Advanced Distributed Algorithms

T. Herman and G. Tel

RUU-CS-93-37
November 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Advanced Distributed Algorithms

T. Herman and G. Tel

Technical Report RUU-CS-93-37
November 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0024-3275

Contents

Preface

A seminar on advanced distributed algorithms was held at Utrecht University during the Spring of 1993. The participants were assumed to have some background knowledge of network protocols and theory of distributed computing. As part of the seminar obligations, students had to write one or two summaries of the literature used in a lecture; the collection of summaries constitutes this syllabus.

The seminar covered four themes: efficient network protocols, fault-tolerant algorithms, stabilizing algorithms, and atomicity. The speakers of the seminar and the date of their presentation are as follows.

#	Date	Subject	Speaker
Part I: Efficient Network Protocols			
1	February 12	Match Making	Bryan Olivier
2	February 19	Sparser and Applications	Maarten Bodlaender
3	February 26	Network Synchronization	Hayo Baan
4	March 5	Resource control with tokens	Davar Rezania
5	March 12	Distributed Data Structures	Jaap-Henk Hoepman
Part II: Fault Tolerance			
6	March 19	Impossibility Results	Ted Herman
7	April 2	Robust Algorithms	Maarten Bodlaender
8	April 23	Conditions for solvability	Dick Alstein
Part III: Stabilizing Algorithms			
9	May 7	Mutual Exclusion	Davar Rezania
10	May 14	Stabilizing Graph Algorithms	Pascale van der Put
11	June 4	Stabilization Methodology	Hayo Baan
Part IV: Atomicity			
12	May 28	Wait free synchronization	K. Vidyasankar

From the table of contents and the list of speakers above, one can see that the scribe for each topic is different from the speaker. In this way was the understanding for the general themes further distributed. Perhaps this was also a contributing factor in the lively discussions that peppered the presentations. Some persistent lines of questioning during the presentations had to do with applicability of the ideas. This was, for example the case during the first set of presentations, which had efficiency as a primary concern. How large need a network be to justify some the approaches that are asymptotically efficient? In many cases, questions from the audience about technical details of message counting were answered "that's taken care of by the $O(\ldots)$". The question of applicability was also raised during the presentations on self-stabilization, where the audience observed that such systems do not (and

cannot) detect when the system has stabilized to correct (desired) behaviour. In a few cases, some fundamental questions about the nature of distributed computation were posed. The first presentation (Chapter 1) proposes a characterization of "distributed" that provoked some discussion. The presentation over impossibility results for fault-tolerant consensus (Chapter 6) provoked similar discussion, where we again asked if such results fundamentally characterize distributed computing.

The topics of the seminar were organized into four themes, which served to give the participants a sense of continuity in the research of a particular area. The chapters in this syllabus cover the individual topics presented, but some broad comments may serve to explain how these topics are thematically connected:

- **Network Protocols** (Part I). This part is not so much about network protocols in the classic sense of routing, reliable delivery, and such; but rather more generally this part's topics take, as model of distributed computing, a message-passing network of nodes and channels that in most cases is not fully connected. Given such a setting, what can be done to reduce communication costs? There are general approaches and specific tactics for given applications. The overriding theme that emerges is that efficient algorithms in such a distributed network can be (much as in ordinary sequential programming) developed with the aid of special data structures, which typically take the form of trees or other graph-theoretic structures. These distributed data structures and the accompanying operations on them often have pleasant combinatorics at the root of their efficiency. However, we did not come to a sense of a general methodology unifying the part's topics or one that suggests a systematic technique for obtaining new efficient algorithms.

- **Fault Tolerance** (Part II). In this part, faults are confined to be of the fail-stop process variety, as opposed to byzantine faults. The topics were chosen to introduce the questions: what cannot be done, what can be done, and is there something between these two extremes? The first topic (Chapter 6) demonstrates the main underlying difficulty in distributed computing, that faults and asynchrony are impossible to distinguish. The second topic (Chapter 7) shows that all is not lost, that in fact some problems do have fault tolerant solutions, though in some cases pure asynchrony is sacrificed. The third topic (Chapter 8) returns to impossible cases, but introduces a device to concentrate and bound the source of impossibility, suggesting directions for practical solutions that relax the conditions of impossibility (e.g. admit the use of time-outs).

- **Stabilizing Algorithms** (Part III). In this part, faults are in a sense the opposite of the previous part. Process do not fail in the fail-stop sense, but may exhibit temporary faulty behaviour; alternatively, this can be seen as an (externally caused) transient disturbance to the distributed computation, from which the algorithm must recover. The first topic (Chapter 9) of this part introduces the basic work in the area, illustrated by mutual exclusion algorithms. The second topic (Chapter 10) presents some graph algorithms, that nicely show how variant functions are useful for proving recovery following a transient fault. These first two topics also illustrate with specific algorithms a theme frequently encountered in distributed computing, that local computation can lead to desired global properties. The final topic (Chapter 11) considers the general question of stabilizing an arbitrary distributed computation, which rests on familiar snapshot and broadcast paradigms.

- **Atomicity** (Part IV). Alas, time did not permit us to cover this area to our satisfaction. We were fortunate to have an expert presentation by Professor Vidyasankar, who carefully gave an overview of the area.

We hope this syllabus will prove useful as a source of reference to the material discussed during the seminar.

Utrecht, Fall 1993,
Ted Herman, Gerard Tel.

Part I

Network Protocols

Chapter 1

Distributed Match-Making

Written by *J.-H. Hoepman.*

In many distributed algorithms a processor may contain information that at any time might be needed in the computation at another processor. This information may change in time, for instance due to local computations on a processor, or because the information moves from one processor to another. As an example of the last case, consider mobile telephone numbers. In such a telephone system every person will have her[1] own telephone number, that will not be bound to any specific physical location. If such a person moves (even from one town to another) she may keep the same telephone number.

There are many schemes to provide a processor with a recent picture of the information stored in the other processors. One solution is to let a processor send all relevant local information to all other processors whenever the local information changes. Then every processor has all the relevant data at hand. For a clique network with n processors this requires n messages per change of information, and $O(n)$ storage per piece of information from a single processor, resulting in $O(n^2)$ storage required for even the simplest of tasks. Another solution would be to broadcast requests for information. Since information may move from one processor to another, one has to request the information from all processors to be sure to find it. This means n messages per information request, but only $O(n)$ total storage.

Naturally there are other schemes to solve this problem more efficiently in both message and storage complexity. The distributed match-making paradigm introduced by Mullender and Vitanyi [MV88] seeks to unify all these schemes in one mathematical framework. Within this framework a trade-off between message and storage complexity is found, with respect to a new measure for the distributedness of the system.

