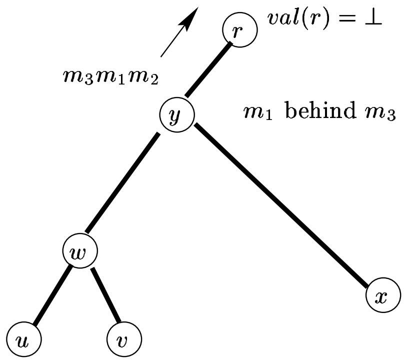


Figure 2.9: Combining Queuing Step 3
Messages m_1m_2 and m_3 combine. m_1 is designated as m_3's successor, and x is informed of m_1. $m_3m_1m_2$ continues towards the root.

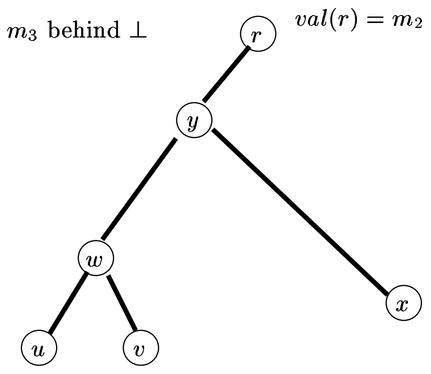


Figure 2.10: Combining Queuing Step 4
Message $m_3 m_1 m_2$ reaches the root. m_3 is the successor of $\perp$ (i.e. m_3 is the first message in the total order). $val(r)$ is now m_3.

2.4 Queuing Using a Counting Network

A counting network, like a sorting network [18] is a directed acyclic graph, whose nodes are simple computing elements called *balancers*, and whose edges are called *wires*.

Counting Networks were introduced by Aspnes, Herlihy and Shavit [8] as a solution to distributed counting, and have been well studied in the distributed computing community. We will only focus on how to adapt counting networks to solve distributed queuing.

A width w counting network has w input wires and w output wires. Each token (operation) enters on one of the network's input wires, traverses a sequence of balancers, and exits on an output wire with a sequence number. If k tokens enter the counting network (either one after the other, or in parallel), then they will exit with the sequence numbers $1, 2 \ldots k$. This orders the tokens into a total order.

But this does not quite solve the queuing problem, since queuing requires that each token inform its predecessor. A token exiting with sequence number i would know that its predecessor has sequence number $i-1$, but doesn't know its identity. Many different types of counting networks, including the bitonic counting network, are described by Herlihy, Aspnes and Shavit in [8].

Counting networks have the following property: *if a token a exits from the output wire i, then its predecessor will (eventually) exit on the output wire $i-1$ (subtraction done modulo w).* If every output wire keeps a record of all the tokens that ever exited from it, then a can go to wire $i-1$ and find out the identity of its predecessor. Because of timing anomalies, it is possible that token a exited the counting network, but its predecessor still hasn't. In such a case, a has to wait for its predecessor token at wire $i-1$ to complete its operation.

Thus, our implementation of queuing using counting networks has two problems:

1. *Unbounded buffers at the output wires.* Space for a token can be freed at an output wire if its successor has found it, but in an asynchronous system, we would still need unbounded space.

2. *Waiting.* A token might have to wait at the output wire of its predecessor if the predecessor hasn't exited yet.

These make counting networks unattractive for queuing. In addition, if the underlying network topology is not a complete graph, there is the additional problem of embedding a counting network on such a graph.

2.5 Other Protocols

There are other protocols for distributed queuing based on path reversal, invented in the context of distributed mutual exclusion. One such algorithm is due to Naimi, Trehel and Arnold (NTA) [51]. The NTA algorithm also uses path reversal and has superficial similarity to the Arrow protocol. However, their path reversal algorithm differs from the Arrow protocol in the following significant ways:

- The NTA algorithm assumes that the underlying network topology is a completely connected graph, while the Arrow protocol does not.
- The Arrow protocol uses a fixed spanning tree, and the pointers can point only to a neighbor in the spanning tree. However, the NTA algorithm does not use a fixed spanning tree, and a node's pointer can point to any node in the graph.

  Thus, in the Arrow protocol, a queuing operation never travels farther than the diameter of the tree, while in NTA, it could travel through every node in the graph. Under certain assumptions on the probability distribution of operations at nodes, Naimi, Trehel and Arnold [51] show that an expected $O(\log n)$ messages are required, where n is the number of nodes in the graph.

Chapter 3

Analysis of the Arrow Protocol

In the previous chapter, we presented a few queuing protocols. In this chapter, we turn to their analyses. The main contribution of this chapter is a novel theoretical analysis of the Arrow protocol under concurrency.

The Arrow protocol is a distributed algorithm, which has been observed to perform very well in practice, especially under high concurrency (we present supporting experimental data in the next section). We aim to analyze this behavior theoretically.

A distributed algorithm, such as the Arrow protocol, has to deal with two sources of uncertainty. One is that it lacks global information, and has to make decisions based on incomplete information about the global state. Another source of uncertainty is the lack of information about the future. For example, a queuing algorithm does not know when or where future queuing operations will be issued, or which other nodes are starting queuing operations simultaneously.

In order to analyze the performance of such a distributed online algorithm, we use a tool called *competitive analysis*. The idea in competitive analysis is to compare the performance of an online algorithm, such as the Arrow protocol, which works with incomplete information, with the performance of the optimal *off-line* algorithm. The off-line algorithm has full knowledge of the global system state and of the future, and no online algorithm could ever do better. The worst case ratio between the performance of the online and the off-line algorithms is the *competitive ratio* of the online algorithm. The idea of competitive analysis of online algorithms was introduced by Sleator and Tarjan [61]. Competitive analysis of distributed online algorithms was studied by Ajtai, Aspnes, Dwork and Waarts [3, 9].

In this chapter, we present a competitive analysis of the Arrow protocol and the Combining protocol.[1] Previous analyses of the Arrow protocol (Demmer and Herlihy [21], Peleg and Reshef [52]) have concentrated on the sequential case, where there is only one operation in progress at a time.

However, the more interesting problem is the protocol's behavior in the presence of concurrent

[1]Most of the chapter focuses on the analysis of the Arrow protocol, since our analysis shows that its performance is better than that of the Combining protocol.

operations. Since the observed behavior under concurrency has been good, a theoretical analysis would give us greater insight into the working of the protocol. In this chapter, we present such an analysis.

This chapter contains the following results.

- We present the results of our experiments on the performance of the Arrow protocol and the Centralized protocol under concurrency.
- We present the first analysis of the Arrow protocol under concurrency, suggesting why the protocol performs well in the presence of concurrent queuing requests.

  We first study the one-shot scenario, where all the operations start at the same time. We show that in this case, the Arrow protocol is competitive to the optimal. We derive almost matching upper and lower bounds on the competitive ratio.
- We generalize the problem to the semi-synchronous model and the long lived case (where all requests do not have to start at the same time), and present our results in this model.
- We study other properties of the Arrow protocol, in particular showing that while the protocol is not linearizable, it has a useful fairness property.
- We present a competitive analysis of the Combining protocol, showing that the competitive ratio of the Combining protocol for the one-shot case is much worse than that of the Arrow protocol.

A preliminary version (specifically, the analysis of the Arrow protocol for the one-shot case) of these results appears in [38].

This chapter is mainly concerned with a theoretical analysis of the Arrow protocol. Before getting to the theoretical analysis, we will describe our experiments with the Arrow protocol.

3.1 Experiments with the Arrow protocol

In this section, we describe our experiments with the Arrow protocol and the Centralized queuing protocol. The aim of these experiments was to test the performance of the protocols under concurrency. The hardware used was an IBM SP2 distributed memory machine, with up to 76 processors. All programs were written using MPI (Message Passing Interface) [50, 33].

The Arrow protocol needs a spanning tree of the network. Since the message latency between any pair of nodes in the SP2 machine was roughly the same, we could treat the network as a complete graph with all edges having the same weight. Our spanning tree was a perfectly balanced binary tree ($\log_2 n$ depth for n nodes).

We measured the time taken for 100,000 queuing operations per processor. In our experiments, a processor issued the next queuing operation immediately after it learnt that its previous operation had completed. The results of running this experiment for different numbers of processors are shown in Figure 3.1.

Note that the total number of queuing operations performed by the system (100,000 per processor) increases linearly with the size of the system. The best we can hope for is that the latency remains constant with increasing system size, which would happen under ideal parallelism. No queuing algorithm can achieve this, since coordination between different processors is necessary (recall that all the operations are being ordered into a single queue).

In this light, it is surprising that the latency of the Arrow protocol initially increases sublinearly and then remains nearly constant with increasing size of the system. In contrast, the Centralized protocol shows near linear slowdown with increasing system size. This is to be expected, since the central processor has to handle a linearly increasing number of messages with increasing system size. It is a tribute to the designers of the IBM SP2 that the system showed a graceful degradation (only a linear slowdown) under such loads. When we tried the same experiments on a network of SUN workstations, the centralized protocol could not scale beyond 20 processors, while the Arrow protocol scaled easily.

Figure 3.2 shows the average number of hops (interprocessor messages) per queuing operation for the Arrow protocol. The average number of hops is less than one because most of the queuing operations find their predecessors locally, and thus generate zero interprocessor messages.

These experiments suggest that the Arrow protocol performs very well under concurrency: operations have low latencies, neighboring operations in the queue are very close on the tree.

Another set of experiments on the Arrow protocol were conducted by Herlihy and Warres [40]. They used Arrow protocol for *distributed directories*, and compared it against a home-based (centralized) protocol. They observed that the Arrow protocol outperformed the home-based over a range of system sizes.

In the next few sections, we give a theoretical explanation of why the Arrow protocol might perform so well under concurrency.

3.2 Arrow Protocol: One-shot Problem

In this section, we give a novel competitive analysis of the Arrow distributed queuing protocol under concurrent access. We analyze the combined latency of r simultaneous operations, and derive a competitive ratio of $s \cdot \log r$, where s is the stretch of a preselected spanning tree in the network. [2]

Our analysis employs a *greedy* characterization of the way the Arrow protocol orders concurrent operations, and yields a new lower bound on the quality of the tour produced by the nearest neighbor heuristic for the Traveling Salesperson Problem.

A queuing algorithm has many options for handling concurrent operations. For example, when presented with simultaneous queuing operations from nodes a, b and c, where a and b are near one

[2]Informally, the *stretch* of a spanning tree of a graph is the overhead of routing on the spanning tree versus routing on the graph.

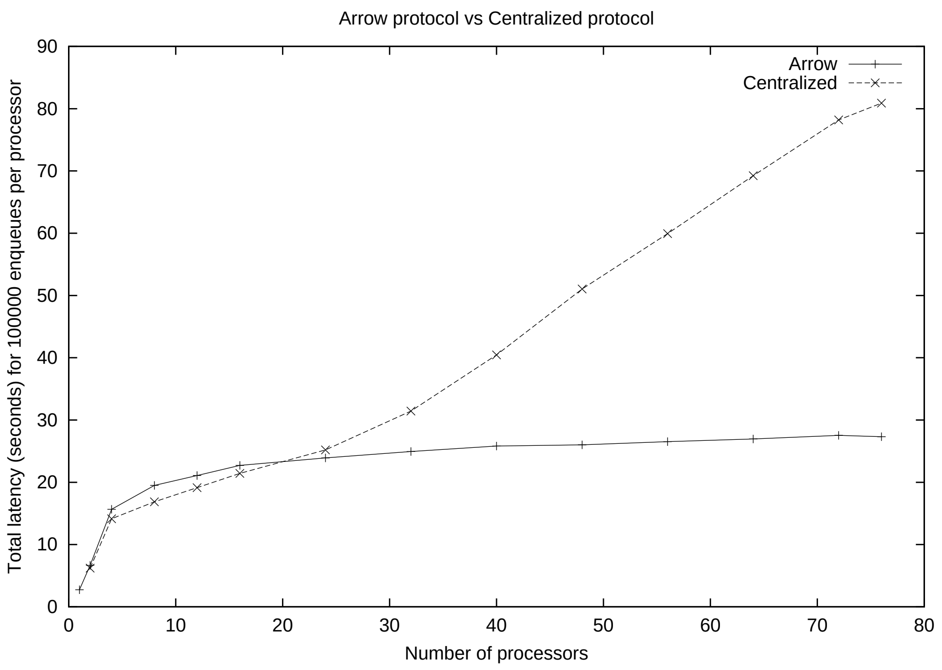


Figure 3.1: Latency of Arrow protocol vs Centralized protocol
The experiments were performed using MPI on an IBM SP2 distributed memory machine, with up to 76 processors. The above graph shows the latency for performing 100,000 enqueues per processor, as a function of the number of processors.

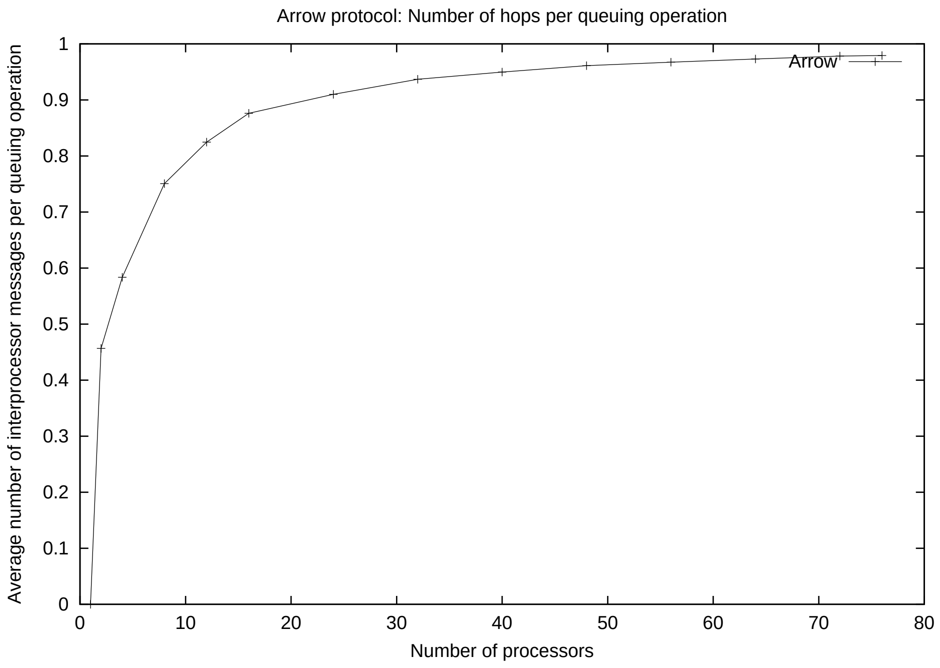


Figure 3.2: Average number of hops per queuing operation for the Arrow protocol.
The experiments were performed using MPI on an IBM SP2 distributed memory machine, with up to 76 processors. The above graph shows the average number of hops (interprocessor messages) per enqueue operation for the Arrow protocol. Note that the number of hops is also the distance between the queuing operation and its predecessor on the tree.

another but c is far, it makes sense to avoid ordering c between a and b. More generally, r concurrent operations can be ordered in any of $r!$ ways, and depending on the origins of the operations, some orderings may be much more efficient than others. The performance of the queuing algorithm thus depends crucially on how it orders concurrent requests.

3.2.1 Prior Work

Demmer and Herlihy [21] gave a competitive analysis of the protocol under sequential access. Sequential access assumes that the Arrow protocol and the adversary would queue operations in the same order. Their analysis is as follows.

Recall that the Arrow protocol executes on a spanning tree T of the network graph G. Arrow would send a message from a node to its predecessor on the shortest (and only) path on the spanning tree while the optimal protocol could do so on the shortest path on the graph. The *stretch* s of a spanning tree T of graph G is the worst-case ratio between the lengths of the shortest paths linking two vertices in T and the same vertices in G. Thus the Arrow protocol has a worst case competitive ratio of s in the sequential case.

While Demmer and Herlihy suggested using a minimum spanning tree, Peleg and Reshef [53] showed that the protocol overhead is minimized by using a *minimum communication spanning tree* [42]. They further showed that if the probability distribution of the origin of the next queuing operation is known in advance, then it is possible to find a tree whose expected communication overhead is 1.5.

However, the above analyses do not apply to concurrent access because they assume that the adversary and the protocol order operations in the same way. The adversary is not allowed to gain by ordering concurrent (or nearly concurrent) operations differently. In our analysis, we strengthen the adversary to have the power of ordering operations differently from the online algorithm.

3.2.2 Our Results

We study the following *one-shot* instance of the problem for concurrent access. At time zero, r nodes simultaneously issue operations to join the queue (and no further operations occur). The *latency* of an operation is the time needed for it to inform its predecessor of its identity. The cost of a protocol is the *sum* of all operations' latencies. The question is: how far away from the optimal can the cost of the Arrow protocol be?

Our main result is as follows: *for r concurrent operations, the competitive ratio of the Arrow protocol for the one-shot case is bounded by $6s \cdot \log r'$, where s is the stretch of the spanning tree (T) chosen by the protocol and r' is the number of operations which are at the leaves of T (thus $r' \leq r$).*

This competitive ratio does not depend on the size of the network, and depends only logarithmically on the degree of concurrency.

Informally, the adversary could win over a distributed protocol like the Arrow protocol in two ways:

- It could communicate over the graph, while the Arrow protocol communicates over a tree, since it needs to synchronize the operations. This leads to the factor of stretch in the competitive ratio.

- The adversary could select the queuing order of the operations in an optimal way, while the distributed protocol, with local information could be sub-optimal in its ordering. For the arrow protocol, this results in a factor of $\log r'$ in the competitive ratio.

  We now clarify the definition of r'. r' is the number of leaves of the spanning tree T which have issued operations at time zero. Hence, r' is always smaller than r, the total number of operations. Sometimes, it can be much smaller. For example, if the tree T were a list, then the number of leaves is two, and r' cannot exceed two; but the number of operations r could be much larger.

This bound has practical implications since there are no hidden constants in the competitive ratio.

Our analysis employs a novel *greedy* characterization of how Arrow orders concurrent operations, and yields an intriguing connection with the nearest neighbor heuristic for the Traveling Salesperson Problem (TSP). The nearest neighbor heuristic for the TSP constructs a tour by starting at any vertex and visiting the closest unvisited neighbor next. Rosenkrantz, Stearns, and Lewis [57] have shown that the nearest neighbor heuristic is a $\log r$ approximation algorithm for TSP on a complete graph with r nodes whose edge weights satisfy the triangle inequality.

We show new upper and lower bounds on the quality of the tour produced by the nearest neighbor TSP algorithm on a *tree metric.* A complete graph $G = (V, E)$ is a tree metric if there exists a tree S such that the nodes of S is a superset of the nodes of G, and the distance between two nodes in G equals the length of the shortest path between them on S. Informally, graph G can be embedded in tree S.