1.1 The Distributed Match-Making Paradigm

Consider a distributed system as a point-to-point communication network described by an undirected graph $G = (V, E)$, with $|V| = n$ processors (numbered from 1 through n) and E the set of bi-directional communication links between two processors in V. Basically, the distributed match-making paradigm associates with every processor i two sets $W(i)$ and

[1] This is an experiment to avoid the clumsy he or she construction. This is not to imply any gender-bias.

$R(i)$, such that the intersection of the sets $W(i)$ and $R(j)$ for any pair of processors (i, j) is non empty. The processors in the intersection of $W(i)$ and $R(j)$ are called the rendezvous elements of the processors i and j (in that order). A specific choice of R and W is called a match-making strategy.

The idea behind match-making is that if processor i changes its local information, it will send *updates* to all processors in $W(i)$, which is hence called the write-set. If a processor needs some information it will send *requests* to all processors in $R(i)$, hence called the read-set. Since the intersection is non-empty, the processors in $R(i)$ together will have all information from all processors in the system available, and will send that information to i upon request. If a processor j obtains some information from a processor i, it is said that a *match* is made between i and j.

The distributed match-making paradigm is introduced by Mullender and Vitányi [MV88], where some non-standard assumptions are made to obtain a simple mathematical framework. In this framework a novel measure of the distributedness of a system is introduced, and a tradeoff between message complexity and storage complexity is derived with this distributedness as parameter. The assumptions are discussed in the next subsection, after which the main results of the paper are described. It is shown that many algorithms use the distributed match-making paradigm.

1.1.1 Assumptions

First of all, the distributed system is assumed to be completely connected (either because G is a clique, or because a lower network level provides routing), and the cost of sending a message is assumed to be the same for every sender-destination pair. So the match-making strategy does not normally take the topology of the network into account.

Second, the interpretation of storage complexity for the match-making algorithm is unclear. If we view a match-making algorithm as an algorithm to communicate information between two processors efficiently (i.e. as outlined in the previous section), we could define it as the amount of storage needed to store the information in all rendezvous elements in the distributed system. With this interpretation in mind, we note that Mullender and Vitányi assume that all processors contain exactly one piece of information, with size $O(1)$. In a more general approach we could assume that processor i owns (at most) α_i information items at any time, where each information item occupies $O(1)$ space. Let p be the total number of information items in the system. The approach taken by Mullender and Vitányi would then be a special case, with $p = n$ and $\alpha_i = 1$ for all i.

Third, the message complexity m of a match making algorithm is measured as the number of messages needed to establish a match between two processors, averaged over all ordered pairs of processors. Mullender and Vitányi [MV88] implicitly assume that the number of requests and the number of updates for a piece of information are roughly equal. Then the message complexity equals

$$m = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} (|W(i)| + |R(j)|) \tag{1.1}$$

The more likely case, in which the number of requests exceeds the number of updates, is called *weighted distributed match-making* and is treated by Kranakis and Vitányi [KV88, KV92].

Last but not least, fault tolerance is hardly addressed. In the derivation of the lower bounds it is assumed that the intersection of any pair $W(i)$ and $R(j)$ contains exactly one

element. It is noted that increasing the minimum number of processors in one such intersection to f will allow crash-failures of at most $f-1$ processors (assuming a clique network). It is not clear how the lower bound results will translate to this more general case; this is left as a subject for further study.

1.1.2 Main Results

The *distributedness* of a match-making strategy is the vector $\vec{k} = (k_1, \ldots, k_n)$, where k_i equals the number of times processor i occurs as rendezvous element for an ordered pair (j, l) (i.e. occurs in the intersection $W(j) \cap R(l)$). For ease of discussion, let us define a "norm"[2] on this vector as

$$\|\vec{k}\| = \left(\sum_{i=1}^{n} \sqrt{k_i}\right)^2 \tag{1.2}$$

The average message complexity for a particular strategy is defined as in equation 1.1. The paper proves the following theorem [MV88, Prop. 2], with $m(n)$ replaced by m[3].

Theorem 1.1 *For an n processor match-making strategy with distributedness $\vec{k} = (k_1, \ldots, k_n)$,*

$$m \geq \frac{2}{n}\sqrt{\|\vec{k}\|} \tag{1.3}$$

The average storage complexity is defined as

$$s_{\text{ave}} = \frac{1}{n}\sum_{i=1}^{n} |W(i)| \tag{1.4}$$

and the following bound on the average storage complexity is derived.

Theorem 1.2 *For an n processor match-making strategy with distributedness $\vec{k} = (k_1, \ldots, k_n)$,*

$$s_{\text{ave}} \geq \frac{\|\vec{k}\|}{n\sum_{i=1}^{n}(m - |W(i)|)} \tag{1.5}$$

Note that this bound can be simplified to

$$s_{\text{ave}} \geq \frac{\|\vec{k}\|}{n^2(m - s_{\text{ave}})}$$

by substitution of s_{ave} for its definition in the denominator. This yields the new bound

$$s_{\text{ave}} > \frac{\|\vec{k}\|}{n^2 m} \tag{1.6}$$

as $m - s_{\text{ave}} < m$. This bound is quite tight if we consider examples 1, 2, 3, and 4 of Mullender and Vitányi [MV88][4]. See also table 1.1, listing four common match making strategies together with their complexity measures. Here $\vec{k}$ equals the distributedness of the strategy, $\vec{s}$ represents the number of information items stored on a processor due to the match-making strategy, and $\vec{w} = (|W(1)|, \ldots, |W(n)|)$. The corresponding R and W sets can be found in the examples presented in [MV88]. One additional complexity measure s is also shown, representing the maximal storage required on a single processor due to the match-making strategy.

[2] This is not a norm in the mathematical sense, as the triangular inequality does not hold.

[3] The definition of $m(n)$ in [MV88] as the optimal message complexity achievable by any match-making strategy on an n processor system is unclear. It seems it doesn't allow the use of $\vec{k}$ in the proposition.

[4] Example 2 in [MV88] contains an error: the values of s and s_{ave} equal n (since $|W(i)| = n$ for all i).

	$\vec{k}$	$\vec{s}$	s	s_{ave}	$\vec{w}$	m
centralised	$(n^2, 0, \ldots, 0)$	$(n, 0, \ldots, 0)$	n	1	$(1, \ldots, 1)$	2
$\|\cdot\|$	n^2	n			n^2	
broadcast	$(n, \ldots, n)$	$(1, \ldots, 1)$	1	1	$(1, \ldots, 1)$	$n+1$
$\|\cdot\|$	n^3	n^2			n^2	
inv. broadcast	$(n, \ldots, n)$	$(n, \ldots, n)$	n	n	$(n, \ldots, n)$	$n+1$
$\|\cdot\|$	n^3	n^3			n^3	
symmetric	$(n, \ldots, n)$	$(\sqrt{n}, \ldots, \sqrt{n})$	$\sqrt{n}$	$\sqrt{n}$	$(\sqrt{n}, \ldots, \sqrt{n})$	$2\sqrt{n}$
$\|\cdot\|$	n^3	$n^2\sqrt{n}$			$n^2\sqrt{n}$	

Table 1.1: An overview of different match-making strategies.

1.2 Applications of Match-Making

Match-making is used in name servers, mutual exclusion algorithms, and information management; only name servers and mutual exclusion are discussed here.

1.2.1 Name Servers

Consider a distributed system where several objects are located at different processors. Each object supports a set of operations, called by users located at arbitrary nodes, but unaware of the location of the objects. Objects are allowed to move from one processor to another, for instance to balance the load on the system. A distributed name server keeps track of the objects, and provides users with the location of an object on which they want to perform an operation.

Name servers can be based on the distributed match making paradigm; given a match-making strategy, the server operates as follows. An object moving to (or created at) processor i posts its location and name to the processors in $W(i)$. A user on processor j requiring the location of an object will post a request with the name of the object to $R(j)$. Upon reception of a request, a processor will respond with the location of the object contained in that request, if it knows that location. The construction of the W and R sets guarantees that any object is found if it exists.

It must be noted that the above naive approach glosses over the fact that objects may move or even cease to exist while a match for that object is being made. Then the location returned by a request may be invalid. For a more thorough discussion of this problem we refer to Awerbuch and Peleg [AP93].

1.2.2 Mutual Exclusion

Maekawa [Mae85] presents a mutual exclusion algorithm based on the distributed match-making paradigm. The algorithm assumes the existence of a sets $S(i)$ of $O(\sqrt{n})$ processors for each processor i, such that $S(i) \cap S(j) \neq \emptyset$ for every pair (i, j). Mutual exclusion is achieved by having processor i lock the processes in $S(i)$ before entering the critical section.

1. To enter the critical section, processor i sends a Request message to all $j \in S(i)$, and waits until it has received Grant messages from all the processors it has sent a Request to.

2. If a processor receives a Request, it will reply with a Grant message as soon as any pending Grant message has been released by a Release message.

3. Upon completing the critical section, processor i sends a Release message to all $j \in S(i)$.

Since $S(i) \cap S(j)$ is non-empty, any two requesting processors need a Grant message from a processor in this intersection. Since it will not send this message to both simultaneously, this solution satisfies the mutual exclusion property.

The naive solution suffers from deadlock, as a processor may fail to obtain all grants it needs to enter the critical section. In order to avoid deadlock, all requests are send together with a timestamp as defined by Lamport [Lam78] and requests with the lowest timestamp are given preference—with processor ID's breaking ties—in the following manner:

1. If a processor receives a Request, this request is stored in a queue together with its timestamp. If this queue contains a smaller timestamp than the request the processor has granted now, it will try to reclaim the grant from the previous requesting processor by sending a Reclaim message.

   This message will either be replied with by a Release message (because the processor managed to get all the grants), or by a Relinquish message. In that case the request with the lowest timestamp in the queue is served with a Grant message and the previous request is put in the queue again (with its old timestamp).

2. If a processor receives a Reclaim message, it will respond with a Relinquish message if it has not yet received all the grants it needs and will forget it received a Grant message from the reclaiming processor.

A processor generates a new timestamp by taking the maximum over its previous timestamp and all timestamps received so far, increasing the result by one.

1.3 Extensions and Conclusion

1.3.1 Lighthouse Locate

A curious algorithm put forward by Mullender and Vitányi *lighthouse locate*, a randomized match-making strategy operating on a grid of processors. Let the distance between two neighbours in the grid be $\epsilon < 1$. A server will send out a random direction beam of length 1 every unit of time. This message will visit the first $\frac{1}{\epsilon}$ nodes that form the approximation of the straight line starting at the server with that random direction. The beam contains the information for which a match must be made, and any processor receiving such a beam will retain the information contained in it for d time units. Assume that the time required for the beam to travel a path of length 1 is so small in relation to d that the trail of the beam appears and disappears instantaneously.

To locate a server a client will do the following. It will repeatedly send out a random direction request beam, until this beam collides with a processor still retaining the required information. The length of the beam will vary, and the time between the sending of two consecutive beams varies accordingly. To maximize the probability that a match is found while keeping the costs low, the length and time-interval should be increased and decreased according to a certain well-chosen pattern. The sequence

$$0102010301020104010201030102010 5 \ldots$$

proposed in [MV88] implies that locating a server at distance d will take $O(e^d)$ time. It remains to be seen whether a more suitable sequence can be found. Analysis of this algorithm is challenging but lies outside the scope of this chapter.

1.3.2 Regional Matchings

Related to the distributed match-making paradigm is the concept of sparse partitions and, more specifically, regional matchings introduced by Awerbuch and Peleg [AP90]. Sparse partitions are used in the sparser described in Chapter 2.

Regional matching, like distributed match-making, associates two sets $W(i)$ and $R(i)$ with every processor. Unlike distributed match-making however, regional matching *does* take the topology of the network into account. Let us define $\mathcal{RW} = \{W(i), R(i) | i \in V\}$ and recall that $dist(u, v)$ equals the length of the shortest path from u to v.

Definition 1.3 *$\mathcal{RW}$ is called an* m-regional matching *(for some $m \geq 1$) if for all $i, j \in V$ such that $dist(i, j) \leq m$ we have $W(i) \cap R(j) \neq \emptyset$.*

So the read and write sets of two processors only need to intersect if the distance between both is less than or equal to m. This implies that these sets can in general be smaller than $\sqrt{n}$ for sufficiently small m. It is possible to construct an m-regional matching using the same algorithm `MAX-COVER` as used in the construction of sparse coarsening covers (see Awerbuch and Peleg [AP90]), starting with the cover consisting of the m-neighbourhoods of all nodes in V. The resulting coarsening cover is then converted to the required read and write sets by picking centers in the clusters of the coarsening cover. The write set of a node is set equal to the center of a cluster that contains the m-neighbourhood of that node. The read set of a node is set to the centers of all clusters to which it belongs.