Since a tree metric obeys the triangle inequality, the result in [57] implies a $\log r$ upper bound for the nearest neighbor algorithm on a tree metric. We show the following results specific to the tree metric:

1. **Improved upper bound:** The nearest neighbor algorithm is a $O(\log r')$ approximation algorithm, where r' is the smaller of the following: (1)the number of leaves of the tree S (2)r, the number of nodes in G.

   In the worst case, r' could be as much as r, but it could be much smaller. For example, this implies that the nearest neighbor algorithm is a constant factor approximation for a list.

2. **New lower bound:** There exist tree metrics on which the nearest neighbor algorithm could be off by as much as a factor of $\Omega(\log r / \log\log r)$ from the optimal. It follows that a graph having the stronger tree metric property, instead of simply the triangle inequality property does not substantially improve the behavior of the nearest neighbor TSP heuristic.

The remaining part of this section is organized as follows: we first explain our model of computation and analysis. We then present our analysis of the Arrow protocol, which has two parts: an upper bound on the cost of the Arrow protocol, followed by a lower bound on the cost of the optimal. This is followed by an almost matching lower bound on the competitive ratio of the protocol, and a discussion of special cases where the competitive analysis can be strengthened.

3.2.3 Synchronous Model

We model the network as a graph $G = (V, E)$ where V is the set of nodes (processors), and E is the set of edges representing reliable FIFO communication links between processors. We assume a synchronous model of computation, where the latency of a communication link is predictable. We denote the latency of edge e by $w(e)$. We later discuss the semi-synchronous model, where the edge weights are not predictable, but lie within some prespecified range.

The Arrow protocol runs on a spanning tree T of this graph, which we can choose. We denote the length of the shortest path between nodes u and v on G by $d_G(u, v)$ and the length of the shortest path between them on T by $d_T(u, v)$. The *stretch* of the tree T with respect to G is defined as $s = \max_{u,v \in V} d_T(u, v)/d_G(u, v)$. This ratio measures how far from optimal a path through the spanning tree T can be.

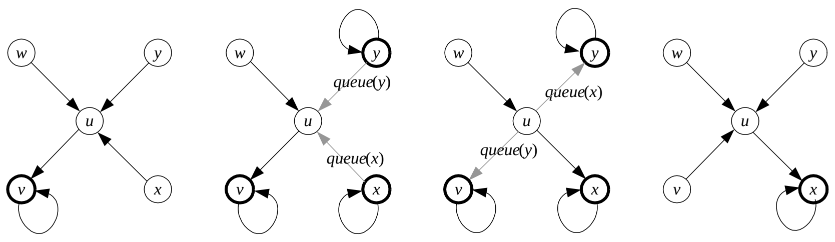
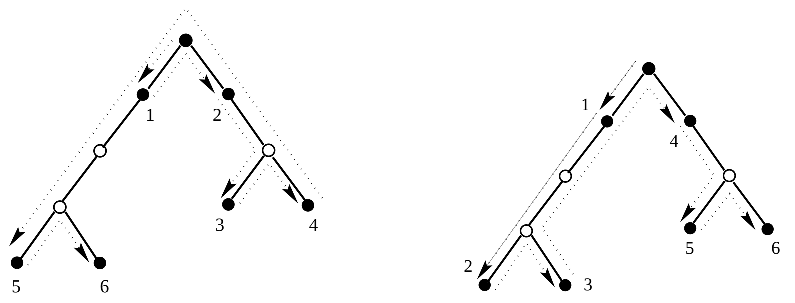


Figure 3.3: Arrow protocol. Concurrent *queue* messages.
Initially node v is selected the tail of the queue. Nodes x and y both choose to join the queue simultaneously. Message $queue(y)$ arrives at node u before $queue(x)$. Finally, $queue(x)$ and $queue(y)$ find their respective predecessors y and v in the queue, and x is the new tail of the queue.

A $queue()$ message arriving at a node is processed immediately, and simultaneously arriving messages are processed in an arbitrary order. This assumes that we can process up to d messages in a time step at a node, where d is the node's degree. In practice, the time needed to service a message is small when compared to communication latencies, and since the degree of a node in the spanning tree is typically small, a node cannot receive many simultaneous messages. For a simple example with concurrent $queue()$ operations, see Figure 3.3.

In the *one-shot* problem, at time zero, a subset $R \subseteq V$ of nodes request to join the queue, with $r = |R|$. Let $l_A(v)$ be the latency of node v's operation. The cost of the protocol is $L^A = \sum_{v \in R} l_A(v)$. We compare L^A to a lower bound L^*, which is the cost of an optimal protocol (with



(a) A greedy walk on a tree. (b) An Euler tour traversal of the same tree.

Figure 3.4: A greedy (nearest neighbor) walk on a tree vs an Euler tour
Vertices belonging to R are marked solid; every edge has weight 1. The numbers at the vertices indicate the order in which they are visited. The traversals start at the root, which is the topmost vertex.

global knowledge and incurring no synchronization costs). The competitive ratio of Arrow is defined to be $\max_{R \subseteq V}\{L^A/L^*\}$.

3.2.4 Competitive Analysis

In this section, we present an upper bound for L^A and a lower bound for L^*. We first analyze L^A.

Suppose the operations are queued in the order $v_1, v_2, \ldots, v_r$. That is, *root* (the initial tail of the queue) is the predecessor of v_1, which precedes v_2 and so on. Note that the order of execution is not necessarily the real time order of completion. All the queuing is done in parallel, so v_2 might learn about its successor v_3 before v_1 learns about v_2.

As shown in [21] a *queue*() message travels along a simple path on the tree until it finds the predecessor, and never waits. Hence $l_A(v_1) = d_T(v_1, root)$, and $l_A(v_i) = d_T(v_i, v_{i-1})$, for $i > 1$. We have:

$$L^A = d_T(v_1, root) + \sum_{i=2}^{r} d_T(v_i, v_{i-1}). \tag{3.1}$$

We now give a simple characterization of the order $v_1 \ldots v_r$. We define a *greedy walk* as follows. A greedy walk on tree T over vertex set $R = \{u_1, \ldots, u_r\}$ visits all vertices in R as follows. It starts at the root of the tree. It then visits the closest unvisited vertex in R, and keeps doing so until all vertices in R have been visited. In other words, the vertices of R are visited in the order $v_1, \ldots, v_r$ with

$$d_T(root, v_1) = \min_{v \in R} d_T(root, v) \tag{3.2}$$

$$d_T(v_i, v_{i+1}) = \min_{v \in R} d_T(v_i, v) \text{ with } v \notin \{v_1, \ldots, v_i\} \tag{3.3}$$

An example of a greedy walk is shown in Figure 3.4. An ordering of vertices is *greedy* if there is a greedy walk that produces the same ordering. Denote the length of a greedy walk on tree T over vertex set R by $greedy(T, R)$.

Theorem 1 *The ordering of the Arrow protocol is greedy: the ordering satisfies equations 3.2 and 3.3.*

Proof: We first prove Equation 3.2. Let C be the set of all the closest operations to the root, and let d be the distance between them and the root at time 0. At time 0, operations in C start traveling towards the root, since the tree is initialized with arrows pointing towards the root. If two (or more) of these $queue()$ operations meet at a node, then one continues towards the root and the others are deflected. Therefore at least one of the operations in C arrives at the root at time d. Since no operation outside C can reach the root in time d or less, we have $v_1 \in C$, as in equation 3.2.

We now prove Equation 3.3. Denote the root by v_0. Consider another starting configuration of the distributed system, where the tree is initialized with v_1 as the root and there is no operation at v_1. Call this configuration F_1 and the original one (with v_0 as the root) F_0.

Lemma 2 *No operation but v_1 will be able to distinguish between the configurations F_0 and F_1 during execution.*

Proof: Refer to Figure 3.5. The only difference between F_0 and F_1 is that all arrows on the path between v_0 and v_1 are in the opposite direction. Assume, for the sake of contradiction, that there is a $queue(v_i)$ with $i > 1$ that is able to see an arrow pointing towards v_0 before $queue(v_1)$ changes it. Then $queue(v_i)$ must reach a node u (u between v_0 and v_1) before $queue(v_1)$. If so, $queue(v_1)$ would be deflected towards v_i, contradicting the assumption that v_1 is the first operation in the total order. ■

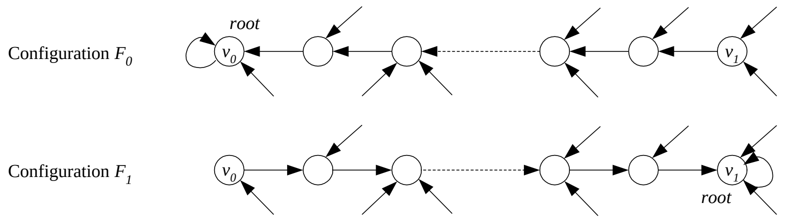


Figure 3.5: Proof of greedy (nearest neighbor) characterization of Arrow protocol
The two configurations are identical to every operation but v_1.

Lemma 2 implies that for the purpose of finding the ordering of $\{v_i | i > 1\}$ we can pretend as if we started in configuration F_1 and $queue(v_2)$ will find v_1 in the resulting execution. Applying Equation 3.2 to F_1, we find that v_2 is one of the operations closest to v_1. Inductively, we define F_i to be the configuration derived from F_0 by removing the operations at nodes $v_1, \ldots, v_i$, and making v_i the root of the tree. For the rest of the operations $\{v_j | j > i\}$, F_i is identical to F_0. Equation 3.3 follows. This concludes the proof of Theorem 1. ■

The above Theorem lets us relate L^A to the cost of traveling salesperson tours. Let $TSP(G)$ denote the cost of the optimal traveling salesperson tour on graph G.

Theorem 3 *Let G_R^T denote the complete graph whose vertex set is R (plus the root) and in which the distance between vertices u and v is equal to $d_T(u,v)$. Let r' denote the number of vertices of R which are at the leaves of the spanning tree T. Then $greedy(T,R) \leq 3\log r' \cdot TSP(G_R^T)$.*

Proof: It is easy to show a ratio of $\log r$ on the right hand side, since it follows directly from Theorem 1 of Rosenkrantz, Stearns, and Lewis [57]. They show that the nearest neighbor heuristic is at most a factor logarithmic in the number of nodes worse than an optimal traveling salesperson tour on a graph satisfying certain conditions. A greedy walk costs exactly as much as the nearest neighbor traveling salesperson tour without returning to the root. We note that the graph G_R^T satisfies the three preconditions of the Theorem, that is, $d(u,v) = d(v,u)$, $d(u,v) \geq 0$, and the triangle inequality $d(u,v) + d(v,w) \geq d(u,w)$.

We now show how to get an improved bound of $\log r'$ on the right hand side. We assume that every leaf of the spanning tree has a operation on it (otherwise, we can get a smaller spanning tree by "trimming" such leaves, and the number of leaves of the spanning tree does not increase).

We denote the sum of the weights of all edges of tree T by $w(T)$. The greedy walk starts at the root of the tree and visits all the operations. Let the operations be visited in the order $v_1 v_2 \ldots v_r$. The cost of the greedy walk is

$$greedy(T,R) = d_T(root, v_1) + d_T(v_1, v_2) + \ldots d_T(v_{r-1}, v_r)$$

We divide this sum into two parts, based on whether the starting node is an internal node or a leaf.

First part,

$$s_1 = \sum_{v_i \text{ is an internal node}} d_T(v_i, v_{i+1})$$

Second part,

$$s_2 = \sum_{v_i \text{ is a leaf}} d_T(v_i, v_{i+1})$$

We bound s_1 by $w(T)$ using a charging argument. If v_i is an internal node, then all the operations in the subtree rooted at v_i are still unvisited when v_i is first visited. We charge the amount $d_T(v_i, v_{i+1})$ to a path in this subtree as follows.

If v_{i+1} is in this subtree, then we charge $d_T(v_i, v_{i+1})$ to the path from v_i to v_{i+1}. If not, then we know that $d_T(v_i, v_{i+1})$ is less than $d_T(v_i, v_j)$ for any vertex v_j in the subtree. In this case, we charge $d_T(v_i, v_{i+1})$ to the path v_i to v_k where v_k is the next vertex in the subtree to be visited by the greedy walk. It can be verified that this charging scheme charges each edge in the tree not more than once. Thus, s_1, which is the total cost of all the movements of the greedy walk starting from internal nodes is not more than $w(T)$.

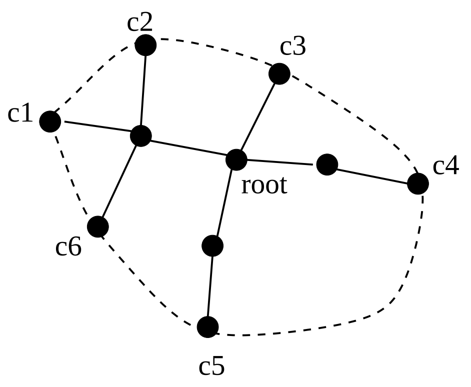


Figure 3.6: Bounding the cost of a nearest neighbor TSP on a tree.
Visualize the leaves of the tree as lying on a circle. We divide the leaves into sets of two vertices each, $\{c_1, c_2\}$, $\{c_3, c_4\}$ and $\{c_5, c_6\}$.

To bound s_2, we use a different method. We visualize all the leaves of the tree as lying on a circle, as shown in Figure 3.6. Denote this order by $c_1 \ldots c_{r'}$. Divide the r' leaves into $r'/2$ pairs, pair i consisting of vertices $\{c_{2i-1}, c_{2i}\}$. Let S be a set of $r'/2$ vertices, consisting of one vertex from each group, chosen as follows: in group i, among c_{2i} and c_{2i-1}, choose that vertex which is visited earlier than the other.

We show that the sum of path lengths from all the vertices in S is not more than $2w(T)$, which is the cost of an Euler path traversal around the edges of the tree. The proof is as follows. Suppose in pair i, c_{2i} was visited earlier than c_{2i-1}. The length of the path from c_{2i} to its successor in the greedy walk is not more than $d_T(c_{2i}, c_{2i-1})$. Adding this over all $r'/2$ pairs, the sum of all path lengths starting from vertices in S is not more than $d_T(c_1, c_2) + d_T(c_3, c_4) + \ldots + d_T(c_{r'-1}, c_{r'})$, which is less than or equal to $2w(T)$ (an Euler path traversal of the tree).

Thus, we have shown that the sum of the path lengths of the paths starting from half the leaves is $2w(T)$. We are now left with $r'/2$ elements. By a similar argument, we can show that the sum of path lengths of half of the remaining vertices is not more than $2w(T)$. Iterating the process, we arrive at a $2w(T) \log r'$ bound for s_2.

From the above discussion, we have

$$greedy(T, R) = s_1 + s_2 \leq w(T) + 2w(T) \log r' \leq 3w(T) \log r' \tag{3.4}$$

Since the optimal TSP on the tree has to visit at least all the leaves of the tree, it has to traverse every edge at least once.

$$TSP(G_R^T) \geq w(T) \tag{3.5}$$

The theorem follows from equations 3.4 and 3.5. ■

Let T_R denote the smallest subtree of T containing all the vertices in R and the root. Note that the optimal TSP on G_R^T corresponds to an Euler tour traversal of T_R, as shown in Figure 3.4, and then returning back to the root.

Theorems 1 and 3 immediately lead to the following corollary.

Corollary 4 $L^A \leq 3 \log r' \cdot TSP(G_R^T)$

Theorem 5 *Let G_R be a complete graph on R (plus the root) with the weight of the edge between two nodes u and v equal to $d_G(u,v)$. Then $L^* \geq TSP(G_R)/2$.*

Proof: Suppose $R = \{u_1, \ldots, u_r\}$, and the optimal algorithm queues the operations in the order $u_1, \ldots, u_r$. u_i's queuing is complete only when u_{i-1} learns of the identity of u_i. For this, a message has to travel from node u_i to u_{i-1}, and this takes time at least $d_G(u_i, u_{i-1})$ (the distance between the two nodes on the graph). Thus, the latency of u_i's operation is at least $d_G(u_{i-1}, u_i)$ (and the latency of u_1's operation is at least $d_G(root, u_1)$).

Thus, the sum of the latencies of the optimal algorithm is at least

$$L^* \geq d_G(root, u_1) + \sum_{i=2}^{r} d_G(u_i, u_{i-1})$$

When finally returning to the root, we have a valid (but not necessarily optimal) TSP tour with cost $C \leq L^* + d_G(u_r, root)$. The graph G_R satisfies the triangle inequality, since its edge weights are lengths of shortest paths between the vertices on G. Thus, $d_G(u_r, root) \leq L^*$, and therefore $TSP(G_R) \leq C \leq 2L^*$. ■

We are now ready to prove the main theorem, which states that the protocol is competitive with the optimal.

Theorem 6 $L^A/L^* \leq 6s \cdot \log r'$.

Proof: The edge between vertices u and v in G_R^T has weight $d_T(u,v)$ while the corresponding edge in G_R has weight $d_G(u,v)$. The edge on G_R^T cannot be longer than the corresponding edge in G_R by more than a factor of s. The same ratio carries over to the length of the optimal TSP tours and we have $TSP(G_R^T) \leq s \cdot TSP(G_R)$. With Corollary 4 and Theorem 5 we get

$$\begin{aligned} L^A &\leq 3 \log r' \cdot TSP(G_R^T) \\ &\leq 3 \log r' \cdot s \cdot TSP(G_R) \\ &\leq 6 \log r' \cdot s \cdot L^*. \end{aligned}$$

■

In the remainder of this section we show that our analysis is (almost) tight. We will construct a tree, along with a set of requesting nodes where the greedy (nearest neighbor) walk is off by $\Omega(\log r / \log\log r)$ from optimal. It follows that having the tree metric does not help (much) over the more general triangle inequality metric. To the best of our knowledge, this is a new result in the area of TSP heuristics as well.

Theorem 7 *There exists a tree T and a set of requesting nodes R such that $L^A = \Omega(\log r/\log\log r)L^*$, where $r = |R|$.*

Proof: The tree T consists of a long "trunk"with many "branches" of varying lengths on it as shown in the Figure 3.7. Each branch is a single edge; one end of the edge is on the trunk and the other end is a leaf with an operation on it. A branch of length 0 is an operation on the trunk. The root is at one end of the trunk and the other end is denoted by $\top$. Our convention is that we move *right* to get from the *root* to $\top$. The distance between two branches is the distance between the endpoints of the branches which are lying on the trunk. Similarly, the distance between a vertex v on the trunk and a branch e is the distance between v and the endpoint of e that lies on the trunk.

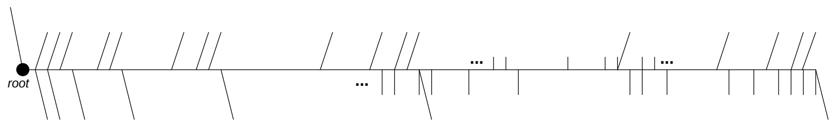


Figure 3.7: Bad example for nearest neighbor TSP on a tree. The Tree T.