The algorithm `MAX-COVER` takes an additional parameter k, and if the resulting coarsening cover is converted using the above procedure one will obtain an m-regional matching with the following properties:

$$\begin{array}{rclcrcl} \widehat{R_m}(i) & \leq & (2k-1)m & \qquad & \widehat{W_m}(i) & \leq & (2k-1)m \\ |W_m(i)| & = & 1 & & |R_m(i)| & \leq & 2ks \cdot n^{1/k} \end{array}$$

where

$$\widehat{R_m}(i) = \max_{j \in R_m(i)} (dist(i, j))$$

A hierarchy of regional matchings is used by Awerbuch and Peleg [AP93] to implement a distributed name server. It implements l levels—from 1 through $\lceil \log D(G) \rceil$—and for every level l a $z = 2^l$-regional matching with read and write sets $R_z(i)$ and $W_z(i)$ stored on every processor i. To locate an object, a user will start requesting at level 1 (i.e. all processors in $R_2(i)$) and continue to the next level if the object was not found. This strategy will ensure that less effort is spent in locating a nearby object. Moving an object from a processor i requires the sending of update information to all processors in the write set $W_{2^l}(i)$ of all levels l, even if the object moves to a nearby location. Special measures are taken to amortize this [AP93].

We can bound the number of messages needed to locate from processor i an object that resides at processor j using the name-server described above. Take $k = \log n$ and $d = dist(i, j)$.

Then the object will sure be found if a request is sent at level $\lceil \log d \rceil$. So the maximal number of messages exchanged is at most

$$\sum_{l=1}^{\lceil \log d \rceil} |R_{2^l}(i)| \cdot \widehat{R_{2^l}}(i)$$

Substituting the bounds we get

$$\sum_{l=1}^{\lceil \log d \rceil} (2k \cdot n^{1/k})(2k-1)2^l$$

which can be simplified to $4 \log n(2 \log n - 1)(2d - 2)$ resulting in $O(d(\log n)^2)$ worst case message complexity to locate an arbitrary object. This is only $O((\log n)^2)$ away from the optimal.

To move an object located at processor i to processor j, it must first be removed from all levels and then be reinserted at all levels by sending updates to all levels. This requires the exchange of at most

$$2 \sum_{l=1}^{\lceil D(G) \rceil} |W_{2^l}(i)| \cdot \widehat{W_{2^l}}(i)$$

messages. Substituting the bounds we get

$$2 \sum_{l=1}^{\lceil D(G) \rceil} (2k-1)2^l$$

yielding $O(D(G) \log n)$ worst case message complexity to move an object in the simple algorithm.

1.3.3 Conclusions

A simple mathematical framework has been presented for the distributed match-making paradigm. Although many of its assumptions are very strong, these can be weakened at the cost of losing simplicity in the model. Some algorithms and topics have been related to the distributed match-making paradigm, showing that distributed match-making is a fundamental concept in distributed computing.

Chapter 2

Sparser

Written by *D. Rezania.*

In this chapter we discuss a mechanism, called a *sparser*, that can be used to message-efficiently simulate distributed algorithms with a low *round complexity* (to be explained later). As an application of the technique, an $O(n^2 \log n)$-message algorithm for the all pairs shortest path problem is obtained. The method was proposed by Afek and Ricklin [AR92]; this is the main reference of this chapter.

2.1 Introduction

An extreme method to solve a network problem, is to collect all its input to one node, run a sequential algorithm on it, and then distribute the output to other nodes. For most applications, however, a truly distributed algorithm, which keeps the input and output of every node only in that node, is more efficient. There is also an intermediate approach, which sometimes yields more efficient distributed algorithms. The idea behind this technique is to partition the network into subsets of nodes, with a distinguished center in each set, and establish a simple path connecting the two central nodes of each pair of neighboring subsets. Each center simulates a given distributed algorithm on behalf of the nodes in its set.

In this way, all messages sent in the simulated algorithm between pairs of nodes in *the same* set are eliminated. On the other hand, a message sent from a node in one set to a node in another set is now sent from the center of the first set to the center of the other set, which may be more costly. The crux of the technique is to find efficient partitions (see Section 2.3) and to identify the communication patterns that can be simulated efficiently (Section 2.4).

Consider the cost of sending a message from one node to all its neighbors; it is bounded by the number of subsets in which the node has neighbors, times the distance between two centers. The existence of a suitable partition such that both the amount of intra cluster communication (for this communication pattern) and the radius of every cluster are small, is crucial to the approach.

An example where such a technique might be useful is the all pairs shortest paths problem, for which a solution with worst case message complexity $O(mn)$ was proposed by Segall [Seg83]. Using the sparser technique, an $O(n^2 \log n)$ message distributed algorithm for the all pairs shortest path problem is obtained.

2.2 Formal Definitions

Let $G = (V, E)$ be an undirected graph, describing the standard model of a point-to-point communication network; n denotes the number of nodes and m the number of edges. A vertex may communicate directly only with its neighbors, and a message between nonadjacent vertices is sent along some path connecting them in the network. For two vertices u, v in a graph G, let $dist_G(u, v)$ denote the (weighted) length of a shortest path in G between these vertices. A *routing scheme RS* for the network G is a mechanism for delivering messages in the network. The *memory requirement* of a protocol is the maximum amount of bits used by the protocol in any single processor in the network. The *communication cost* of transmitting a message over edge e, is the weight $w(e)$ of that edge. The communication cost of a protocol is the sum of the communication costs of all message transmissions performed during the execution of that protocol.

For a set of vertices U, the *neighborhood* of U is

$$\Gamma(U) = U \cup \{v | (v, u) \in E \wedge u \in U\}.$$

The *j-neighborhood* of a vertex $v \in V$ is defined as

$$N_j(v) = \{w : dist(w, v) \leq j\}.$$

Given a subset of vertices $R \subseteq V$, denote the *j-neighborhood cover* of R by

$$\mathcal{N}_j(R) = \{N_j(v) : v \in R\}.$$

The *diameter* of the network is

$$D = Diam(G) = \max_{u,v \in V}(dist(u, v)).$$

For a vertex $v \in V$ let

$$Rad(v, G) = \max_{w \in V}(dist_G(v, w)).$$

Let $Rad(G)$ denote the *radius* of the network, i.e.,

$$Rad(G) = \min_{v \in V}(Rad(v, G)).$$

A *center* of G is any vertex v realizing the radius of G (i.e., such that $Rad(v, G) = Rad(G)$).

Given a set of vertices $S \subseteq V$, let $G(S)$ denote the subgraph induced by S in G. A *cluster* is a subset of vertices $S \subseteq V$ such that $G(S)$ is connected. A *cover* is a collection of clusters $\mathcal{S} = \{S_1, ..., S_m\}$ such that $\bigcup_i S_i = V$. Given a collection of clusters $\mathcal{S}$, let $Diam(\mathcal{S}) = \max_i D$ $_i$) and $Rad(\mathcal{S}) = \max_i Rad(S_i)$. For every vertex $v \in V$ let $deg_{\mathcal{S}}(v)$ denote the degree of v in the hypergraph $(V, \mathcal{S})$, i.e., the number of occurrences of v in clusters $S \in \mathcal{S}$. The *maximum degree* of a cover S is defined as

$$\Delta(\mathcal{S}) = \max_{v \in V} deg_{\mathcal{S}}(v).$$

Given two covers $\mathcal{S} = \{S_1, ..., S_m\}$ and $\mathcal{T} = \{T_1, ..., T_k\}$ we say that $\mathcal{T}$ *subsumes* $\mathcal{S}$ if, for every S_i there exist a T_j such that $S_i \subseteq T_j$.

A set $\mathcal{P}$ is a *partition* of V if $\mathcal{P}$ is a set of pairwise disjoint nonempty sets and $\cup\mathcal{P} = V$. The elements of a partition are called *cells*. For a node $v \in V$ we define *degree of v relative to $\mathcal{P}$*, to be the number of cells in $\mathcal{P}$ which are at distance 1 or less from v. Formally:

$$\Delta_v(\mathcal{P}) = |\{P : v \in \Gamma(P), P \in \mathcal{P}\}|.$$

Let the *density* of a partition $\mathcal{P}$ be:

$$dens(\mathcal{P}) = \sum_{P \in \mathcal{P}} |\Gamma(P)| \left(= \sum_{v \in V} \Delta_v(\mathcal{P}) \right).$$

Definition 2.1 *A* sparser *is a pair $(\mathcal{C}, \mathcal{P})$ where $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ is a collection of clusters and $\mathcal{P} = \{P_1, P_2 \ldots, P_k\}$ is a partition of V, such that*

$$\forall i \in \{1, ..., k\} \ : \ \{P_i \subseteq C_i \wedge (\forall j \neq i)(P_j \not\subseteq C_i)\}.$$

Thus, in a sparser $(\mathcal{C}, \mathcal{P})$, each cell of $\mathcal{P}$ is covered by exactly one cluster of $\mathcal{C}$. A sparser whose density is d and radius r is an (r, d)-sparser.

2.3 Construction of the Cover

The following theorem is a main graph-theoretic result on which the construction of sparse covers and the simulation technique are based.

Theorem 2.2 *Given a graph $G = (V, E)$, $|V| = n$, a cover $\mathcal{S}$, and an integer $k \geq 1$, it is possible to construct a cover $\mathcal{T}$ that satisfies the following properties:*

1. *$\mathcal{S}$ subsumes $\mathcal{T}$,*
2. *$Rad(\mathcal{T}) \leq (2k-1)Rad(\mathcal{S})$,*
3. *$\Delta(\mathcal{T}) \leq 2k|\mathcal{S}|^{1/k}$.*

The theorem is proved by giving an algorithm to construct a cover with these properties.

The problem of constructing a subsuming cover is handled by reducing it to the subproblem of constructing a *partial cover*. The input of this problem is a graph $G = (V, E)$, $|V| = n$, a collection $\mathcal{R}$ of (possibly overlapping) clusters, and an integer $k \geq 1$. The output consists of a collection of *disjoint* clusters $\mathcal{DT}$ that subsume a subset $\mathcal{DR} \subseteq \mathcal{R}$ of the original clusters. The goal is to subsume "many" clusters of $\mathcal{R}$, while maintaining the radii of the output clusters in $\mathcal{DT}$ relatively small. The procedure `Cover(`$\mathcal{R}$`)` (Alg. 2.1) achieves this goal.

Procedure `Cover(`$\mathcal{R}$`)` starts by setting $\mathcal{U}$, the collection of *unprocessed* clusters, to $\mathcal{R}$. Then in each stage of the main loop, it constructs one cluster $Y \in \mathcal{DT}$, by merging some clusters of $\mathcal{U}$. The stage begins by arbitrarily picking an input cluster $\mathcal{T}$ in $\mathcal{U}$ and designating it as the kernel of the output cluster, to be constructed next. In the internal repeat loop the cluster is then repeatedly merged with intersecting input clusters from $\mathcal{U}$. At each such *internal iteration* the original cluster is viewed as the *kernel* Y, and the resulting cluster Z consists of all input clusters in $\mathcal{U}$ that intersect Y. For the next iteration Y is set to the current Z. The merging process performed by the internal loop carried repeatedly until reaching the appropriate *sparsity condition*. Upon reaching this condition the current output

$\mathcal{U} \leftarrow \mathcal{R}$ /* set of unprocessed input clusters */
$\mathcal{DR} \leftarrow \emptyset$ /* set of subsumed input clusters */
$\mathcal{DT} \leftarrow \emptyset$ /* set of subsuming output clusters */
repeat /* main stage */
  Select an arbitrary cluster S $\in \mathcal{U}$ /* initial kernel */
  $\mathcal{Z} \leftarrow \{S\}$
  repeat /* merge layers around kernel */
    $\mathcal{Y} \leftarrow \mathcal{Z}$
    $Y \leftarrow \cup_{S \in \mathcal{Y}} S$
    $\mathcal{Z} \leftarrow \{S |\ S \in \mathcal{U}, S \cap Y \neq \emptyset\}$
  until $|\mathcal{Z}| \leq |\mathcal{R}|^{1/k}|\mathcal{Y}|$ /* sparsity condition */
  $\mathcal{U} \leftarrow \mathcal{U} - \mathcal{Z}$
  $\mathcal{DT} \leftarrow \mathcal{DT} \cup \{Y\}$
  $\mathcal{DR} \leftarrow \mathcal{DR} \cup \mathcal{Y}$
until $\mathcal{U} = \emptyset$
Output($\mathcal{DR}$,$\mathcal{DT}$).

Algorithm 2.1: THE PROCEDURE `cover`($\mathcal{P}$).

cluster is finalized to be the kernel Y of the resulting cluster Z, and the procedure adds Y to the output collection $\mathcal{DT}$. Also every input cluster in the collection $\mathcal{Y}$ is added to $\mathcal{DR}$ and every input cluster in the collection $\mathcal{Z}$ is removed from $\mathcal{U}$. The main stage proceeds until $\mathcal{U}$ is exhausted.

Lemma 2.3 *Given a graph $G = (V,E)$, $|V| = n$, a collection of clusters $\mathcal{R}$, and an integer k, the collections $\mathcal{DT}$ and $\mathcal{DR}$ constructed by procedure* `Cover`($\mathcal{R}$) *satisfy the following properties:*

1. *$\mathcal{DT}$ subsumes $\mathcal{DR}$,*
2. *$Y \cap Y' = \phi$ for every Y, $Y' \in \mathcal{DT}$,*
3. *$|\mathcal{DR}| \geq |\mathcal{R}|^{1-1/k}$, and*
4. *$Rad(\mathcal{DT}) \leq (2k-1)\ Rad(\mathcal{R})$.*

Proof. First note that since the elements of $\mathcal{U}$ at the beginning of the procedure are clusters, the construction process guarantees that every set Y added to $\mathcal{DT}$ is a cluster. Property (1) now holds directly from the construction.