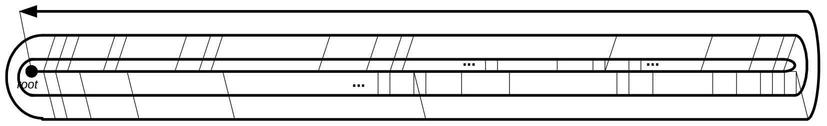


Figure 3.8: Greedy traversal of Tree T

The idea is as follows: by careful placement of branches on the trunk, we will make the greedy walk traverse the length of the trunk many times, as shown in Figure 3.8. The trunk contributes a significant fraction of the weight of the tree, and we get the length of the greedy walk to be super-linear in the size of the tree and hence the length of the Euler tour. The details follow.

Let the length of the trunk be w. Let $k = \log w/\log\log w$ rounded down to the nearest odd number. We have $k + 1$ sets of branches, $B_0 \ldots B_k$. Each branch in B_i is of length i. Thus operations in B_0 are on the trunk, while those in B_1 are at distance 1 from the trunk, and so on. When the context is clear, we use B_i to refer to the set of operations that lie on the branches in B_i.

There is only one branch in B_k and this is at the root. Once we have placed all the branches in B_j, we place those in B_{j-1} as follows.

Suppose j was odd. Let $e_1, e_2 \in B_j$ be two consecutive branches in B_j starting at vertices u_1 and u_2 from the trunk respectively (i.e there are no branches in B_j with endpoints between u_1 and u_2). Suppose u_1 is closer to the root than u_2. Let l be the least integer such that $2^{l+1} \geq d_T(u_1, u_2)$. We place branches in B_{j-1} at vertices between u_1 and u_2 at geometrically increasing distances $1, 3 \ldots 2^l - 1$ from u_1. This is shown in Figure 3.9. If the farthest branch from the root in B_j starts at u, then we place branches in B_{j-1} between u and $\top$ at distances $1, 3 \ldots 2^l - 1$ from u until $2^{l+1} > d(u, \top)$. We also place a branch in B_{j-1} at $\top$.

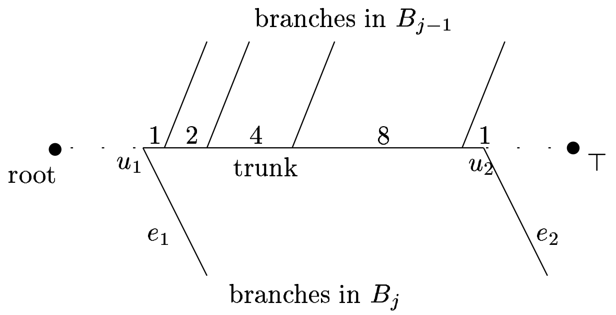


Figure 3.9: Placement of branches on the trunk of Tree T.
Location of branches in B_{j-1} between two branches in B_j.

Suppose that j was even. The construction is similar to the above, but the role of $\top$ and the root are interchanged. In other words, if $e_1, e_2 \in B_j$ are two consecutive branches in B_j starting at vertices u_1 and u_2 from the trunk respectively and suppose u_1 is closer to the root than u_2. Let l be the least integer such that $2^{l+1} \geq d_T(u_1, u_2)$. We place branches in B_{j-1} at vertices between u_1 and u_2 at distances $1, 3 \ldots 2^l - 1$ from u_2 (not u_1). Similarly, we place branches in B_{j-1} between the branch in B_j that is closest to the root and the root, and we place a branch at the root.

Lemma 8 *For any vertex on the trunk, one of the closest operations in the set of branches $\cup_{i \geq c} B_i$ is in B_c.*

Proof: Let x be a vertex on the trunk. We show that the distance to the closest operation in B_i is less than or equal to the distance to the closest operation in B_{i+1}. Suppose r_p was the closest operation in B_{i+1}, whose branch starts from the trunk at vertex p.

Consider the following case: p is to the left of x (or $p = x$) and i is even (or zero). There is a branch in B_i to the right of p at distance 1 from p (if p is $\top$, then there is a branch in B_i at $\top$). The operation on this branch is closer to x than r_p, or at the same distance.

The other cases, p on the left of x and i odd, and the analogous cases for p to the right of x can be checked similarly. ■

Theorem 9 *The following is a greedy walk on T. Start from the root. Visit all operations in order of the branch size (i.e., all operations in B_0 first, followed by those in B_1 and so on until B_k). If i is even (or zero), visit the operations in B_i in order of increasing distance from the root. If i is odd, then visit them in order of increasing distance from $\top$.*

Proof: We show that visiting all the operations in B_0 in order of increasing distance from the root is a prefix of a greedy walk. This portion of the walk ends at $\top$. Then the proof follows, since after that we can treat $\top$ as the root, and it is a similar situation.

Clearly, the first operation visited is the closest operation in B_0 because of Lemma 8. Suppose we are at vertex x on the trunk. All the operations in B_0 to the left of x have been visited. None to the right have been visited yet. Let c_0 be the closest operation in B_0 that is to the right of x. We will show that c_0 is one of the closest unvisited operations. Going to c_0 next would be greedy, and this way we visit all the operations in B_0 in order of increasing distance from the root and reach $\top$.

To prove that c_0 is one of the closest unvisited operations, we first show that c_0 is not further away from x than the closest operation in B_1 (say c_1). By Lemma 8, one of the closest operations to x in the set $\{\cup_{i\geq 1} B_i\}$ is in B_1, and the proof follows.

We now show that c_0 is not further away from x than c_1. Note that the operation c_1 could be to the right or left of x, but c_0 is to the right of x. Suppose c_1 was to the right of x. Recall that in our construction, to the right of every branch in B_1, there is a branch in B_0 at a distance of 1 (or if c_1 is $\top$, then there's a branch in B_0 at $\top$). The operation on this branch in B_0 is at least as close to x as c_1. Now suppose that c_1 was to the left of x. There are branches in B_0 at distances $1, 3 \ldots 2^l - 1$ from c_1 and it can be seen that the closest operation in this set of branches to the right of x is not further away from x than c_1. ■

We now compute the length of the greedy walk. It traverses the trunk $k+1$ times and the branches at least once each. Thus, the length of the greedy walk is $L^A \geq (k+1)w + w_B$ where w_B is the sum of the weights of all the branches. The Euler tour traverses each edge of the tree at most twice, thus $L^* \leq 2(w + w_B)$.

Now, we give an upper bound for w_B. The number of branches in B_i is given by the following Lemma.

Lemma 10 *We have* $|B_{k-i}| < 2\log^i w$.

Proof: By induction: $|B_k| = 1$, and $|B_{k-1}| = \log w + 1$. With the induction assumption we have $|B_{k-i+1}| < 2\log^{i-1} w$. The maximum distance d between two branches of B_{k-i+1} is less than w, therefore $|B_{k-i}| < 2\log^{i-1} w \cdot \log d < 2\log^{i-1} w \cdot \log w = 2\log^i w$. ■

Thus, w_B is bounded by,

$$w_B = \sum_{i=0}^{k} |B_{k-i}| \cdot (k-i) < 2\sum_{i=0}^{k} (k-i) \cdot \log^i w < 2\frac{\log^{k+1} w}{(\log w - 1)^2}$$

We use $k = \log w / \log\log w$ and get $w_B < 2w$. Thus,

$$L^A/L^* \geq \frac{(k+1)w + w_B}{2(w + w_B)} \geq \frac{(k+1)w}{6w} = \Omega(\log w / \log\log w)$$

This concludes the proof of Theorem 7. ■

3.2.5 Special Graphs

Some common graphs do not need the extra $\log r$ overhead. If the network itself is a tree, and there are enough concurrent operations, then we can apply a different analysis to strengthen our result.

Theorem 11 *Let G be a tree with constant degree c. Suppose all operations R come from a subtree T of G. Let h denote the height of the subtree, and let $w(e) = 1$ for each edge e in T. Then the cost* *of the Arrow protocol is bounded by c^h. If the number of concurrent operations is significant, i.e. $r = |R| = \Omega(c^h)$, then the Arrow protocol is asymptotically optimal, that is, $L^A = O(L^*)$.*

Proof: Let e be an edge in T. Denote the number of times the greedy walk traverses edge e by $t(e)$. The distance between an edge e and vertex v is defined to be the distance between v and the adjacent vertex of e that is closest to v. Let edge e be at level l (at a distance of l from the *root*, $0 \leq l \leq h-1$).

Let u_i be the operation that is visited right after the ith traversal of edge e, with $i = 1, \ldots, t(e)$. Note that node u_i with odd (even) index i has $d_T(\mathit{root}, u_i) > (\leq)\ d_T(\mathit{root}, e)$. Moreover, $d_T(u_i, u_{i+2}) \geq d_T(u_i, u_{i+1})$. Since $d_T(u_i, u_{i+1}) = d_T(u_i, e) + 1 + d_T(e, u_{i+1})$, and $d_T(u_i, u_{i+2}) = d_T(u_i, e) + d_T(e, u_{i+2})$, we conclude that

$$\begin{aligned} d_T(u_{i+2}, e) &= d_T(u_i, u_{i+2}) - d_T(u_i, e) \\ &\geq d_T(u_i, u_{i+1}) - d_T(u_i, e) \\ &= d_T(e, u_{i+1}) + 1. \end{aligned}$$

With $d_T(e, u_1) \geq 0$ we have $d_T(e, u_i) \geq i - 1$. Let k be the greatest odd number which is not greater than $t(e)$. In other words, $t(e) \leq k + 1$. Using $d_T(e, u_k) \geq k - 1$ we get $t(e) \leq d_T(e, u_k) + 2$. Let h be the height of the tree. Since the tree has height h, $d_T(e, u_k) \leq h - l - 1$, thus $t(e) \leq h - l + 1$.

Because the tree has constant degree c, we know that the number of edges at level l is bounded by c^l. The greedy walk is bounded by the sum of traversals of all the edges in the tree, that is

$$\mathit{greedy}(T, R) \leq \sum_{e \in T} t(e) \leq \sum_{l=0}^{h-1} c^l (h - l + 1) = O(c^h).$$

Applying Theorem 1 we immediately get $L^A = O(c^h)$. On the other hand, the optimal TSP tour has to visit at least r nodes, and since no two operations are at the same node, we have $L^* \geq r$. The second claim follows. ■

3.3 Generalizations

In this section, we analyze the Arrow protocol in more general scenarios. We generalize the problem in two directions.

- The first direction is to move from the synchronous to the semi-synchronous model. How does the Arrow protocol fare if the communication delay of a link is not fixed, but could be any number in a prespecified range?
- The second direction generalizes the problem from the one-shot case (where all operations start at the same time) to the long lived case, where operations could start at arbitrary times.

3.3.1 Semi-Synchronous Model

So far, we have assumed that the message latency of each edge is fixed, and that the protocol knows these latencies. It is easy to see that the result holds in the case where the latencies are fixed, but the protocol did not know them, since the Arrow protocol does not use the knowledge of the latencies anywhere during its execution.

A natural question is: *what about the case when the message latencies are not fixed?* We discuss this question (but do not provide a complete solution) in the next few paragraphs.

Suppose the message latency of edge e was any number in the range $[l(e), h(e)]$. Consider an execution E of the protocol for the one-shot case, from time zero until the time when all the operations have been queued. In such an execution, different messages might experience different latencies for traversing the same edge. Similar to G_R and G_R^T (which were defined in Section 3.2.4, we define the graph G_R^E, whose nodes are the operations and the distance between nodes u and v equals *the time that a message took to go from u to v in execution E.* Note that this graph is specific to the particular execution E. Some of these distances are determined by E, since a message actually took a path between those two vertices during the execution, while some of the distances are unknown (because no message ever took that path).

Observation 1 *In execution E, the ordering of the operations is determined by a greedy walk on G_R^E.*

If G_R^E were symmetric, and obeyed the triangle inequality, we can show that the nearest neighbor heuristic is a $\log n$ approximation to the optimal TSP on G_R^E, and obtain a result similar to that of Theorem 6 (with the definition of stretch suitably modified). In the semi-synchronous case, this is not true in general.

3.3.2 Long Lived Case

We have so far analyzed the one-shot instance of the Arrow protocol where all the operations start at the same time. This yields a competitive ratio of $s \cdot \log r$. The other end of the spectrum is the *sequential* case, where the operations are so far apart in time that the Arrow protocol and an optimal protocol would choose the same queuing order. For the sequential case, the competitive ratio for Arrow is just the stretch of the tree, s. This leads to the following natural question: *what about the general* long lived *case, which is neither sequential nor one-shot?*

One might expect the answer to be somewhere between s and $s \log r$, but we do not have a complete answer yet. We now present a partial solution.

Completion Time vs Latency

We first explain our modeling of the problem. As in the one-shot case, we assume the synchronous model, where all communication links have predictable latencies, and the edge weight is the latency of the communication link. We are given a schedule of operations, $r_i = (v_i, t_i)$, for $i = 0, \ldots, k$, where v_i is the node of origin and t_i the time at which the operation is initiated. We adopt the following convention: $r_0 = (v_0, 0)$ is a special operation designated to be the first in queue (note that r_0 starts at time zero). For example, in the Arrow protocol, r_0 would reside at the root of the initial state of the tree.

The *completion time* of an operation is the time at which it informs its predecessor in the queue. The completion time of operation r is denoted by $c(r)$. The *latency* of an operation r, denoted by $l(r)$, is the difference between the completion time and the time of origin. If $r = (v, t)$ then $l(r) = c(r) - t$.

We are interested in two metrics:

- **Metric 1:** Sum of latencies
$$\sum_{i=1}^{k} l(r_i)$$
- **Metric 2:** Sum of completion times
$$\sum_{i=1}^{k} c(r_i)$$

Note that the two differ by a constant, the sum of the starting times of all the operations. Also note that for the one-shot case, the two metrics are the same, since all operations start at time zero. Our aim is to analyze the Arrow protocol for the long lived case using competitive analysis.

Our Results

We have been able to analyze the Arrow protocol w.r.t. metric 2 (sum of the completion times), with results similar to the one-shot case, but this does not translate into a competitive analysis of metric 1 (sum of latencies).

Our analysis of the sum of completion times involves two parts: (1)an upper bound on the sum of the completion times of the Arrow protocol, and (2)a lower bound on the sum of completion times of any protocol. We present the lower bound first.

Lower bound on Optimal

As in the one-shot case, we can formulate a lower bound on the cost of the optimal as the cost of a traveling salesperson tour on a graph. We will begin by proving a lower bound on the latency of an operation.

Lemma 12 *If operation $r_i = (v_i, t_i)$ is queued behind operation $r_j = (t_j, v_j)$, then the following is a lower bound on r_i's completion time:* $\max(t_j, t_i + d_G(v_i, v_j))$.

Proof: The earliest time that r_i can be enqueued behind r_j is the earliest time when r_i can deliver a message to r_j.

(1) The message from r_i to r_j cannot be sent before t_i, and it takes at least $d_G(v_i, v_j)$ time for the message to reach v_j. Thus $t_i + d_G(v_i, v_j)$ is a lower bound on r_i's completion time.

(2) Since operation r_j cannot be informed before t_j (when it starts), t_j is also a lower bound on r_i's completion time, if r_i is queued behind r_j.

■

To find a lower bound on the cost of the "optimal" queuing protocol, we define the directed graph $C(G)$ whose nodes are the operations $r_i = (v_i, t_i), i = 1 \ldots k$, and the (directed) distance from node r_i to r_j is $\max(t_j, t_i + d_G(v_i, v_j))$. Any queuing protocol induces an ordering of the vertices of $C(G)$, and thus, a traveling salesperson tour of $C(G)$. Since the distance between two vertices of $C(G)$ is a lower bound on the completion time of the successor vertex (if they appear in that order), we have the following lemma.

Lemma 13 *The cost of the optimal traveling salesperson path on $C(G)$ starting from r_0 is a lower bound on the sum of completion times of any protocol.*

Upper bound on the Cost of the Arrow protocol

We now present the upper bound on the cost of the protocol. For this purpose, we define the graph $C(T)$, similar to $C(G)$. The vertices of $C(T)$ are all the operations, $\{r_i | i = 0 \ldots k\}$. The (directed) distance between r_i and r_j is defined to be $t_i + d_T(v_i, v_j)$ (recall: d_T denotes the distance between two vertices on the tree). Note that an edge from r_i and r_j in $C(T)$ is no more s times than the corresponding edge from r_i to r_j in $C(G)$.

Lemma 14 *The cost of the Arrow protocol is the cost of a nearest neighbor traveling salesperson path on $C(T)$ starting from r_0.*

Proof: We first show that the operation that gets queued behind r_0 is that operation r_i which is closest to r_0 on $C(T)$.

The operation that gets queued behind r_0 is the one which gets to v_0 the earliest. The time at which a message from operation $r_i = (v_i, t_i)$ would reach r_0 (traveling on the tree) is $t_i + d_T(v_i, v_0)$, which is also the distance between r_i and r_0. Thus, the successor of r_0 in the queue is the operation that is closest to r_0 on $C(T)$.

Suppose r_l was queued behind r_0. For the other operations, it is as if r_0 did not exist and they were executing on a tree whose arrows were initially pointing towards r_l (this is a similar argument to the one-shot case). The successor of r_l will be that operation in $\{r_i | (0 \leq i \leq k) \wedge (i \neq 0) \wedge (i \neq l)\}$ which is closest to r_l and so on.

The corresponding traveling salesperson tour starts at r_0 and visits nodes (operations) according to the rule: visit the closest unvisited operation next. The cost of the Arrow protocol is the cost of this nearest neighbor tour.

■

Thus, we have the following theorem.

Theorem 15 *For the long lived case, the competitive ratio of Arrow protocol (according to the metric: sum of completion times) is bounded by the ratio between the cost of a nearest neighbor heuristic on $C(T)$ and the cost of an optimal traveling salesperson tour on $C(G)$.*

What's missing?

There is a problem with the metric of the sum of completion times. The sum of completion times can be made arbitrarily large by making the initial start times of the operations large.

A more meaningful metric is the sum of latencies. This differs from the sum of completion times by a constant (the sum of the start times of all the operations). Unfortunately, knowing that the ratio between the sum of completion times of Arrow and that of the optimal is small does not tell us much about the ratio between the sum of latencies of Arrow and that of the optimal.

For example, suppose the sum of completion times of Arrow is $C = 1100$, the sum of start times is $S = 950$, and the ratio of the sum of completion times to optimal is 1.1 (quite small). The lower bound on the optimal sum of completion times is $1100/1.1 = 1000$. The corresponding lower bound on the optimal sum of latencies is $1000 - 950 = 50$. The sum of latencies for the protocol is $1100 - 950 = 150$. The ratio between the sum of latencies and the optimal is now $150/50 = 3$ (not so small).