To prove property (2), suppose, seeking to establish a contradiction, that there is a vertex v such that $v \in Y \cap Y'$. Without loss of generality, suppose that Y was created earlier than Y'. Since $v \in Y'$, there must be a cluster S' such that $v \in S'$ and S' was still in $\mathcal{U}$ when the algorithm started the main stage constructing Y'. However, every such cluster S' satisfies $S' \cap Y = \phi$. Therefore, in the main stage that created Y, the construction step creating the collection $\mathcal{Z}$ from Y at the last internal iteration should have added S' in to $\mathcal{Z}$ and eliminated it from $\mathcal{U}$, a contradiction.

$\mathcal{R} \leftarrow \mathcal{S}$ /* $\mathcal{R}$ is a collection of remaining (unsubsumed) clusters */
$\mathcal{T} \leftarrow \emptyset$ /* $\mathcal{T}$ is the output cover */
repeat
  $(\mathcal{DR}, \mathcal{DT}) \leftarrow$ Cover($\mathcal{R}$) /* invoke procedure **Cover** */
  $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{DT}$
  $\mathcal{R} \leftarrow \mathcal{R} \backslash \mathcal{DR}$
until $\mathcal{R} = \emptyset$

Algorithm 2.2: THE MAIN ALGORITHM.

Property (3) is derived as follows. It is immediate from the sparsity condition that the resulting pair $\mathcal{Y}$, $\mathcal{Z}$ satisfies $|\mathcal{Z}| \leq |\mathcal{R}|^{1/k}|\mathcal{Y}|$. Therefore

$$|\mathcal{R}| = \sum_{\mathcal{Z}} |\mathcal{Z}| \leq \sum_{\mathcal{Y}} |\mathcal{R}|^{1/k}|\mathcal{Y}| = |\mathcal{R}|^{1/k}|\mathcal{DR}|,$$

which proves property (3).

Finally, to analyze the increase in the radius of clusters in the cover, consider some main stage of procedure cover in Alg. 2.1, starting with the selection of some cluster $S \in \mathcal{R}$. Let J denote the number of internal iterations executed during this main stage. Denote the initial set $\mathcal{Z}$ by $\mathcal{Z}_0$. Denote the set $\mathcal{Z}$ (respectively, Y, $\mathcal{Y}$) constructed on the i^{th} internal iteration ($1 \leq i \leq J$) by $\mathcal{Z}_i$ (respectively, $Y_i, \mathcal{Y}_i$). Note that for $1 \leq i \leq J$, $\mathcal{Z}_i$ is constructed on the basis of $\mathcal{Y}_i$, $\mathcal{Y}_i = \mathcal{Z}_{i-1}$ and $Y_i = \cup_{S \in \mathcal{Y}_i} S$.

Claim 2.4 *$|\mathcal{Z}_i| \geq |\mathcal{R}|^{i/k}$ for every $0 \leq i \leq J-1$, and strict inequality holds for $i \geq 1$*

Corollary 2.5 *It holds that $J \leq k$*

Claim 2.6 *For every $1 \leq i \leq J$, $Rad(Y_i) \leq (2i-1)Rad(\mathcal{R})$.*

The proof of claims 2.4 and 2.6 is by induction on i. It follows from Corollary 2.5 and Claim 2.6 that

$$\text{Rad}(Y_i) \leq (2k-1)\text{Rad}(\mathcal{R}),$$

which completes the proof of the last property of the lemma. □

In order to construct a cover as in Theorem 2.2, one should use Algorithm 2.2. The input to the algorithm is a graph G, a cover $\mathcal{S}$ and an integer $k \geq 1$. The partial cover $\mathcal{DT}$ constructed by the procedure consist of *disjoint* clusters, which ensures that each phase contributes at most one to the degree of each vertex in the output cover $\mathcal{T}$.

The above algorithm, proposed by Awerbuch and Peleg [AP92], is a sequential algorithm for constructing a $(z, n^{1+1/z})$-sparser, z integer. One can extend this algorithm to a distributed algorithm with message complexity $O(m + n \cdot z)$, by using the standard **DFS** and **BFS** techniques. Moreover, the distributed implementation readily produces the following additional structures: distinguishes a center in each cluster and construct a breadth-first search tree spanning each cluster, rooted at the center. Each pair of neighboring cells selects one inter-cell link, called the *preferred link*. The collection of breadth-first search trees and the preferred links is the structure spanning the network that is used by the simulation to pass messages between centers. All of this is achieved with the same message complexity, $O(m + n \cdot z)$.

2.4 The Simulation Technique

Assuming that an (r, d)-sparser $(\mathcal{C}, \mathcal{P})$ is given in the network, the sparser simulation technique proceeds in three major steps. First, collect the topological information (including the identity of cells incident to nodes in the cell and links leading to these cells) of all the nodes in each cell in $\mathcal{P}$ to the center of the cluster that covers the cell. Second, simulate the algorithm by exchanging messages between the center nodes. Third, the outputs of the algorithm are distributed by the center of each cell to nodes in the cell along a breadth-first search tree. The first step requires $O(rm)$ messages since information about $m = |E|$ links is sent to distance r. If the size of the output of the algorithm at each node is $O(y \log n)$ bits, the third phase, costs $O(yrn)$ messages. A distributed asynchronous algorithm proceeds at each node in cycles of three steps: (1) *message receipt*, (2) *local computation of a new local state*, and (3) *message transmission to a subset of neighbors.* Call a distributed algorithm in which nodes in the third step of each cycle send either the same message to all their neighbors, or a message to only one neighbor, or no message at all, a *type-α* algorithm.

Definition 2.7 *The* cycle complexity *of an asynchronous algorithm A, is the maximum, over all nodes, of the number of cycles a node goes through during the execution of the algorithm in the worst case.*

Lemma 2.8 *Given a network with an (r, d)-sparser and a type-α distributed algorithm A of cycle complexity K, then A can be simulated in the network in $O(mr + Kdr + yrn)$ messages, where $y \log n$ is the size of the output of the algorithm at each node in bits.*

Proof. The term $O(yrn)$ is the cost of distributing the output of A to the nodes. The term $O(mr)$ is the cost of collecting the topology of the neighborhood of each cell in the center to the cell. In each cycle of computation at node v the simulation sends at most $2r + 1$ messages for each neighboring cell of v. Thus at most $\sum_v \Delta_v(\mathcal{P}).(2r + 1)$ messages are sent, per cycle, resulting in a total $O(Kdr)$ over the entire run of the algorithm. □

2.5 Example: All Pairs Shortest Paths

Consider the following algorithm for the all pairs shortest paths problem, proposed by Segall [Seg83].

Each node starts by sending its identity to all its neighbors. When a node receives the identity of all its neighbors, it marks them to be in distance one from it, and starts the second step. In the i^{th} step a node builds a message which consists of all the nodes which it marked to be at distance $i - 1$ from it, and sends the message to all its neighbors. The node waits until it receives all the messages sent by its neighbors during the i^{th} step, marks the nodes which it receives their identity for the first time to be in distance i from it, and pass to the next step. A node terminates when it receives no new identities in a round.

The message complexity of Segall's algorithm is $\Theta(n \cdot m)$ because every identity is set over each link once in each direction.

Theorem 2.9 *The upper bound on the message complexity of the unweighted all pairs shortest paths problem is $O(n^2 \log n)$ messages (each message is of size $O(\log n)$ bits).*

Proof. The cycle complexity of Segall's algorithm is $O(n)$ because a node must receive at least one new identity in order to proceed to the next round. The size of the output at each node is also $O(n)$, because the output consists of a list of n destinations, each with a distance (and preferred neighbor for this destination).

From Lemma 2.8 it follows that, using a $(\log n, n)$-sparser, Segall's algorithm can be simulated in $O(n^2 . \log n)$ messages. □

Chapter 3

Synchronizers

Written by *P.I.A. van der Put.*

Synchronous algorithms are usually simpler to understand and more efficient than asynchronous algorithms. The design and analysis of a synchronous algorithm is less complicated. So the problem of simulating a synchronous algorithm by an asynchronous algorithm is investigated.

A simulation technique, referred to as a *synchronizer*, is proposed by Awerbuch [Awe85] and is subject of this chapter. A particular synchronizer called γ constructs low-complexity asynchronous algorithms from good (i.e. low-complexity) synchronous algorithms.

3.1 Model

In an asynchronous network a node sending a message hands the message over to the communication subsystem and proceeds with its own local task. The communication subsystem delivers the message at its destination after some finite (but unbounded) time. In a synchronous network the existence of a global clock is assumed, generating pulses in every node simultaneously. A node sending a message hands the message over to the communication subsystem just when a clock pulse is generated. The communication subsystem delivers the message at its destination before the next pulse.

Each node has a distinct identity. Links have distinct identities derived from the node identities.

Each message contains a bounded amount of information. Messages do not become lost or duplicated. Messages may arrive in a different order than the order in which they were sent. A node may send messages to any of its neighbors.

Each message arrives after a finite delay, but there is no a priori bound on the propagation delay. The inter-message delay is defined as the maximal time between the sending of two consecutive messages on the same link.

One time unit in a synchronous network is defined as the time between two pulses (or rounds). One time unit in an asynchronous network is defined as the maximum propagation delay of any message. So one time unit in an asynchronous network differs with each execution.

The time for computations at the nodes of the network is not taken into account. Only the time of transmission of the messages is considered. So one time unit serves as an upper

bound on the propagation delay in a particular execution of an algorithm which is either asynchronous or synchronous.

3.1.1 Safety

A node in a (simulated) algorithm is *safe* with respect to a certain pulse if each message sent by that node at that pulse has already arrived at its destination. In order for a node to find out that it is safe extra messages are added. So an ACK message is sent in response to each message of the original synchronous algorithm.

A new pulse may be generated at a node whenever all neighbors of that node are safe with respect to the previous pulse. Note that the node itself may not be safe yet.

3.1.2 Complexity measures

The communication and time complexities of algorithms are expressed in terms of the number of messages and time units. The overhead introduced by a synchronizer is expressed explicitly.

C_s Communication complexity of a synchronous algorithm. This is the number of messages used during an execution of the algorithm.

C_a Communication complexity of an asynchronous algorithm. This is the number of messages used during an execution of the algorithm.

$C(v)$ Communication requirements added by synchronizer v per each pulse of the synchronous algorithm.

$C_{init}(v)$ Communication complexity of the initialization phase of synchronizer v.

T_s Time complexity of a synchronous algorithm. This is the number of pulses (or rounds) used during an execution of the algorithm.

T_a Time complexity of an asynchronous algorithm. This is the number of time units used during an execution of the algorithm.

$T(v)$ Time requirements added by synchronizer v per each pulse of the synchronous algorithm.

$T_{init}(v)$ Time complexity of the initialization phase of synchronizer v.

Synchronizers may need an initialization phase so the communication and time complexities of the initialization phase are taken into account. But the overhead per pulse is more important.

The combination of synchronizer v with a synchronous algorithm S results in an asynchronous algorithm A. The total complexity of A depends on the overhead introduced by v.

$$\begin{aligned} C_a &= C_{init}(v) + T_s \cdot C(v) + C_s \\ T_a &= T_{init}(v) + T_s \cdot T(v) + T_s \end{aligned} \tag{3.1}$$

Note that acknowledgement of all messages of the original synchronous algorithm does not increase the communication (or time) complexity asymptotically, but with a constant factor, 2.

3.2 Synchronizer α

Each node sends a SAFE message to all its neighbors when it finds out that it is safe. The next pulse is generated at a node when it has received SAFE messages from all its neighbors. Note that a node can receive SAFE messages from all its neighbors before sending its own SAFE messages.

No initialization is needed for synchronizer α, hence $C_{init}(\alpha)$ and $T_{init}(\alpha)$ are 0. The α synchronizer has a wasteful communication complexity, but an favorable time complexity.

$$\begin{aligned} C(\alpha) &= O(|E|) \\ &= O(|V|^2) \\ T(\alpha) &= O(1) \end{aligned} \tag{3.2}$$

3.3 Synchronizer β

The synchronizer β has an initialization phase, in which a leader r and a spanning tree with root r are computed.

When node q detects that all nodes in its subtree are safe q sends a SAFE message to its father. Node q detects that all its descendants are safe by receiving SAFE messages from all its children. This mechanism is called *convergecast.*

Node r detects that the entire tree is safe by detecting it is safe itself together with receiving SAFE messages from all its children. The node r responds by sending a PULSE message to all its children. Any other node sends a PULSE message to all its children when it receives a PULSE message from its father. This mechanism (of sending PULSE messages) is called *broadcast* along the spanning tree.

Node r detects that all its neighbors are safe by means of the convergecast mechanism (because all its neighbors are children). Any other node detects that all its neighbors are safe when it receives a PULSE message from its father (because this implies that all nodes are safe).