Thus, the above theorem does not imply that the Arrow protocol is competitive to the optimal w.r.t. the metric of the sum of latencies. While we believe that the Arrow protocol is competitive w.r.t. the sum of latencies, we do not know how to prove it.

3.4 Other Properties of the Arrow Protocol

We turn to the following questions in this section.

- Is the protocol linearizable? If operation a completes before operation b begins, then is a guaranteed to appear earlier in the queue than b?

  The answer is that the Arrow protocol is not linearizable, as we elaborate below.

- Is the protocol fair? If operation a starts much before operation b, then is a guaranteed to appear earlier in the queue than b? What is the time difference needed to ensure this?

- We have analyzed the protocol w.r.t the metric of total latency of all the requests. What about other metrics? We discuss one other metric, the maximum latency.

3.4.1 Linearizability

Linearizability is the requirement that the ordering of operations in the queue reflects the real-time order in which they were performed. This correctness condition was introduced by Herlihy and Wing in [41]. The use of linearizable data structures simplifies the design of parallel and distributed programs.

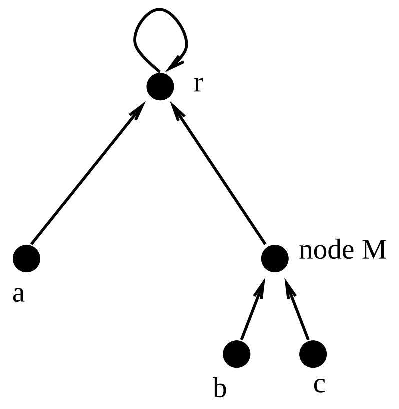


Figure 3.10: The Arrow protocol is not linearizable

A queuing protocol is linearizable if is satisfies the following condition. If operation a completes before operation b begins, then a will appear earlier in the queue than b.

Lemma 16 *The Arrow protocol is not linearizable.*

Proof: The proof is by example. Refer to Figure 3.10 for the initial state of the protocol. The current tail of the queue is located at r. We construct a scenario which is not linearizable.

Operations c and b start simultaneously (at the nodes indicated in the picture). b reaches node M before c. c proceeds towards b's originating node and is queued behind b. Operation a starts next, proceeds towards r, and gets there before b. Thus a gets queued first, then b and then c. In the above execution, operation c finished before operation a started. In spite of this, c comes later in the queue than a. This is clearly not linearizable. ■

3.4.2 Fairness

We have just proved that the Arrow protocol is not linearizable, i.e. the real-time order in which operations complete may not be consistent with the order in which they are queued.

However, we now show that the protocol has a useful *fairness* property. Informally stated, if two operations (perhaps issued at different nodes) are "sufficiently far apart in time", then the earlier operation is queued before the later operation. For example, if distributed queuing was used to manage a mobile object, this means that the operation which asked for the object earlier would get it first. What "sufficiently far apart" means depends on the distance on the tree between the nodes of origin of the operations. We state this formally in the following theorem. We assume a synchronous model, where edge latencies are fixed.

Theorem 17 *If two operations $r_1 = (v_1, t_2)$ and $r_2 = (v_2, t_2)$ satisfy the following condition: $d_T(v_1, v_2) < (t_2 - t_1)$, then r_1 (the operation issued earlier) will be queued before r_2.*

Proof: Proof by contradiction. Suppose r_2 was ordered before r_1 in the queue. Say $r = (v, t)$ was the predecessor of r_2.

Consider the directed graph of operations $C(T)$ (which is described in the long lived analysis of the Arrow protocol). The nodes of $C(T)$ are all the operations, and the (directed) distance from $r_i = (v_i, t_i)$ to $r_j = (v_j, t_j)$ is $t_i + d_T(v_i, v_j)$.

By Lemma 14, we know that the Arrow protocol orders operations according to the nearest neighbor heuristic on $C(T)$. This implies that the distance from r_2 to r on $C(T)$ is less than or equal to the distance between r_1 and r. Otherwise, the protocol would have ordered r_1 before r_2.

$$t_2 + d_T(v_2, v) \leq t_1 + d_T(v_1, v) \tag{3.6}$$
$$(t_2 - t_1) \leq d_T(v_1, v) - d_T(v_2, v) \tag{3.7}$$

But since d_T is the length of the shortest path between two vertices on the tree, we know that

$$d_T(v_1, v) \leq d_T(v_1, v_2) + d_T(v_2, v) \tag{3.8}$$
$$d_T(v_1, v) - d_T(v_2, v) \leq d_T(v_1, v_2) \tag{3.9}$$

From equations 3.7 and 3.9, we have $(t_2 - t_1) \leq d_T(v_1, v_2)$. This contradicts the initial condition $d_T(v_1, v_2) < (t_2 - t_1)$.

■

3.4.3 Other Cost Measures

We have so far analyzed the sum of latencies of all the operations (or equivalently, the average latency of an operation). Another cost measure is the delay till all the operations have been queued up, i.e. the maximum latency. In the Arrow protocol, every operation takes a simple path on the tree. Hence the delay until every operation has been queued is trivially bounded by the diameter of the tree. However, the protocol is not competitive w.r.t. the optimal under the metric of maximum latency.

Lemma 18 *The competitive ratio of the Arrow protocol w.r.t. the metric of maximum latency is $\Omega(n)$ where n is the number of concurrent operations.*

Proof: The proof is by example. The bad case is shown in Figure 3.11. Operations are present on nodes v_1 to v_n of the tree. The protocol that optimizes maximum latency would order operations as $v_1, v_2 \ldots v_n$, incurring a maximum latency of 4. Arrow might order the operations as $v_2, v_3 \ldots v_{n-1}, v_n, v_1$, which has a maximum latency of n. ■

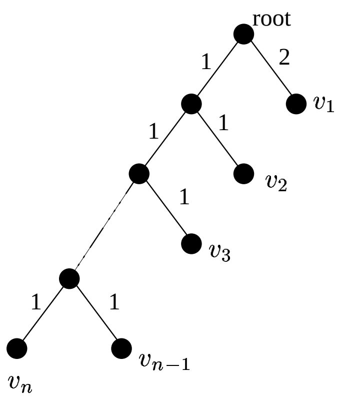


Figure 3.11: The maximum latency of Arrow can be far off from optimal. The numbers on the edges indicate the edge latencies. The same example serves as a bad case for Combining Trees.

3.5 Combining Protocol

We now analyze the Combining protocol using the same metrics as the Arrow protocol. We find that the Combining protocol is not competitive to the optimal with respect to the metric of total latency. We again focus on the one-shot scenario, where all the queuing operations start at the same time.

Theorem 19 *The competitive ratio of the Combining protocol for the one-shot case with n operations is $\Omega(s \cdot n)$, where s is the stretch of the tree chosen for combining.*

Proof: The proof again is by example. The Combining protocol is sensitive to timing, and its performance depends on two operations arriving at a node at the same time so that they can combine. It is easy for an adversary to foil it by making sure that the operations arrive a little apart in time, so that there is no combining.

We refer to Figure 3.11. We will consider the latencies of the operations located at nodes $v_2, v_3 \dots v_{n-1}$.

Since these operations are all at different distances from the root of the tree, and also from any particular node in the tree, none of them will combine. Thus, every operation travels all the way to the root, and the latency of operation v_i is at least i, for $i = 2 \dots n-1$. The total latency is at least

$$\sum_{i=2}^{i=n-1} i = n(n-1)/2 - 1$$

The optimal would order them as $v_1 \dots v_n$. Operation v_i would travel directly to its predecessor v_{i-1}, and its latency would be $d_T(v_i, v_{i-1})$. The total latency of the "optimal" is less than $4n$. Thus, the competitive ratio of the Combining protocol is at least $\frac{[n(n-1)-2]/2}{3n}$, which is $\Omega(n)$.

∎

We could improve the competitive ratio of the Combining protocol by adding waiting. In this scheme, when a operation arrives at a node, it waits for some time, hoping that other operations will combine with it, and then proceed up the tree. The competitive ratios will depend on the specific waiting strategy used.

Finally, we note that competitive analyses of distributed algorithms is an analysis of the worst case. If an algorithm has a bad competitive ratio, its actual behavior might still be good.

Chapter 4

Fault Tolerance of the Arrow Protocol

This chapter deals with the fault tolerance of distributed queuing in general, and of the Arrow protocol in particular. The following questions arise with respect to the fault tolerance of distributed queuing: (1)What kind of faults do we want to tolerate? (2)How much resources (space/time) are we willing to spend on it?

The larger the class of faults that we want to tolerate, the greater resources we will need, so the question is to find the right compromise. We first show that completely fault tolerant distributed queuing is impossible in an asynchronous distributed system in the presence of faults, irrespective of how much resources we are willing to spend.

The following defines a fault tolerant distributed queuing computation. The faults might be lost messages or process crashes. Note that we don't deal with Byzantine (malicious) failures here.

Definition 1 Fault tolerant distributed queuing: *At the end of the computation, the distributed queue contains all requests from processes which are still alive, and none from those which have crashed. The first request knows that it is at the head of the queue, and every other request knows the identity of its predecessor.*

As is the case with many such coordination problems, including the *consensus* problem, it is impossible to achieve this kind of fault tolerance in an asynchronous distributed system.

Theorem 20 *In an asynchronous distributed system, fault tolerant queuing is impossible in the presence of processes which can crash.*

Proof: We reduce the consensus problem to distributed queuing. Since consensus is impossible in the presence of faulty processes (Fischer, Lynch and Paterson [25]), so is queuing.

In the consensus problem, each process starts off with an input value (either zero or one). We have available to us a fault tolerant distributed queue, Q. Every process joins Q. Once a process has joined Q, it knows its successor, and sends it its own id.

A process waits till it receives its predecessor's id. If the process is at the head of the queue, it outputs its own input. Otherwise, the process follows the predecessor links on the queue till it reaches the head of the queue, and output the value at the head of the queue.

This ensures that all the processes return the same value, and the return value is the input value of some process. ■

The implications of this theorem are that we have to settle for less strict definitions of fault tolerance. In contrast with fault *tolerance*, we focus on fault *recovery*. Specifically, we show how to make the Arrow protocol *self-stabilizating*. We add actions to the the Arrow protocol, so that it can recover from failures which might drive it to an inconsistent, or "illegal" state. A description of the self-stabilizing Arrow protocol has been published in [37].

4.1 Self-stabilization

This section describes a self-stabilizing Arrow protocol. Informally, a protocol is self-stabilizing if, starting from an arbitrary (possibly illegal) initial global state, it eventually reaches a "legal" global state, and henceforth remains in a legal state. The idea of self-stabilization was first introduced into distributed systems by Dijkstra [22].

Self-stabilization is appealing for its simplicity. Rather than enumerate all possible failures and their effects, we address failures through a uniform mechanism. Our self-stabilizing Arrow protocol is *scalable*: each node interacts only with its immediate neighbors, without the need for global coordination.

Of course, self-stabilization is appropriate for some applications, but not others. For example, ordered multicast is one natural application of distributed queuing, in which all participating nodes receive the same set of messages in the same order. An ordered multicast protocol based on self-stabilizing queuing might omit messages or deliver them out of order in the initial, unstable phase of the protocol, but would eventually stabilize and deliver all messages in order. Our protocol is appropriate only for applications that can tolerate such transient inconsistencies.

The key idea is that the global predicate defining the legality of a global protocol state can be written as the conjunction of many purely local predicates, one for each edge of the spanning tree used by the Arrow protocol. We show that the delay needed to self-stabilize the Arrow protocol differs from the delay needed to self-stabilize a rooted spanning tree by only a constant. Since distributed queuing is a global relation, it might seem surprising that it can be stabilized in constant additional time by purely local actions.

We show that the Arrow protocol is *locally checkable* (as defined in [11]) and we could use the general technique devised by [11] to do local correction. But this would lead to a stabilization time of the order of the diameter of the tree, where as our scheme gives a constant stabilization time.

4.1.1 Model and Assumptions

We assume all communication links are FIFO, and that message and processor delays are bounded and known in advance. In particular, a node can *time out* if it is waiting for a response. If the time out occurs, no response will be forthcoming. Gouda and Multari [32] have shown that such a timeout assumption is necessary for self-stabilization.

Self-stabilizing protocols can be built in a layered fashion [58]. The protocol presented here is layered on top of a self-stabilizing rooted spanning tree protocol [1, 7, 23].

In this paper, we focus only on the upper layer, assuming that our protocol runs on a *fixed rooted spanning tree.* Every node knows of its parent and children in the spanning tree, and for our purposes this data is incorruptible. We show how to stabilize the arrows and the find messages. The stabilization of the spanning tree (lower layer) and the stabilization of the arrows (upper layer) execute in parallel.

4.1.2 Local and Global States

Recall that an underlying self-stabilizing protocol yields a rooted spanning tree T which we treat as fixed. Every node knows its neighbors in the spanning tree. As described above, in the standard Arrow protocol, each node v has a pointer denoted by $p(v)$. Nodes communicate by find messages. [1]

Initially, each node is in a legal local state (for example, integer variables have integer values), but local states at different nodes can be inconsistent with each other, so that the global state is not legal. We assume that the network edges can hold a finite number of messages. The algorithm executing at a node is fixed and incorruptible.

A global state of the protocol consists of the value of $p(v)$ for every vertex v of T (that is, the orientation of the arrows) and the set of find messages in transit on the edges of T.

It is natural to define a legal protocol state as one that arises in a normal execution of the protocol. In the initial quiescent state, following the pointers from any node leads to a unique "sink"(a node whose arrow points to itself). A node initiates a queuing request by sending a find message to itself. When a node v gets a find message, it forwards it in the direction of $p(v)$ and flips $p(v)$ to point to the node where the find came from. If $p(v)$ is v, then the find has been queued behind v's last request. Any of these actions is called a *find transition.* A legal execution of the protocol moves from one global state to the next via a find transition.

Definition 2 *A state is* quiescent *if following the arrows from any node leads to a unique sink and there are no find messages in transit.*

Definition 3 *A state is* legal *either if it is a quiescent state or it can be reached from a quiescent state by a finite sequence of find transitions.*

[1]These *find* messages are the same as the *queue* messages used in the description of the Arrow protocol in chapter 2.

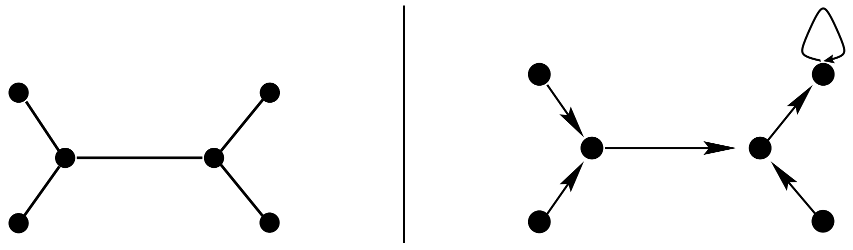


Figure 4.1: Quiescent state of the Arrow protocol
On the left is the spanning tree T. On the right is a *legal quiescent* state of the protocol

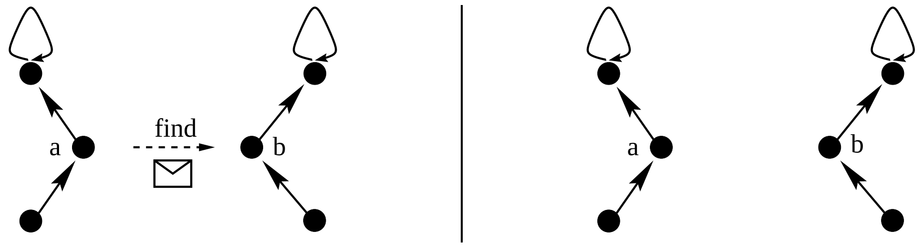


Figure 4.2: Non Quiescent States of the Arrow protocol
On the left is a *legal* state which is not *quiescent*. On the right is an *illegal* state

In a possible (illegal) initial state, $p(v)$ may point to any neighbor of v in T, and each edge may contain an arbitrary (but finite) number of find messages in transit in either or both directions. See Figure 4.1 and Figure 4.2.

4.1.3 Local Stabilization Implies Global

Though the predicate defining whether a protocol state is legal or not is a global one, which depends on the values of all the pointers and the finds in transit, we show that it can be written as the conjunction of many local predicates, one for each edge of the spanning tree.

Suppose the protocol was in a quiescent state (no *finds* in transit). Let e be an edge of the spanning tree connecting nodes a and b. e divides the spanning tree into two components, one containing a and the other containing b. There is a unique sink which either lies in the component containing a or in the other component. Since all arrows should point in the direction of the sink, either b points to a or vice versa, but not both.

Now if the global state were not quiescent and there was a find message in transit from a to b, it must be true that a was pointing to b before it sent the find, but no longer is (the actions of the protocol cause the arrow turn away from the direction it just forwarded the find to the direction the find came from). a and b both point away from each other when the find is in transit.

The above cases motivate the following definition. Denote the number of find messages in transit on e by $F(e)$. $p(a, e)$ is 1 if a points on e (i.e to b) and 0 otherwise. $p(b, e)$ is defined similarly. For an edge e, we define $\phi(e)$ by

$$\phi(e) = p(a, e) + p(b, e) + F(e)$$

We say that edge e is *legal* if $\phi(e) = 1$ (either $p(a) = b$ or $p(b) = a$ or a find is in transit, but no two cases can occur simultaneously).

We now state and prove the main theorem of this section.

Theorem 21 *A protocol state is legal if and only if every edge of the spanning tree is legal.*

Proof: Follows from Theorems 22 and 23.

■

Theorem 22 *If a protocol state is legal, then every edge of the spanning tree is legal.*

Proof: In a quiescent state, there are no finds in transit and we claim that for any two adjacent nodes on the tree a and b, either a points to b or vice versa, but not both.

Clearly, a and b cannot both point to each other since we will not have a unique sink in that case. Now suppose that a and b pointed away from each other. Then we can construct a cycle in the spanning tree as follows. Suppose s was the unique sink. Following the arrows from a and b leads us to s. These arrows induce paths p_a and p_b in the tree, which intersect at s (or earlier). The cycle consists of: edge e, p_a and p_b. Thus $\phi(e)$ is 1 for every edge e.

Further, any find transition preserves $\phi(e)$ for every edge e. To prove this, we observe that a find transition could be one of the following (v is a node of the tree).
(1) v receives a find from itself; it forwards the find to $p(v)$ and sets $p(v) = v$
(2) v receives a find from u and $p(v) \neq v$; it forwards the find to $p(v)$ and sets $p(v) = u$
(3) v receives a find from u and $p(v) = v$; it queues the request at v and sets $p(v) = u$.

In each of the above cases, it is easy to verify that $\phi(e)$ is preserved for every edge e. Since every legal state is reached from a quiescent state by a finite sequence of find transitions, this concludes the proof.

■

Theorem 23 *If every edge of the spanning tree is legal then the protocol state is legal.*

Proof: Let L be a protocol state where every edge is legal. Consider the directed graph A_L induced by the arrows $p(v)$ in L. Since each vertex in A_L has out-degree 1, starting from any vertex, we can trace a unique path. This path could be non-terminating (if we have a cycle of length greater than 1) or could end at a self-loop.

Lemma 24 *The only directed cycles in A_L are of length one (i.e self loop).*

Proof: Any cycle of length greater than two would induce a cycle in the underlying spanning tree, which is impossible. A cycle of length two implies an edge $e = (a, b)$ with $p(a) = b$ and $p(b) = a$. This would cause $\phi(e)$ to be greater than one and is also ruled out.

■

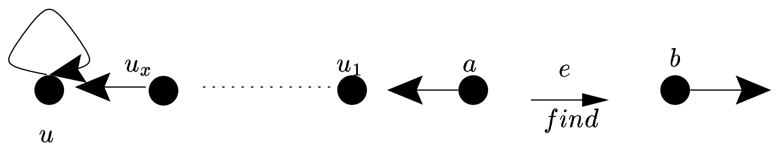


Figure 4.3: Proof of Theorem 23.
Global State L has a find on e and a self-loop on u.

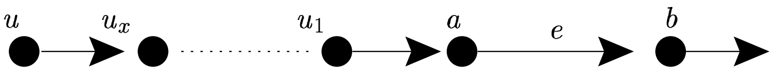


Figure 4.4: Proof of Theorem 23.
Global State L' has one find message less than L. L can be reached from L' by a sequence of find transitions

The next lemma follows directly.

Lemma 25 *Every directed path in A_L must end in a self-loop.*

We are now ready to prove the theorem. We show that there exists some quiescent state Q and a finite sequence of find transitions seq which takes Q to L. Our proof is by induction on k, the number of find messages in transit in L.

Base case: $k = 0$, i.e no find messages in transit. We prove that L has a unique sink and is a quiescent state itself and thus seq is the null sequence.

We employ proof by contradiction. Suppose L has more than one sink and s_1 and s_2 are two sinks such that there are no other sinks on the path connecting them on the tree T. There must be an edge $e = (a, b)$ on this path such that neither $p(a) = b$ nor $p(b) = a$. To see this, let n be the number of nodes on the path connecting s_1 and s_2 (excluding s_1 and s_2). The arrows on these nodes point across at most n edges. Since there are $n + 1$ edges on this path there must be at least one edge e which does not have an arrow pointing across it. For that edge, $\phi(e) = 0$, making it illegal and we have a contradiction.

Inductive case: Assume that the result is true for $k < \ell$. Suppose L had ℓ find messages in transit. Suppose a message was in transit on edge e from node a to node b (see Figure 4.3). Since $\phi(e) = 1$, a should point away from b and b away from a. We know from Lemma 25 that the unique path starting from a in A_L must end in a self-loop. Let $P = a, u_1 \ldots u_x, u$ be that path with u having a self-loop.

Clearly, we cannot have any find messages on an edge in P, because that would cause ϕ of that edge to be greater than one (an arrow pointing across the edge and a find message in transit). Consider a protocol state L' (see Figure 4.4) where u did not have a self-loop. Instead, $p(u) = u_x$ and all the arrows on path P were reversed i.e. $p(u_x) = u_{x-1}$ and so on till $p(u_1) = a$. e was free of find messages and $p(a) = b$. The state of the rest of the edges in L' is the same as in L.

We show that the ϕ of every edge in L' is one. The edges on P and the edge e all have ϕ equal to

1, since they have exactly one arrow pointing across them and no finds in transit. The other edges are in the same state as they were in L and thus have ϕ equal to 1.

Moreover, L can be reached from L' by the following sequence of find transitions $seq_{L',L}$: u initiates a queuing request and the find message travels the path $u \rightarrow u_x \rightarrow u_{x-1} \ldots u_1 \rightarrow a$, reversing the arrows on the path and is currently on edge e.

Since L' has $\ell - 1$ find messages in transit and every edge of T is legal in L', we know from induction that L' is reachable from a quiescent state Q by a sequence of find transitions $seq_{L'}$. Clearly, the concatenation of $seq_{L'}$ with $seq_{L',L}$ is a sequence of find transitions that takes quiescent state Q to L.

■

Self Stabilization on an Edge

Armed with the above theorem, our protocol simply stabilizes each edge separately. Stabilizing each edge to a legal state is enough to make the global state legal. Nodes adjacent to an edge e repeatedly check $\phi(e)$ and "correct" it, if necessary.

The following decisions make the design and proof of the protocol simpler:

- The corrective actions to change $\phi(e)$ are designed not to change $\phi(f)$ for any other edge f. This is a crucial point so that now the effect of corrective actions is local to the edge only and we can prove stabilization for each edge separately.

- Out of the two adjacent nodes to an edge e, the responsibility of correcting $\phi(e)$ rests solely with the parent node (parent in the underlying rooted spanning tree T). The child node never changes $\phi(e)$.

If the value of $\phi(e)$ could be determined locally at the parent, then we would be done. The problem though, is that $\phi(e)$ depends on the values of variables at the two endpoints of e and on the number of find messages in transit. This can be computed by the parent after a round trip to the child and back, but the value of $\phi(e)$ might have changed by then.

The idea in the protocol is as follows. The parent first starts an "observe" phase when it observes $\phi(e)$. It does not change $\phi(e)$ during the observe phase. Since the child never changes $\phi(e)$ anyway, $\phi(e)$ remains unchanged when the parent is in the observe phase. The parent follows it up with a "correct" phase during which it corrects the edge if it was observed to be illegal.

The corrective actions are one of the following. We reemphasize that these change $\phi(e)$ but don't change ϕ of any other edge of the spanning tree. a is the parent and b is the child of e.
(1) If $\phi(e)$ is 0, inject a new find message onto e (without any change in $p(a)$), increasing $\phi(e)$ to one.
(2) If $\phi(e) > 1$, and $p(a) = b$, then reduce $\phi(e)$ by changing $p(a) = a$.
(3) If $\phi(e) > 1$ but $p(a) \neq b$, then there must be find messages in transit on e. We show that eventually these find messages must reach a which can reduce $\phi(e)$ by simply ignoring them.

It remains to be explained how the parent computes $\phi(e)$. At the start of the observe phase, it sends out an *observer* message which makes a round trip to the child and back. Since the edges are FIFO, by the time this returns to the parent, the parent has effectively "seen" the number of finds in transit. The observer has also observed $p(b)$ on its way back to the parent. The parent computes $\phi(e)$ by combining its local information with the information carried back by the observer. Once the observer returns to the parent, it enters a correct phase and the appropriate corrective action is taken.

To make the protocol self-stabilizing, we start an observe phase at the parent in response to a timeout and follow it up with a correct phase. The timeout is sufficient for two round trips from the parent to the child and back. If we have an observe phase followed by a "successful" correct phase, then the edge would be corrected, and would remain legal thereafter. Each observe phase has an "epoch number" to help the parent discard observers from older epochs, or maliciously introduced observers.

4.1.4 Protocol Description

In this section, we describe the protocol for a single edge e connecting nodes a and b where a is the parent node.

States and Variables:
Node a has the following variables.
(1) $p(a)$ is a's pointer (or arrow), pointing to a neighbor on the tree or to itself.
The rest are variables added for self-stabilization:
(2) *state*, is boolean and is one of *observe* or *correct*.
(3) *sent* is an integer and the number of finds sent on e since the current observe phase started.
(4) *epoch* is an integer which is the epoch number of the current observe phase.
(5) v is an integer and is a's estimate of $\phi(e)$ when it is in a correct state.

The only variable at b is the arrow, $p(b)$.

Messages: There are two types of messages. One is the usual find message. The other is the *observer* message, which a uses to observe $\phi(e)$. In response to a timeout, a increments *epoch* and sends out message *observer(epoch)*, indicating the start of observe epoch *epoch*. Upon receipt, b replies with *observer(c,p(b,e))*.

Transitions: The transitions are of the form (event) followed by (actions). A timeout event occurs when a's timer exceeds twice the maximum round trip delay from a to b and back. The timer is reset to zero after a timeout.

Transitions for a (the parent).

- Event: Timeout
  -Reset *state* to *observe*, *sent* to 0 and increment *epoch* {the epoch number}.
  -Send observer(*epoch*) on e.

- Event: ($state = observe$) and (receive find from b)

  - If ($p(a) = a$) then set $p(a) \leftarrow b$ and the find is queued behind the last request from a. If ($p(a) \neq a$) and ($p(a) \neq b$), then forward the find to $p(a)$ and $p(a) \leftarrow b$. {the normal Arrow protocol actions.}

  - If ($p(a) = b$), send the find back to b on e; increment *sent*.

- Event: ($state = correct$) and (receive find from b) {Eventually, $state = correct$ implies that $v = \phi(e)$ }
  -If $v > 1$, then ignore the find; decrement v {since $\phi(e)$ has decreased}
  -Else (if $p(a) = b$) send the find back to b {this situation would not arise in a legal execution}.
  -Else, normal Arrow protocol actions.

- Event: ($state = observe$) and (receive *observer(d,x)* on e)
  -If $epoch \neq d$ then ignore the message. {This observer is from an older epoch or is spurious}.
  -Else change *state* to *correct*. $v = sent + x + p(a, e)$. {This is a's estimate of $\phi(e)$, and is eventually accurate}.

  Take corrective actions (if possible).
  - If $v = 0$ then send find to a; increment v.
  - If ($v > 1$ and $p(a) = b$), then $p(a) \leftarrow a$ and decrement v.

- Event: (receive find from node $u \neq b$ on an adjacent edge) and ($p(a) = b$)
  - normal Arrow protocol actions; increment *sent* {since a find will be sent on e}.

Actions for b (the child).

- Event: receive find from a.
  -If $p(b) = a$, then send find back to a.
  -Else, normal Arrow protocol actions.

- Event: receive *observer(c)*. {a wants to know $p(b, e)$.}
  -Send *observer(c,p(b,e))* on e.

4.1.5 Proof of Correctness

We prove two properties for every edge. The first is *closure*: if an edge enters a legitimate state then it remains in one. The second is *stabilization*: each edge eventually enters a legitimate state. We prove these properties with respect to a stronger predicate than $\phi(e) = 1$, since in addition to the arrows and find messages being legal, we will need to include the legality of the variables introduced for self-stabilization.

Each observer has a count (or sequence number). *sent* is a counter at a which is reset to zero at the beginning of every observe phase and incremented every time a sends out a find message on e. The current epoch number at a is *epoch*.

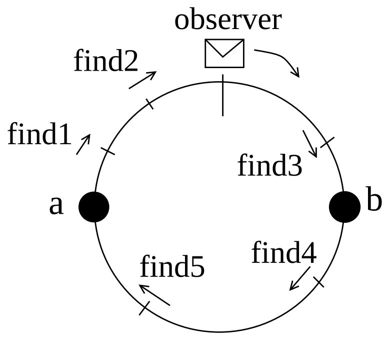


Figure 4.5: The edge is a cycle.
Messages find1 and find2 belong to F_{al} and the other three finds to F_{la}

We visualize the edge as a directed cycle (see Figure 4.5), with the link from a to b forming one half of the circumference and the link from b to a the other half. The position of the observer(s), the find messages in transit, the nodes a and b are all points on this cycle. Messages travel clockwise on this cycle and no message can overtake another (FIFO links).

Notation. Let $L_{a \to b}$ denote the maximum latency from node a to b, and $L_{b \to a}$ denote the maximum latency from b to a. Let R denote the maximum round trip time of a message from a to b and back, i.e., $R = L_{a \to b} + L_{b \to a}$.

Timeouts: Node a times out and starts a new observe phase after time $2R$.

Suppose there was only one "current observer", i.e. an observer whose count matched the current epoch number (*epoch*) stored at a. Let F denote all the find messages in transit on e. We can divide F into two subsets: F_{al}, find messages between a and the current observer and F_{la}, the find messages between the current observer and a. Clearly, $|F_{al}| + |F_{la}| + p(a,e) + p(b,e) = \phi(e)$. If the observer is between b and a, it contains the value of $p(b,e)$ as observed when it passed b. We denote this value by $p_{obs}(b,e)$.

Predicate 1 *The current observer is between a and b, and $\phi(e) = sent + p(a,e) + p(b,e) + |F_{la}|$.*

Predicate 2 *The current observer is between b and a, and $\phi(e) = sent + p(a,e) + p_{obs}(b,e) + |F_{la}|$.*

Lemma 26 *Suppose a was in an observe phase and there was only one current observer. If predicate 1 is true to start with, and a does not time out until the observer returns back, then the following will be true of the observer's trip back to a.*
(1) When the observer is between a and b, predicate 1 will remain true.
(2) After the observer crosses b and is between b and a, predicate 2 will be true.
(3) When the observer returns to a, a enters a correct state where $v = \phi(e)$.

Proof: Until the observer returns to a, it will remain in an observe phase, and $\phi(e)$ will not change. Recall that $\phi(e) = |F| + p(a,e) + p(b,e)$.

As long as the observer hasn't reached b, every find in F_{al} must have been injected by a after the observer left a and thus $F_{al} = sent$. We have $\phi(e) = p(a,e) + p(b,e) + |F_{la}| + |F_{al}| = p(a,e) + p(b,e) + |F_{la}| + sent$, satisfying predicate 1. This proves part (1).

Suppose the observer is just about to cross b. We have $\phi(e) = sent + p(a,e) + p(b,e) + |F_{la}|$. Immediately after the observer crosses b, we have $p_{obs}(b,e) = p(b,e)$. Since $\phi(e)$ has not changed and none of the other quantities $p(a,e)$, $|F_{la}|$ have changed in the meanwhile (think of the observer crossing b as an atomic operation) $\phi(e) = sent + p(a,e) + p_{obs}(b,e) + |F_{la}|$ after the observer has crossed, and predicate 2 is true.

We now prove by induction over the size of $|F_{la}|$ that predicate 2 continues to hold. Suppose it was true when $|F_{la}|$ was k. If $|F_{la}|$ decreases to $k-1$, then a find must have been delivered to a. If $p(a,e)$ was 1, then the find would have bounced back on e and *sent* would have increased by 1. If $p(a,e)$ was zero (a was pointing away from b), then $p(a,e)$ would increase to 1 and *sent* would remain the same. In either case, the sum $sent + p(a,e)$ would increase by 1, and $|F_{la}| + sent + p(a,e)$ would remain unchanged. This proves part (2).

When the observer reaches a again, F_{la} will be the empty set and we thus have $\phi(e) = sent + p(a,e) + p_{obs}(b,e)$. Once the observer reaches a, a will enter a correct state and sets v to the above quantity $(sent + p(a,e) + p_{obs}(b,e))$. This proves part (3).

■

We will now define the set of legitimate states for an edge, this time including the variables introduced for self-stabilization as well.

- R_1 denotes the predicate: $\phi(e) = 1$.
- We denote the AND of the following predicates by R_2.
  (1) a's state is observe
  (2) there is exactly one current observer
  (3) the other observers have counts less than *epoch* (the current epoch number at a)
  (4) predicates 1 or 2 should be satisfied
- We denote the AND of the following predicates by R_3.
  (1) a's state is correct
  (2) there is no observer with a count greater than or equal to *epoch*
  (3) $v = \phi(e)$ (i.e, a knows $\phi(e)$)

Since a can be in either the observe state or in the correct state, but never both at the same time, only one of R_1 or R_2 can be true at a time.

Definition 4 *The edge is in a legitimate state iff the following is true:* $R_1 \wedge (R_2 \vee R_3)$.

To prove self-stabilization of the protocol, we first prove stabilization and closure for the predicate $R_2 \vee R_3$ and then for the predicate $R_1 \wedge (R_2 \vee R_3)$. This technique has been called a *convergence stair* in [32].

Lemma 27 *If $R_2 \vee R_3$ is true, then it will continue to remain true.*

Proof: We consider two cases and all the possible actions that could occur in each case.

(1) R_3 is true. As long as a is in the correct state, it will not introduce any new observers. $v = \phi(e)$ to start with; any changes to $\phi(e)$ are made at a and are also reflected in v, thus v will remain equal to $\phi(e)$. Suppose a times out and enters an observe state, it increments *epoch* and sends out an observer with sequence number *epoch*. There is only one current observer and it satisfies predicate 1 trivially. R_2 is true now and so is $R_2 \vee R_3$.

(2) R_2 is true. If the observer does not reach a and a does not time out, then the observer remains on the edge e and predicate 1 or predicate 2 will continue to hold due to Lemma 26. If a times out before the observer reaches it, then it will enter an observe state, *epoch* is incremented, and a new observer is injected into the channel with sequence number *epoch*. R_2 is still true (only one current observer; no observers with sequence number greater than *epoch*; predicate 1 is true). If the observer reaches a before it times out, then a will go to a correct state and by Lemma 26, $v = \phi(e)$ at a. Thus R_3 is true at a. ■

Lemma 28 *Closure: If $R_1 \wedge (R_2 \vee R_3)$ is true, then it will continue to remain true.*

Proof: From Lemma 27, we know that $R_2 \vee R_3$ will continue to remain true.

We have to show that R_1 will continue to hold. Since R_1 is true initially, we have $\phi(e) = 1$ to start with. If we can show that $\phi(e)$ is never changed, then we are done.

If a is in the observe state (R_2 is true), then $\phi(e)$ is never changed. If a is in the correct state (R_3 is true), we have $v = \phi(e) = 1$ (by predicate R_3). If $v = 1$, then a will not take any corrective action, and thus $\phi(e)$ is never changed.

■

Definition 5 *A state is* fresh *if a has just timed out, and thus the next time out is $2R$ time steps away. It is* half-fresh *if the next time out is at least R time steps away.*

Lemma 29 *Within $3R$ time of any state, we will reach a state where R_2 is true and the state is fresh.*

Proof: If there are any observers in transit with sequence numbers greater than *epoch*, then they will reach a within time R (the round trip time). All the observers that a injects have sequence numbers less than or equal to *epoch*. Clearly, after time R there will never be an observer with sequence number greater than *epoch*.

Within time $2R$ after that, a will time out, enter an observe state, increment *epoch* and send out a new observer resetting r to zero. Clearly R_2 is now true; there is only one current observer, the other observers have sequence numbers less than *epoch*, and predicate 1 is true. Since a has just timed out, the state is fresh.

■

Lemma 30 *Starting from any state, within $4R$ time we will reach a state where R_3 is true and the state is half-fresh.*

Proof: From Lemma 29, we know that R_2 will be true within time $3R$. Thus a is in an observe state and its current observer obeys predicates 1 or 2. If the observer reaches a before a times out (going into an observe state of a later epoch), then a will enter a correct state. And by Lemma 26, $v = \phi(e)$. Thus R_3 will be true at a.

Since the state we started out is fresh, the next time out will occur only after time $2R$. Since R is the maximum round trip time, the observer will indeed reach a within time R (before the timeout), and the next time out is at least R away. Thus the state is half-fresh.

■

Lemma 31 *Stabilization: In time $5R$ we will reach a state where $R_1 \wedge (R_2 \vee R_3)$ is true.*

Proof: From Lemma 30 within time $4R$, we are in a state where R_3 (and thus $R_2 \vee R_3$) is true and the state is half-fresh. From Lemma 27, we know that $R_2 \vee R_3$ will remain true after that. We now show that within R more time steps, predicate R_1 will also be true.

If R_3 is true then $v = \phi(e)$. If $\phi(e) = 1$, then R_1 is already true. If $\phi(e) = 0$, then a will increase $\phi(e)$ immediately and R_1 will be true. If $\phi(e) > 1$ and $p(a) = b$, then a reduces ϕ by setting $p(a) = a$.

We are now left with the case when $\phi(e) > 1$ and $p(a) \neq b$. Since $\phi(e) = p(a,e) + p(b,e) + |F|$ (where F is the set of all find messages in transit), and $p(a,e) = 0$, it must be true that $|F| > 0$. Let ϕ_c be the current value of $\phi(e)$. We prove that within R time steps, at least $\phi_c - 1$ find messages must arrive at a on e. Since the timeout is at least R time away (the state is half-fresh), a remains in a correct state for at least time R and by ignoring all those find messages, it reduces $\phi(e)$ to 1, thus satisfying R_1.

We now show that at least $\phi_c - 1$ find messages arrive at a within the next R time steps. We use proof by contradiction. Let the current state of the system be *start*, the state of the system after R time steps be *finish* and the state of the system after the maximum latency between a and b be *middle*. The time interval between *middle* and *finish* is greater than or equal to the maximum latency for the link between b and a, since $R = L_{a \to b} + L_{b \to a}$.

Suppose less than $\phi_c - 1$ messages arrived at a in time R. This means that all the while from *start* to (and including) *finish*, $v = \phi(e) > 1$ and $p(a) \neq b$. Thus, a never forwarded any finds onto e after *start*. After *middle*, and until *finish*, there will be no finds in transit from a to b, since all the find messages that were in transit at *start* would have reached b. We have two cases.

If $p(b,e) = 0$ in *middle*, then b will not forward any more finds on e till *finish*. By the time we reached *finish*, all finds that were in transit from b to a in *middle* would have reached a. At *finish*, there are no find messages in transit on e (a did not send any after *start* and neither did b after *middle*) and $p(a,e) = 0$ and $p(b,e) = 0$, and thus $\phi(e) = 0$ in *finish*, which is a contradiction.

If $p(b,e) = 1$ in *middle*, then b might forward a find on e between *middle* and *finish* and this would change $p(b,e)$ to 0. But since no finds are forthcoming from a, this is the last find that would arrive from b before *finish*. The rest of the finds would have reached a by *finish* and thus at *finish*, the value of $\phi(e)$ is at most 1 (≤ 1 finds in transit, $p(a, e) = 0$ and $p(b, e) = 0$), which is again a contradiction.

$\blacksquare$

Stabilization time: From Lemma 31, each edge will stabilize to a legitimate state in $5R$ time steps where R is the maximum round trip time for that edge. The protocol stabilizes when the last edge has stabilized. Thus the stabilization time of the protocol is $5R_{max}$ where R_{max} is the maximum of the round trip times of all the edges.

Chapter 5

Ordered Multicast

5.1 Ordered Multicast: Definition and Applications

In this section, we study the problem of *Ordered Multicast*, and how it is related to distributed queuing. Ordered multicast is the problem of ensuring that messages multicast to a group of nodes are delivered in the same order everywhere, even when these messages are generated concurrently at several sources.

Ordering multicast messages from a single source can be accomplished by adding a token number to each message, and is easy to do, if we know how to do reliable multicast. However, when there are many sources multicasting messages, the problem becomes more interesting from a distributed computing point of view. When there are many sources multicasting messages, ordered multicast requires that all the messages sent by all the sources are ordered consistently across all receivers. Note that the actual ordering of messages is not important, but it is important that all the receivers see the same order (thus seeing the same "view of the world"). This form of multicast is sometimes called *totally ordered multicast*. We are interested in *scalable* multicast protocols that allow nodes to enter or leave the multicast group without the need for global reconfiguration.

Ordered multicast arises in a variety of applications. For example, consider a distributed object cached at multiple nodes in a network. Updates to this object are "pushed" to the cache copies using multicast, a technique called *push-based* cache coherence. If several nodes issue concurrent updates, all the copies must apply those updates in the same order. This can be solved by disseminating the updates using ordered multicast.

More generally, ordered multicast is useful in any kind of middleware where the order of events originating at different nodes in a distributed system must be reflected consistently among the nodes "listening" to such events. Publish-subscribe systems such as Gryphon [12] can use such a service to provide ordered delivery of events to subscribers. Maxemchuk [47] describes a stock market application which uses ordered multicast to disseminate stock prices, and buy and sell offers. In this case, the multicasts have to be ordered in the same way at all sites sites for fairness, rather than for consistency.

Another application is a distributed implementation of a multiplayer computer game, which uses multicast to disseminate moves from a player to all other players. We might require all the players to see the moves in the same order, which calls for ordered multicast. There is currently a project underway at Brown University Computer Science, building such a game using ordered multicast [48].

In this chapter, we show a reduction from ordered multicast to distributed queuing, and explore its consequences. Earlier ordered multicast protocols employed distributed counting for ordering messages, while distributed queuing could be a more efficient means to achieve the same result. In Section 5.5, we show that on high diameter graphs, such as the list and the mesh, *distributed queuing is an inherently easier problem than distributed counting.*

The reduction works as follows. A distributed queuing protocol can be combined with any reliable multicast protocol from the literature (such as SRM [26] or RMTP [46]), to yield a reliable ordered multicast protocol. For the queuing itself, we could use any of the protocols that we have studied, such as the Arrow protocol or the Combining protocol.

We believe that any truly scalable multicast protocol must be *anonymous*, in the sense that a node performing a multicast may not be aware of the identities of the recipients. IP multicast [19], SRM [26], and RMTP [46] are all anonymous in this sense. If all nodes (or even some nodes) must know every group member, then entering or leaving the group requires a global reconfiguration, which is clearly not scalable beyond local area networks. Our approach to ordered multicast thus differs in fundamental ways from that employed by systems based on notions of *virtual synchrony* [4, 15, 49, 65], in which each node knows the exact group membership. In these systems, each node entering or leaving the group provokes a global reconfiguration (called a "view change"). These systems trade fault-tolerance for scalability, while we do the opposite.

5.2 System Assumptions

We assume that the following are available to us:

- a reliable FIFO unicast service (such as TCP), and
- a reliable single-source FIFO [1] (but otherwise unordered) multicast service (such as SRM [26] or RMTP [46]).

Our goal is to provide a layer on top of these services that forces a total order on message delivery.

We believe that a layered approach, in which we focus exclusively on ordering, reflects a sensible separation of concerns. Many existing proposals for reliable multicast [26, 46, 45] stack a repair layer on top of an unreliable multicast layer. In the same spirit, we stack an ordering layer on top of reliable multicast so we can focus on ordering independently of repair and retransmission.

[1] "Single-source FIFO" means that messages sent by any *single* processor are delivered in the order sent.

We distinguish between *receiving* a message at a node, which typically involves placing the message in a buffer, and *delivering* the message to the application running at the node. Ordered multicast requires only that message deliveries be totally ordered. Thus, messages might be received in different orders at different nodes, but are delivered in the same order at all nodes.

We assume here that nodes do not crash, and that with sufficient retransmission, any message can eventually cross any link. Fault tolerance is an important concern for ordered multicast. However, the fault tolerance needs vary with the application, and it is not possible to talk about a general scheme for fault tolerance of ordered multicast. We discussed various schemes for the fault tolerance of queuing in Chapter 4. Fault tolerant queuing can be combined with reliable multicast to yield fault tolerant ordered multicast.

Some of the results that we present in this section have appeared in [39].

5.3 The Reduction from Queuing to Ordered Multicast

We know that distributed queuing organizes operations into a total order, and each operation informs its predecessor of its identity. In the following discussion, we will change the requirement slightly, requiring that an operation, in addition to informing its predecessor, also returns to the node of origin with the identity of the predecessor. This change in definition will not change any of the analyses that we presented in earlier chapters.

The idea in imposing a total order on reliable multicasts is simple. Each multicast group has an associated queuing object. Each message m has a unique identifier, $id(m)$. When node u wants to multicast message m, it performs the operation $queue(m)$. [2] The operation adds m to the total order, and returns the identity of the predecessor, say $id(m')$. [3] Node u then packs the contents of message m, and the identifier of message m' into a single message $\langle id(m'), m\rangle$, and calls the reliable multicast service to send the pair $\langle id(m'), m\rangle$ to each member of the multicast group.

A node that receives $\langle id(m'), m\rangle$ delivers m only after delivering m'. A message with predecessor $\perp$ is delivered immediately.

Alternatively, when u performs $queue(m)$, suppose v is the originating node of message m's predecessor m'. When v learns of m, the successor of m', v itself can multicast $\langle id(m'), m\rangle$. The advantage is that this avoids the cost of sending $id(m')$ back to u. The disadvantage, however, is that u's message is multicast by some other node, v. Since reliable multicast is an expensive operation and requires storing the message for future retransmissions, this would consume v's resources for multicasting u's message.

[2] The distributed queuing itself can be done using the Arrow protocol or the Combining protocol.

[3] The whole message m' is not returned, just the message identifier, which is typically a few bytes long.

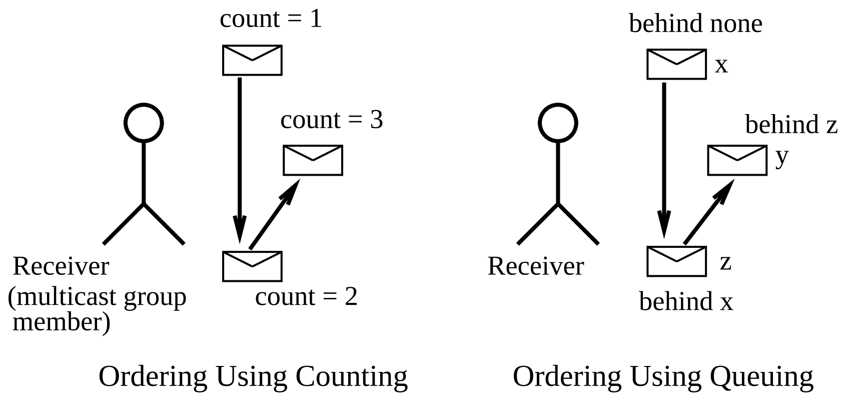


Figure 5.1: Ordered Multicast: Counting vs Queuing
The two approaches, multicast using distributed counting and using distributed queuing differ in the ordering information that comes with the messages, and in the way the ordering information is obtained.

Queuing vs Counting for Ordering

To see why it may be advantageous to use queuing to implement ordered multicast, let us consider another, perhaps more obvious reduction. The *distributed counting* problem requires issuing successive integers to requesting processes. One could implement ordered multicast simply by using any distributed counting protocol [8, 59, 68] to assign a *sequence number* to each message, and delivering a message only after all lower-numbered messages have been delivered. The two approaches are contrasted in Figure 5.1.

However, *it is sometimes faster to identify a message's immediate predecessor than to discover the number of predecessors of that message.* The two distributed queuing protocols that we discuss, the Arrow queuing protocol and the Combining queuing protocol are actually "truncated" versions of counting protocols, in which each participant quits early, after discovering its immediate predecessor, but without waiting to count the total number of its predecessors.

For example, let us consider using the Arrow protocol for distributed counting. Counting could be implemented by first queuing all the access requests (using the Arrow protocol for queuing) and then passing a *counter* object down the queue. This would lead to long latencies in case of many concurrent requests. Suppose there were three concurrent requests a, b and c ordered as a, b, c. Request c is the last in the queue; it's latency for counting is now the latency for queuing plus the message latencies from a to b and from b to c. Clearly, this could be much larger than the queuing latency. Thus we have seen that the latency of counting using the Arrow protocol is higher than the latency of queuing using the same protocol. The same is true for Combining trees.

In Section 5.5, we go a step further, and show that on high diameter underlying topologies, *distributed queuing is an inherently easier problem than distributed counting.*

5.3.1 Entering and Leaving Multicast Groups

We now briefly discuss how a node might enter or leave a multicast group, if we used the above distributed queuing protocols to implement ordering. Both the Arrow and the Combining protocols use spanning trees for connecting the member nodes. In either protocol, joining the group is straightforward: a node u just links itself as a leaf adjacent to any node v already in the tree, and informs v that it has done so. In the arrow protocol, u's arrow points to v, while in the combining protocol, v is u's parent in the tree.

Leaving a group is similar. In the arrow protocol, node u first "locks" its immediate neighbors in the tree, ensuring that message traffic between u and the neighbors has quiesced. If u is a sink, it chooses a neighbor v and makes v a sink in its place, adjusting the arrows of the new sink and neighbors to reflect this. Node u then unlocks its neighbors and leaves the group. The combining protocol is similar, except that the issue with the sink does not arise.

The important point is that group membership changes are *local* operations whose complexity depends on the degree of the node, not the size of the multicast group.

5.4 Multiple Group Ordering

Consider the scenario when there are many multicast groups, and a single node may be part of more than one group. An example is a publish subscribe system where there are many "groups" (each group might be defined by a predicate on the messages). Receivers may subscribe to many such groups and we might have two different receivers a and b, both receiving messages from two different groups, say X and Y. While single group ordering guarantees that messages from X are ordered consistently at a and b, and messages from Y are ordered similarly, it does not guarantee anything about the relative ordering across groups. It might happen that a message from X and a message from Y arrive in different orders at a and b. In some situations, we might need a stronger guarantee, that of consistent ordering across groups. This problem is called *multiple group ordering.* It has been studied earlier in [27], where it has been defined as follows.

Original definition of multiple group ordering:
If messages m_1 and m_2 are delivered to two processes, they are delivered in the same relative order (even if they come from different sources and are addressed to different but overlapping multicast groups).

However, this definition sometimes might allow scenarios which are not intuitive to the programmer. We will illustrate with an example. Suppose there were three multicast groups, the first one consisting of nodes $\{a, b\}$, the second one $\{b, c\}$ and the third one $\{c, a\}$. Suppose messages m_1, m_2 and m_3 were sent to the first, second and third groups respectively. Suppose the order of delivery at the nodes is as follows (see Figure 5.2).
b orders as m_1, m_2
c orders as m_2, m_3

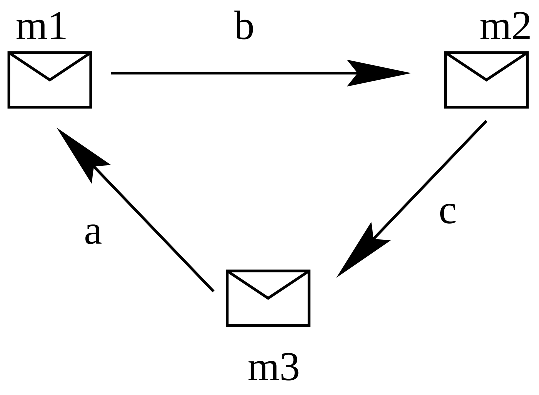


Figure 5.2: The Original Definition of Multiple Group Ordering can be Counterintuitive. Nodes a, b and c deliver messages in the order shown. Though this obeys the original definition of multiple group ordering, it might be counterintuitive to the programmer since the ordering is globally inconsistent (m_1 precedes m_2 precedes m_3 precedes m_1).

a orders as m_3, m_1

This execution obeys the above definition of multiple group ordering. However, since b saw event m_1 before m_2 and c saw event m_2 before m_3, the programmer might expect a to see m_1 before m_3, which is not the case. Thus, we suggest the following alternative definition of multiple group ordering, which is simpler to understand, and which does not allow executions such as the above.

Our revised definition of multiple group ordering:

There exists an ordering O of all the multicast messages generated by all sources such that if a node delivers a subset of these messages, then it delivers them in the order consistent with O. [4]

We note that this definition is stricter than the earlier one, i.e., if an execution is correct according to this definition, then it is also correct according to the earlier definition.

Previous Work

In [27], the authors give an algorithm called the *propagation graph* algorithm for multiple group ordering (the first definition described above). Their algorithm organizes all the nodes into a single tree, the tree structure being based on the group membership information. The multicast messages travel through this tree and are ordered by intermediate nodes. However, this scheme requires that all messages to a group be routed through a central node (one per group). This node could be a central bottleneck for a multicast group with heavy traffic group. In addition, group membership changes have to go through this central node.

[4] We say that an ordering O_1 is consistent with another ordering O_2 if the common elements in O_1 and O_2 are ordered in the same way.

5.4.1 Ordering Algorithm

We now describe our multiple group ordering algorithm. We layer this on top of a single group ordering algorithm, such as the one described in Section 5.3. We use the single group ordering algorithm to order all the messages within a multicast group.

Now, we focus on a single node v. Say v has subscribed to groups $G_1, G_2 \ldots$. From each group, v gets a sequence of messages ordered by the single group ordering algorithm. We can formulate the multiple group ordering problem in the following way: *v has to find the right interleaving of the sequences of messages from different groups $G_1, G_2 \ldots$.*

We use time stamps to determine the interleaving of the messages. Every message is multicast with a globally unique time stamp (such time stamps can be generated fairly easily using Lamport's logical clocks [44]). Given two groups G_1 and G_2, v will wait for both G_1 and G_2 to have at least one message deliverable each. Say that m_1 is the first message from G_1 and m_2 is the first message from G_2. If the timestamp of m_1 is lesser than that of m_2, then v will deliver m_1, otherwise it will deliver m_2.

The algorithm is similar for more than two groups. Node v waits till it has at least one message from each group. Then, it delivers the message with the lowest timestamp among the topmost messages from all the groups.

The advantages of this scheme are:

- It is completely anonymous, i.e. a node does not need to know about the other nodes in the multicast group.
- It can be built on top of single group ordering protocols. If the single group ordering algorithms are scalable, then this is scalable too. [5]

In our algorithm, timestamps are used only to determine the interleaving between groups. To deliver a message to the application, a node needs to receive at least one message from each group. This would be a problem if there were groups which had infrequent traffic. Suppose a node had subscribed to two groups, one with low traffic and the other with high traffic. The low traffic group might not have any messages at all, while messages from the other group are being held up, and not being delivered. One possible solution is to have every group transmit at least one message at regular intervals, so that the messages from other groups which were held up can be delivered. A single such message on a low traffic multicast group may cause delivery of pending messages in active (high traffic) multicast groups at a node. Further discussion of these issues is beyond the scope of this thesis.

However, note that our protocol waits for one message from each *group* before delivering any message, rather than one message from each *sender*, as is the case in protocols based on *virtual synchrony.*

We now prove the correctness of our protocol. Consider a hypothetical receiver r who has subscribed to all the groups. Let O be the ordering of messages at r. We show the following: if

[5] Though, as we explain further, this algorithm may not work well when a node has subscribed to many groups.

a (real) receiver v subscribes to a subset of the groups $\{G_1, G_2, \ldots\}$, then its ordering of messages is consistent with O. This would prove that the ordering obeys our (revised) definition of multiple group ordering.

Let $m_1, m_2, \ldots$ be the first messages from groups $G_1, G_2, \ldots$ respectively. The first message that v delivers is the one with the lowest timestamp among $m_1, m_2, \ldots$, say m_1. We claim that this is consistent with O. Suppose otherwise, and O ordered m_2 before m_1 even though m_2's timestamp is greater. Then r must have found (perhaps after delivering a few messages) that m_2 has the lowest timestamp among the set of all the messages at the top of the (undelivered) the sequence for each group. But this set will also include m_1, since m_1 is the first message in group G_1, and is not delivered by r yet. This implies that m_2 has a lower timestamp than m_1, which is a contradiction. This shows that the first message delivered by v is consistent with O. By induction, we can show that the total order delivered by v is consistent with O.

5.5 Queuing versus Counting

In this section, we show that on high diameter graphs, concurrent queuing is an inherently easier problem than concurrent counting. We would like to emphasize that our result *is about the problems* of queuing and counting and *is not about specific solutions* to them.

Specifically, we show that concurrent queuing has lower latency than concurrent counting. We analyze the concurrent one-shot scenario for distributed queuing and distributed counting, and show that for large diameter graphs such as the mesh and the list, the cost of a *specific queuing protocol* (the Arrow protocol) is less than a lower bound on the cost of *any counting protocol.*

Counting and queuing both order operations into a total order. However, the result in each case is different. In counting, the processor gets back the rank of its operation in the total order. In queuing, the processor gets back the identity of the successor (or predecessor). Since queuing only requires *local* information about the total order (who's next?) while counting requires global information (what's my rank in the total order?), it seems plausible that queuing is an easier problem.

To formalize this, we study the one-shot scenario for queuing and counting. We are given a network $G = (V, E)$ of processors connected by FIFO communication links. The weights of the edges are the delays of the communication links, which we assume to be fixed. A subset of these processors $R \subset V$ issue queuing (counting) requests at time zero.

For a queuing algorithm A, the queuing latency of node v, denoted by $l_Q(v, A)$, is the time till v receives the identity of its operation's predecessor.

Similarly, for a counting algorithm A, the counting latency of node v, denoted by $l_C(v, A)$, is the time till v receives the rank of its operation.

For a counting algorithm A, the cost of concurrent counting is:

$$C_C(A) = \max_{R \subset V} \left\{ \sum_{v \in R} l_C(v, A) \right\}$$

The counting complexity, denoted by C_C, is the minimum cost of concurrent counting over all counting algorithms.

$$C_C = \min_A \{C_C(A)\}$$

For a queuing algorithm B, the cost of concurrent queuing is:

$$C_Q(B) = \max_{R \subset V} \left\{ \sum_{v \in R} l_Q(v, B) \right\}$$

The queuing complexity, denoted by C_Q, is the minimum cost of concurrent queuing over all queuing algorithms.

$$C_Q = \min_B \{C_Q(B)\}$$

We first prove the following lemma, which will later help us derive lower bound on C_C for specific graphs.

Lemma 32 *For any counting algorithm, if node v receives a count i, then it must have got a message directly or indirectly from at least $i - 1$ other nodes.*

Proof: Suppose v received messages from less than $i - 1$ nodes. Then, it is possible that there are less than i requesting nodes, but node v got a count of i. This would violate the requirement of counting, which says that nodes should be assigned consecutive natural numbers starting from 1. ■

We now show that for high diameter graphs, such as the list and the mesh, the counting complexity is strictly greater than the queuing complexity.

5.5.1 G is a list

We will first study the case when G is a list of n processors, numbered $v_1 \dots v_n$ is the list order. The following lemma gives us a lower bound on the counting complexity.

Lemma 33 *If G is a list of n nodes, $C_C = \Omega(n^2)$.*

Proof: The following argument applies to *any* counting algorithm. Suppose node v receives a count of i. From Lemma 32 it must have got messages from at least $i - 1$ nodes. Since the underlying graph is a list, at least one of these nodes (say u) must be at a distance of greater than or equal to $(i - 1)/2$ from v. Since v has to receive a message from u, the latency of v's counting operation is at least $(i - 1)/2$.

If all the n nodes in the list started counting at time zero, then $n/2$ of the nodes will receive counts of at least $n/2$. For each of these $n/2$ nodes, the counting latency is at least $(n - 1)/4$.

The sum of the counting latencies is

$$C_C \geq n/2 \cdot (n - 1)/4 \geq n(n - 1)/8$$

The lemma follows. ■

The next lemma gives an upper bound on the queuing complexity, C_Q.

Lemma 34 *If G is a list of n nodes, $C_Q = O(n)$.*

Proof: Consider a specific queuing protocol, the Arrow protocol on the list G.

In chapter 3, we have analyzed the concurrent complexity of this protocol. From Theorem 3, we know that the concurrent cost of the Arrow protocol is upper bounded by $3 \log r' \cdot TSP(G_R^T)$. r' is the number of leaves of the spanning tree, which is two, since the graph is a list. $TSP(G_R^T)$ is the cost of the optimal traveling salesperson tour visiting all the nodes of the list, which is $2n$.

Thus, the concurrent queuing cost of the Arrow protocol is upper bounded by $12n$. Since the concurrent complexity of queuing is not greater than the cost of a specific solution, we have $C_Q = O(n)$.

■

From Lemmas 33 and 34, we have the following theorem.

Theorem 35 *If G is a list, $C_C > C_Q$.*

5.5.2 G is a mesh on n nodes.

We can show a similar result for the mesh of n nodes, of dimension $\sqrt{n} \times \sqrt{n}$.

Lemma 36 *If G is a mesh of n nodes, $C_C = \Omega(n\sqrt{n})$.*

Proof: If all the n nodes in the mesh started counting at time zero, then $n/2$ of them will receive counts of $n/2$ or greater. Let v be a node which received a count of $n/2$ or greater.

From Lemma 32, we know that v must have received messages from at least $n/2 - 1$ other nodes. At least one of these $n/2 - 1$ nodes must be at a distance of greater than $\sqrt{n}/6$ from v, since the number of nodes at a distance of less than $\sqrt{n}/6$ from v is less than $2\sqrt{n} \cdot \sqrt{n}/6 = n/3$. Since v has to get a message from some node which is at a distance of $\sqrt{n}/6$ from it, v's latency is at least $\sqrt{n}/6$.

Summing this over all the $n/2$ such nodes which receive counts of $n/2$ or more, we get the counting complexity to be at least $n/2 \cdot \sqrt{n}/6 = \Omega(n\sqrt{n})$. ■

Lemma 37 *If G is a mesh of n nodes, $C_Q = O(n \log n)$.*

Proof: Similar to the proof of Lemma 34, we analyze the cost of the Arrow protocol on the mesh. We choose any spanning tree of the mesh to execute the protocol. This tree has n nodes, and hence, $n - 1$ edges.

Theorem 3 shows that the cost of the Arrow protocol is bounded by $3 \log r' \cdot TSP(G_R^T)$. r' is the number of leaves of the tree, which is less than n. $TSP(G_R^T)$ is the cost of the Euler tour around the tree, which is less than $2n$, since the tree has $n - 1$ edges. Thus, the concurrent cost of the Arrow protocol for the mesh is bounded by $6n \log n$. So is the concurrent complexity of queuing on the mesh. ■

Lemmas 36 and 37 lead to the following theorem.

Theorem 38 *If G is a mesh on n nodes, $C_C > C_Q$.*

Chapter 6

Beyond Distributed Queuing: Distributed Fetch-and-ϕ

6.1 A New Data Structuring Technique

Thus far, we have been studying the relatively "simple" problem of distributed queuing. Queuing orders operations, and informs each operation about its successor. We have shown that just this much information can be useful in applications such as ordered multicast and distributed directories. In this chapter, we show how to use such a simple primitive as queuing to build a *a general distributed data structure.*

A general distributed data structure encapsulates a value v and supports the *fetch-and-ϕ* operation for any binary function ϕ. The operation *fetch-and-$\phi(o)$* changes v to $\phi(o, v)$ and returns the old value of v. As is the case with any concurrent data structure, many processors can concurrently initiate *fetch-and-ϕ* operations on this data structure too.

All the distributed data structures that we have studied, such as queuing and counting, are special cases of this general *fetch-and-ϕ* data structure. For the counting data structure, $\phi(o, v)$ is the increment function which adds one to v. For the adding data structure, $\phi(o, v)$ is the addition function, which adds o to v. For the test and set data structure, $\phi(o, v)$ sets v to 1.

We now present our construction of the general distributed data structure. More precisely, given a sequential implementation of a data structure, we will transform it into a concurrent implementation, which has both low contention and latency. Low contention means that no single node (or group of nodes) in the network faces an undue amount of message traffic (i.e. there is no *hot spot*), while low latency means that every operation completes within a "small" number of message delays.

This is the first such general purpose data structuring technique that we know of which is both parallel (i.e. many operations are progressing at the same time) and distributed (no single node in the network is loaded with an undue share of the work).

Our construction combines two ideas: distributed queuing and pointer jumping. We will first

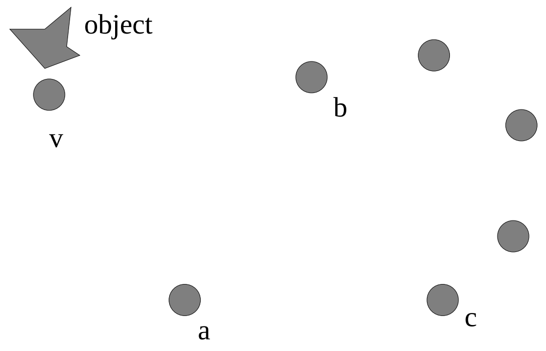


Figure 6.1: Object Management using Queuing: Concurrent Requests

revisit a solution which is based on queuing. This solution is distributed, but sequential (only one operation is progressing at a time). We then show how to add parallelism to this solution by using the idea of pointer jumping.

In this discussion, we assume a message passing model of computation.

6.2 The Algorithm using Distributed Queuing

First, we revisit a simple way of building a distributed data structure using queuing. The object that we have to implement, say O resides at one node at a time. Say O is currently at node u. When an operation is initiated at some node v (possibly different from u),

- Distributed Queuing: v's operation is queued behind u and u is informed of v's operation.
- Move object: Once u has performed its operation on O, it moves the object to v, and v can perform its operation.

Concurrent operations are treated similarly. The operations are queued and the object passes from holder to successor. Since the operations are applied sequentially on O, all we need to know is a sequential implementation of O, and we can turn it into a distributed implementation using the above. The advantage of this queuing based scheme is low contention. If we have a good queuing algorithm, then no single processor will be a bottleneck. This algorithm is shown in Figures 6.1 and 6.2.

However, the above algorithm has a problem: high latency. Processors have to wait till the object comes to them. If many processors concurrently requested the object, then the queue of processors waiting for the object could be long, leading to high latency for the processors at the end of the queue. In addition, there is only one operation proceeding at a time, so that there is no parallelism. *This algorithm is distributed, but not parallel.*

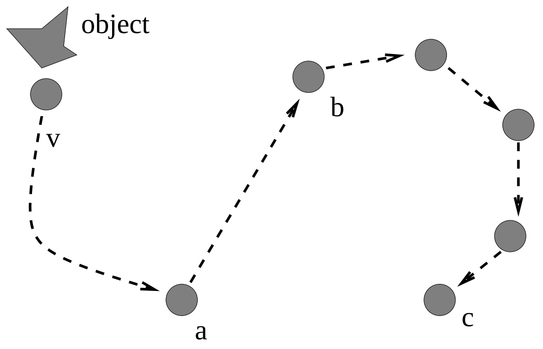


Figure 6.2: Object Management using Queuing: Long latencies
Concurrent requests are queued, but processors (such as c) might face a long delay before the object reaches them.

6.3 The Algorithm using Distributed Queuing and Pointer Jumping

Now we show how to add parallelism to the above algorithm. Suppose that we have a long queue of operations, as shown in Figure 6.3. Suppose that the queue of operations is $v_1 v_2 \ldots v_k v$, and S is the state of the object O before v_1's operation. To make the presentation easier, we will overload the symbol u and use it to denote both the node u and node u's operation. The following is the key observation.

Observation 2 *If v knows S and the (ordered) list of operations $v_1 v_2 \ldots v_k$, then it is equivalent to v getting the object O.*

The reason is that v could apply the operations $v_1 v_2 \ldots v_k$ on S in order *locally* and the result would be the same if O were passed to it down the queue, after being operated by $v_1 \ldots v_k$.

Now, finding the identities of all the elements of the list $v_1 \ldots v_k$ is something that the processors could do when they are waiting for the object to come to them. Moreover, this can be done in parallel by all the processors, using a technique known as *pointer jumping* [18].

Pointer Jumping: The technique of pointer jumping has been used in parallel list ranking algorithms. It was first introduced by Wyllie [70]. It was used by Tarjan and Vishkin [63] to compute depths of nodes in trees in parallel.

We show how to use this idea profitably in reducing latencies and adding parallelism in distributed object management. We will explain our idea using an example.

Model: We will assume that the underlying comunication network is a complete graph. That is, every processor is connected to every other processor by a direct communication link. We will assume a synchronous model, where all the processors proceed in rounds. In each round, a processor can send a message (or a constant number of messages) to any other processor. We will later explain how our algorithm can be adapted to an asynchronous model.

The aim for each processor is to find all operations ahead of itself in the queue. The algorithm is described below. Refer to Figures 6.3 to 6.7 for an example execution.

- To start off, each processor knows the identity of its successor in the queue (provided to it by distributed queuing).
- In the first round, each processor contacts its successor and informs the successor of its identity. Thus, at the end of the first round, each processor knows its predecessor to a distance of 1.
- In the second round, each processor contacts its predecessor. Processor v_i's predecessor, v_{i-1} replies back with the identity of its predecessor, v_{i-2}, and at the end of the round, v_i knows v_{i-1} and v_{i-2}, and each processor knows its prefix to a distance of 2.
- Similarly, in the third round, v_i contacts v_{i-2} and finds out about v_{i-3} and v_{i-4}.
- At the end of the ith round, each processor knows its prefix till a distance of 2^{i-1}. Thus, after $\log k + 1$ rounds, v will know the identity of all the operations $v_1 \dots v_k$ (in order), and the value of O before v_1 is applied. As noted in Observation 2, this is equivalent to getting the object itself.

Thus, after $\log k + 1$ rounds, v's operation is complete. Assuming that local computation is negligible, so that message delays account for the latency, each "round" consists of a message and its response, which is two message delays. Thus, v's operation latency is the time for queuing plus $2(\log k + 1)$ message delays.

If the object were to travel down the queue (as in the original algorithm), then v's latency would be k message delays. We have achieved a substantial saving in latency k versus $2 \log k$. This comes at the cost of increased number of messages. These are analyzed in detail in the next section.

The algorithm presented above is distributed, and also parallel, since the operations are progressing in parallel (finding the prefixes in parallel). Thus we have a new parallel and distributed data structuring scheme. The algorithm for the asynchronous model is presented in detail in Figure 6.9.

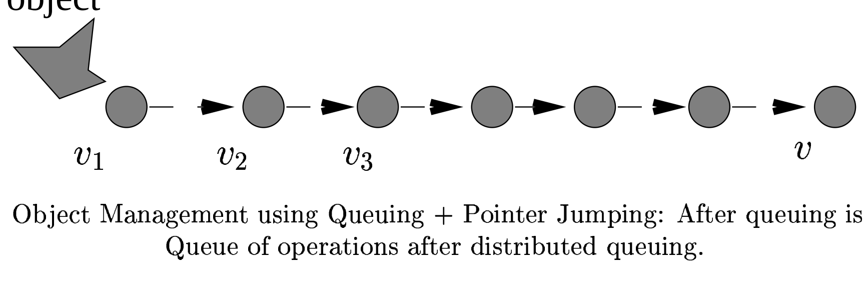


Figure 6.3: Object Management using Queuing + Pointer Jumping: After queuing is complete. Queue of operations after distributed queuing.

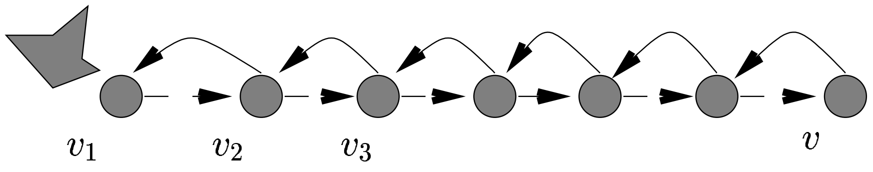


Figure 6.4: Object Management using Queuing + Pointer Jumping: After one round. v_2 knows about v_1's operation, and the state of the object before it is applied. Thus v_2 has the object. The dotted lines denote the queuing order while the solid lines denote the earliest predecessor that a processor knows about.

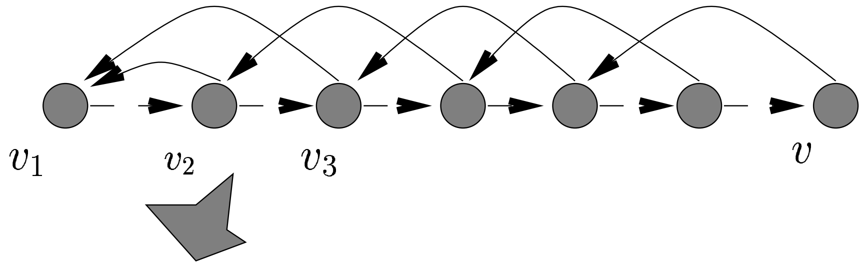


Figure 6.5: Object Management using Queuing + Pointer Jumping: After two rounds. Processors know the prefix of the queue till a distance of 2.

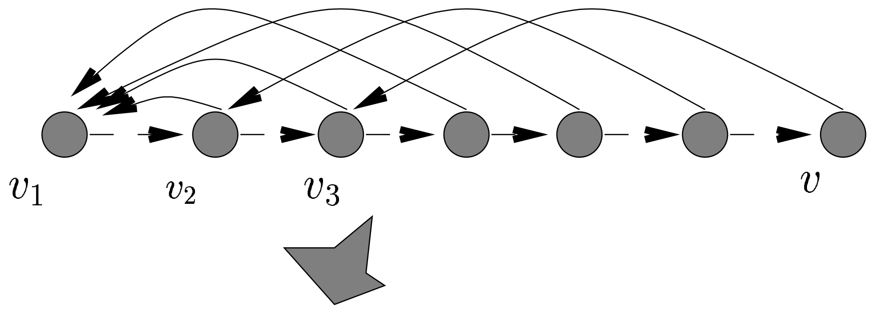


Figure 6.6: Object Management using Queuing + Pointer Jumping: After three rounds. After i rounds, processors know their prefix till a distance of 2^{i-1}, or till the object has been reached.

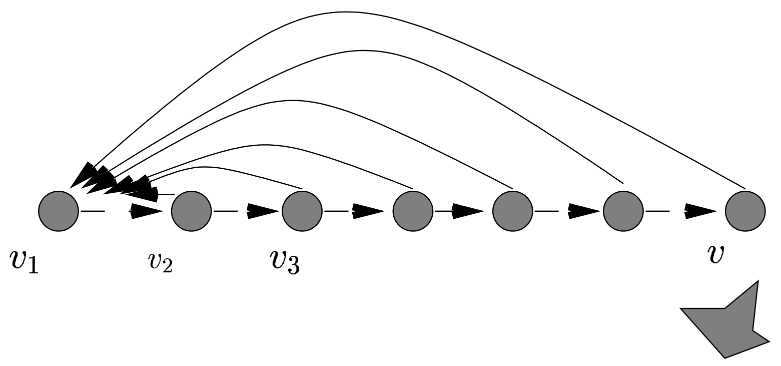


Figure 6.7: Object Management using Queuing + Pointer Jumping: After four rounds. v has the object, since it knows all the operations of $v_1 \ldots v_k$, and can apply them locally.

For simplicity, we assume that each node issues at most one operation. The algorithm can easily be extended for the case when a node issues multiple operations. Without any ambiguity, we can now refer to node v's operation by v. A node u is called the *ancestor* of a node v if u's operation is ordered before v's. Similarly, node v is called the *descendant* of u.

Each node v has the following variables:

- *ancestor*: a processor which is an ancestor of v
- *cumulativeOperation*: a list of operations

  Invariant: *cumulativeOperation* is the sequence of all operations from my operation up to, but excluding the *ancestor*.
- *isOpComplete*: a boolean variable, initially False
- *result*: valid if *isOpComplete* is True

There are three message types:

- *ObjectRequest*: request for an object
- *ObjectResponse*: response with the value of the object
- *PartialResponse*: if a node does not have the object, it replies to its descendant with information that will extend the descendant's prefix.

Figure 6.8: The variables and message types for object management using pointer jumping.

We will describe the algorithm in an event-driven manner. Each message receipt is an event, triggering some action. The initiation of a node's operation is also an event, though arising internal to the node.

```
Node Initiates Operation $op$:
    $ancestor \leftarrow$ distributedQueuing()
/* The Queuing algorithm returns with ordering information. One operation is the head of the
operation queue, and gets the value NULL. The other operations get the id of the predecessor */
    if ($ancestor$ is NULL)
        send($ObjectResponse$(NULL)) to self
    else
        $cumulativeOperation \leftarrow op$
        $isOpComplete \leftarrow$ False
        send($ObjectRequest$()) to $ancestor$

Receive $ObjectRequest()$ from processor $p$:
    if ($isOpComplete$) then
        send($ObjectResponse$($result$)) to $p$
    else
/* I don't have the object to send back, but have information that will extend $p$'s prefix */
        send($PartialResponse$($cumulativeOperation$, $ancestor$)) to $p$

Receive $PartialResponse(oplist, newancestor)$ from processor $p$:
/* extend my prefix information. Recall the invariant: $cumulativeOperation$ is the list of all oper-
ations till the operation of processor $ancestor$ */
    $cumulativeOperation \leftarrow cumulativeOperation + oplist$
    $ancestor \leftarrow newancestor$
    send($ObjectRequest$()) to $ancestor$

Receive $ObjectResponse(object)$ from processor $p$:
/* apply the other operations locally and get the result */
    $result \leftarrow$ apply($cumulativeOperation$, $object$)
    $isOpComplete \leftarrow$ True
```

Figure 6.9: The pointer jumping algorithm for object management, for the asynchronous model of computation.

6.4 Complexity

We analyze the complexity of the algorithm in a synchronous model. We assume that message delay between any two processors is one time unit.

6.4.1 Latency and Message Complexity

We will first analyze a simpler problem: if all operations were queued at time zero, how much time would it take for the last operation in the queue to complete (to learn all its predecessors)? Let m denote the length of the queue at time zero.

This is answered by the analysis of pointer jumping. This analysis is well known and is due to Wyllie [70]. Each round of pointer jumping involves a message to a predecessor, asking for prefix information, followed by a response which doubles the length of the prefix. We can show that after $2i$ timesteps (i rounds), each processor knows its predecessors till a distance of 2^i. Thus, the last operation would complete in time $2 \log m$. Contrast this with time m, if the object were passed from processor to processor, down the queue.

We note that in the pointer jumping algorithm, in the synchronous model, each processor needs to handle only a constant number (two) of messages in each time step.

If we measure the number of messages, each processor sends up to two messages in each round, and there are $\log m$ rounds. Thus, the total number of messages exchanged is $2m \log m$. Contrast this with m messages, if the object were passed down the queue.

Thus, we arrive at the following comparison:

- Lazy (pass object down the queue): latency m, message complexity m
- Eager (pointer jumping): latency $2 \log m$, message complexity $2m \log m$

At a cost of more messages (a factor of $\log m$), we are getting greatly reduced latency ($O(m)$ versus $O(\log m)$.

There exist randomized work-efficient parallel prefix algorithms (due to Anderson and Miller [6, 5]), which can perform pointer-jumping in $\log m$ rounds with high probability, yet use a total of $O(m)$ messages. It would be interesting to explore if the same techniques can be used to reduce the message complexity of our distributed data structuring algorithm.

We note that m (the length of the operation queue) is the degree of concurrency, and cannot exceed the number of processors. Thus, pointer jumping would lead to significant gains in latency when the degree of concurrency is high.

Now, we turn to the analysis of the case when all the requests are not queued simultaneously (at time zero). We analyze the one-shot case, when all requests start at time zero, and compute the time till the last request in the queue gets an answer. The analysis is the same as the analysis above, except that we have to add the time for the queuing. If Q is the delay till all the requests are queued, then the last operation in the queue gets its result after time $Q + 2 \log m$. For the Arrow protocol for example, Q is bounded by the diameter of the tree chosen.

6.4.2 Message Length

Clearly, if we are passing lists of operations around, messages become very long, and it may be expensive to send such messages. The largest message exchanged would contain $O(m)$ operations, and could thus be of length $O(m)$.

Fortunately, in many cases, it is possible to find concise representations for a list of operations. For example, if the operations were addition, and the list of operations were add 1,add 2, add 3, then this can be concisely represented by the single operation add 6. In such a case, transmitting a list of operations is no more expensive than transmitting one operation. Most arithmetic operations, such as multiplication, division (or a combination of those) can be handled in this way.

6.5 Related Work

Combining Trees: Another general purpose distributed data structuring technique is Combining trees [29, 71], which we have encountered earlier. Combining trees can be used to implement any general distributed data structure. Operations move up a tree and combine enroute. They are applied together at the root, and then the results come back down the tree.

Combining Trees suffer from the lack of *locality* due to the fixed location of the root. Even if only one node is applying operations on the data structure repeatedly, the operations travel all the way up to the root and back.

Our algorithm's performance depends on the queuing protocol that we use. If we combine pointer jumping with the Arrow queuing protocol, then the resulting algorithm exhibits locality.

Herlihy's Universal Construction: Maurice Herlihy introduced a way of converting sequential specifications of objects into wait-free concurrent implementations [34], known as a universal construction. His work focuses on shared memory. The emphasis was on showing that wait-free constructions were possible, and not on the performance aspects: latency and contention. Our algorithm is also a universal construction in that sense, but isn't wait-free.

Chapter 7

Conclusions and Future Directions

In this thesis, we have presented a study of distributed queuing. We have identified queuing as a fundamental coordination problem. We show how to use queuing in ordered multicast. We show how queuing can be used in the construction of a solution to the general distributed data structuring problem of *fetch-and-ϕ*. We have compared the concurrent complexities of queuing and counting, showing that queuing is an inherently easier problem than counting on high diameter graphs.

We have focused on the Arrow queuing protocol for most of this thesis, since it has been found to be efficient both in theory and in practice. We showed that its performance under concurrency is competitive to that of the optimal. This competitive analysis of a protocol under concurrency is novel, and we have not seen it applied to distributed data structures before. We have studied the fault tolerance of the Arrow protocol, and show how to make it self-stabilizing.

This thesis is the first systematic study of the distributed queuing problem and a demonstration of its importance as a basic distributed coordination problem.

A note on the model of computation: We have used the message passing model of computation in our description of the algorithms. However, the same results (analysis of the Arrow protocol, self-stabilizing Arrow protocol, etc) can be readily extended to the shared memory model [10]. In the shared memory model, processors communicate by reading and writing to a shared memory space.

We conclude this thesis with a list of topics for future research.

7.1 Future Directions

Queuing versus Counting: We showed in Chapter 5 that on high diameter graphs, concurrent queuing is inherently an easier problem than concurrent counting. Our lower bounds on concurrent counting were based on showing that certain counting operations have to wait for a message from a faraway node, and hence have high *latencies*. Such latency-based lower bounds will not work on graphs with low diameters, such as the complete graph. But, if we model the contention (so that a node cannot process too many messages in a time step), then we might be able to get lower bounds for concurrent counting even on low diameter graphs.

Wattenhofer and Widmayer [69] show a lower bound on the contention of any counting algorithm. They show a lower bound on the number of messages that some processor has to exchange in a sequence of n counting operations spread over n processors. However, this lower bound deals with the sequential case, where the next operation is issued only after the previous one has completed and "taken full effect".

We also showed in Chapter 6 that counting can be reduced to queuing, at the cost of additional messages and latency. However, this reduction is *not wait-free*, since it requires processors to wait for messages from each other.

Counting Algorithms for Arbitrary Topologies: Most distributed counting protocols, including counting networks (perhaps the most popular family of distributed counting schemes) assume a fixed interconnection of processors. If we wanted to implement them on a real network (which is not completely connected), we would have to embed these fixed networks onto the given network. Not much is known about counting algorithms which work for general interconnection networks. In related work, Aiello et.al. [2] present approximate load balancing (related, but not the same as counting) algorithms for general networks, and use expansion properties of the graphs to express their time bounds.

Concurrent Analysis of other Distributed Data Structures: We have not seen competitive analyses of other distributed data structures in the concurrent scenario. For example, a concurrent analysis of the Counting Network.

Analysis of Arrow protocol for the Long-lived case: An obvious open question is the complete analysis of the Arrow protocol for the long-lived case (Chapter 3). The competitive ratio of Arrow for the two extremes are known. For the sequential case, it is s (the stretch of the spanning tree); for the one-shot concurrent case, it is $s \log r$, where r is the degree of concurrency. For the general long-lived case, we conjecture that it is still $s \log r$.

Extensions of the Arrow protocol: There are many directions that the Arrow protocol itself can be extended. The following problems might have solutions similar to the Arrow protocol.

- **Concurrent dequeues:** So far, we have studied the problem of enqueuing efficiently. A natural question is: what about dequeues? Can we use an algorithm like the Arrow protocol to support concurrent dequeues? We can show that distributed queuing with concurrent dequeues is *harder than distributed counting.* Thus, we would need significant modifications to the Arrow protocol to solve this. A thorough analysis of this problem is yet to be undertaken.
- **l-mutual Exclusion and Group Mutual Exclusion:** We know that the Arrow protocol can be used for solving token based distributed mutual exclusion. The l-mutual exclusion

problem is a variant of distributed mutual exclusion [24], which allows l different processors to be enabled at the same time. The regular mutual exclusion is a special case when $l = 1$. There are no queuing based algorithms for solving l-mutual exclusion. It would be interesting to find solutions similar to the Arrow protocol for this problem.

A related, but different problem called *group mutual exclusion* has recently been proposed by Joung [43]. In this problem, a set of processes can be enabled at the same time if they all belong to the same process *group*. The regular mutual exclusion is a special case when every such group has only one process in it. It seems possible that there is a queuing based algorithm for this problem too, but one has not been discovered so far.

- **Priorities:** Is it possible to add priorities to the operations in the queue, so that the operations with higher priorities are enqueued earlier than those with lower priorities?

Practical Matters: From a more practical perspective, the following questions are worth pursuing. The first is the practical performance of ordered multicast using queuing. The goal of completely anonymous, scalable ordered multicast is a tall one. There is a project underway at Brown University Computer Science, whose goal is to implement a distributed multiplayer game using ordered multicast [48]. While the scalability of the Arrow protocol for queuing has been established on local area networks (see Chapter 3), there are other unresolved issues, such as the kind of fault tolerance to be provided, how to perform unordered multicast, etc. While there has been work on scalable reliable (unordered) multicast, a scalable ordered multicast system has not been implemented yet.

Another is the experimental comparison of our new *fetch-and-*ϕ algorithm with existing algorithms for *fetch-and-*ϕ. Our preliminary experiments suggest that the new algorithm based on queuing plus pointer jumping is much faster when compared with *fetch-and-*ϕ using only queuing. A comprehensive comparison with other general techniques for *fetch-and-*ϕ, such as combining trees would be worthwhile.

Bibliography

[1] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Springer-Verlag LNCS:761*, pages 400–410, 1993.

[2] W. Aiello, B. Awerbuch, B. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 632–641, 1993.

[3] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 401–411, 1994.

[4] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[5] R. J. Anderson and G. L. Miller. *1988 Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, chapter Deterministic parallel list ranking, pages 81–90. Springer-Verlag, 1988.

[6] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. Unpublished Manuscript, 1988.

[7] G. Antonoiu and P. K. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *Euro-par'97 Parallel Processing, Proceedings LNCS:1300*, pages 480–487. Springer-Verlag, 1997.

[8] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, Sept. 1994.

[9] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *ACM Symposium on Theory of Computing*, pages 237–246, 1996.

[10] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.

[11] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 268–277, 1991.

[12] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the International Conference on Distributed Computing Systems*, 1999.

[13] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proc 37th IEEE Symposium on Foundations of Computer Science*, pages 184–193, 1996.

[14] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 161–168, 1998.

[15] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, Dec. 1993.

[16] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Trans on Computer Systems*, 2(3):251–273, Aug 1984.

[17] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proceedings of the 39th IEEE Symposium on the Foundations of Computer Science*, 1998.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[19] S. Deering. Host extensions for IP multicasting. *RFC 1112*, August 1988.

[20] S. Deering. IP multicast extensions for 4.3BSD and related systems. Stanford University, 1989.

[21] M. Demmer and M. Herlihy. The arrow directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing*, Sept. 1998.

[22] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[23] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[24] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 234–254. IEEE, 1979.

[25] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[26] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, Dec. 1997.

[27] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[28] Gnutella, http://www.gnutella.com.

[29] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. of the Third ASPLOS*, pages 64–75. ACM, 1989.

[30] A. Gottlieb and C. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, 9, 1981.

[31] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Prog. Lang. Syst.*, 5(2):164–189, Apr. 1983.

[32] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[33] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI- The Complete Reference, Volumes 1 and 2.* Scientific and Engineering Computation Series. The MIT Press, 2000.

[34] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[35] M. Herlihy. The Aleph toolkit: Support for scalable distributed shared objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC)*, January 1999.

[36] M. Herlihy, B.-H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[37] M. Herlihy and S. Tirthapura. Self stabilizing distributed queuing. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*, pages 209–223. Springer, October 2001. Appears in LNCS 2180.

[38] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, August 2001.

[39] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–96, January 2001.

[40] M. Herlihy and M. Warres. A tale of two directories: Implementing distributed shared objects in java. *Concurrency - Practice and Experience*, 12(7):555–572, 2000.

[41] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[42] T. C. Hu. Optimum communication spanning trees. *SIAM Journal on Computing*, 3(3):188–195, 1974.

[43] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[44] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[45] B. N. Levine and J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems Journal (ACM/Springer)*, 6(5), Aug. 1998.

[46] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *Proceedings of IEEE INFOCOM*, pages 1414–1424, 1996.

[47] N. F. Maxemchuk. DIMACS workshop on multicasting: Algorithms, architectures and applications
http://dimacs.rutgers.edu/workshops/multicasting/abstracts.html, May 2001.

[48] A. Mohan. personal communication, 2002.

[49] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, April 1996.

[50] The official MPI standard. http://www.mpi-forum.org.

[51] M. Naimi, M. Trehel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *Journal on Parallel and Distributed Computing*, 34(1):1–13, 1996.

[52] D. Peleg and E. Reshef. Deterministic polylog approximation for minimum communication spanning trees. In *Proc 25th International Colloquium on Automata Languages and Programming*, pages 670–681, July 1998.

[53] D. Peleg and E. Reshef. A variant of the arrow distributed directory protocol with low average complexity. In *Proc 26th International Colloquium on Automata Languages and Programming*, July 1999.

[54] R. Rajaraman. *Sharing Resources in Distributed Systems.* PhD thesis, The University of Texas at Austin, 1997.

[55] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[56] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, 25(2):47–50, 1991.

[57] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, Sept. 1977.

[58] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[59] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, November 1996.

[60] N. Shavit and A. Zemach. Combining funnels. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 61–70, 1998.

[61] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, pages 202–208, February 1985.

[62] J. Snepsheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2:113–115, 1987.

[63] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[64] J. L. A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2(2):113–115, 1987.

[65] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in Horus. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, Aug. 1995.

[66] G. Varghese. Self-stabilization by counter flushing. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

[67] G. Varghese, A. Arora, and M. Gouda. Self-stabilization by tree correction. *Chicago Journal of Theoretical Computer Science*, 1997(3):1–32, 1997.

[68] R. Wattenhofer. *Distributed Counting: How to Bypass Bottlenecks.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 1998.

[69] R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. *Journal of Parallel and Distributed Computing*, 49(1):135–145, February 1998.

[70] J. C. Wyllie. *The Complexity of Parallel Computations.* PhD thesis, Department of Computer Science, Cornell University, 1979.

[71] P. C. Yew, N. F. Tzeng, and D. H. Lawrie. Distributed hot-spot addressing in largescale multiprocesors. *IEEE Transactions on Computers*, C-36(4):338–395, April 1987.