The communication (and time) complexity during initialization depends on what algorithm is used for computing the spanning tree (seeGallager *et al.* [GHS83]).

The synchronizer has an favorable communication complexity, but a wasteful time complexity.

$$\begin{aligned} C(\beta) &= O(|V|) \\ T(\beta) &= O(|V|) \end{aligned} \tag{3.3}$$

The time complexity added per pulse is linear in the depth of the spanning tree. So by choosing an appropriate spanning tree a time complexity of $O(|D|)$ is possible.

3.4 Synchronizer γ

The set of nodes of the network is partitioned into *clusters* by means of a particular partition algorithm during the initialization phase of the synchronizer.

The partition algorithm starts by selecting a node which is to be leader of the first cluster. The partition algorithm creates clusters around chosen leader nodes. New leader nodes and

clusters are created until all nodes are covered. Any two neighboring clusters communicate via a specific link called a preferred link. The preferred links are chosen during and after the creation of the clusters.

Synchronizer β is applied within each cluster. Synchronizer α is applied among the clusters.

Sending SAFE messages to all neighboring clusters (due to synchronizer α) is represented by a broadcast of CLUSTER-SAFE messages along the spanning tree and via the preferred links. Receiving SAFE messages from all neighboring clusters is represented by a convergecast of READY messages along the spanning tree.

Synchronizer α can be seen as an (extreme) example of the synchronizer γ. Each node in the network forms a cluster. Synchronizer β can be seen as an (extreme) example of the synchronizer γ. The entire network forms one cluster.

A partition p (of clusters) is characterized by E_p and H_p, where

E_p is the set of links in spanning trees and preferred links, and

H_p is the maximum of height of any spanning tree.

For a given parameter k with $2 \leq k \leq |V|-1$ the partition algorithm computes a partition p with $|E_p| = k \cdot |V|$ and $H_p = \log_k |V|$. The use of the partition algorithm leads to the following communication complexity and time complexity of the initialization phase of the γ synchronizer.

$$\begin{aligned} C_{init}(\gamma) &= O(k \cdot |V|^2) \\ T_{init}(\gamma) &= O(|V| \cdot \log_k |V|) \end{aligned} \tag{3.4}$$

The added communication complexity and time complexity of synchronizer γ depend on the partition itself.

$$\begin{aligned} C(\gamma) &= O(|E_p|) \\ T(\gamma) &= O(H_p) \end{aligned} \tag{3.5}$$

3.5 Discussion

Comparing the time complexities of algorithms is hard due to the nature of the definition of one time unit. However, the following may be stated. The time complexity of a synchronous algorithm simulated by means of synchronizer γ is only marginally bigger than the time complexity of the original synchronous algorithm.

The synchronizer γ is an optimal synchronizer (in comparison to the synchronizers α and β) when considering the trade-off between communication complexity and time complexity.

Applications

Simulating a synchronous algorithm by means of a synchronizer is only useful if there does not exist any asynchronous algorithm with improved communication and time complexities. An example of such a useful application of a synchronizer is shown. Asynchronous algorithms that solve the bread-first search problem were proposed by Gallager [Gal82]. One of these asynchronous algorithms has a communication complexity of $O(|V| \cdot |E|)$ and a time complexity

of $O(|V|)$. Another one has a communication complexity of $O(|V|^{2+x})$ and a time complexity of $O(|V|^{2-2x})$ for a given x with $0 \leq x \leq 0.25$. An algorithm that solves the breadth-first search problem in a synchronous network was given by Eckstein [Eck77]. The algorithm has a communication complexity of $O(|E|)$ and a time complexity of $O(|V|)$. Simulation of this synchronous algorithm by means of the synchronizer γ results in an asynchronous algorithm which has a communication complexity of $O(k \cdot |V|^2)$ and a time complexity of $O(|V| \cdot \log_k |V|)$ for a given k with $2 \leq k \leq |V| - 1$.

A synchronizer may also be used in the case of distributed maximum flow algorithms (see Awerbuch [Awe85]).

Combining with the sparser

Combinations of the sparser (see chapter 2 and Afek and Ricklin [AR92]) and a synchronizer may be considered. One assumption in the model of the sparser should be taken into account. It is assumed that communication in a network is divided about equally among all links in the network. An asynchronous algorithm which is the result of synchronizer γ may not meet this requirement because most of the communication takes place along the preferred links. The application of synchronizer α followed by the application of the sparser may result in an asynchronous algorithm which resembles the result of applying synchronizer γ.

Chapter 4

Token-Based Management

Written by *M. P. Bodlaender.*

In this chapter solutions to two quite different problems are presented, yet the algorithms have a technique in common. The problems of allocating local disk buffers and the use of global resources are both solved using tokens, which are passed through the network. Possession of a token typically allows a process to continue its computation for some bounded number of steps. These algorithms were originally proposed by Afek *et al.* [AAPS87] and Afek, Kutten, and Peleg [AKP91].

The standard assumptions of an asynchronous point-to-point communication network are made, e.g., as by Galagger *et al.* [GHS83]. The network topology is described by an undirected *communication graph* $G = (V, E)$, where V represents the processors in the network and E represents the bi-directional communication channels between neighboring processors. There is no common memory, no common clock. Messages sent over a link incur an arbitrary but finite *time delay.* One unit of time is defined as the maximal time delay of any message in the network.

4.1 Deadlock-Free Packet Switching

This section deals with the design of deadlock-free routing schemes with small overhead in communication and space.

Processors in the network introduce packets which must be forwarded by the network to their destinations somewhere in the network. The processors along the route followed by the packet have to temporarily store the packet in a buffer. If the network becomes congested with packets, free bufferspace will become sparse and deadlocks may arise.

In the algorithm now presented, a packet that is delivered to its destination permits some node (not necessarily its source or destination) to introduce a new packet in the network. This restriction comes from the observation that if more packets are introduced in the network than are delivered, delay times and buffer requirements grow to infinity.

The *inherent communication requirement* of a packet is the distance from a packets source to its destination. The *actual communication* of a protocol is the actual number of messages sent during execution. The *communication overhead* of a routing protocol is its worst case ratio of actual to inherent communication. The maximum is taken over all possible sequences of message arrival histories. Space is measured by the number of buffers used in each node.

4.1.1 Algorithm Overview

Each node is internally organized as a hierarchy of d levels, where $d = \log n + 1$. Every node has $3d$ buffers, partitioned into d levels, with level i owning two *regular buffers* A_i and B_i and an *elevator buffer* E_i for $1 \leq i \leq d$.

The packet is packed in a token T (in level 1), whose task is to carry the packet to its destination and terminate. A token in level i proceeds using only buffers of level i.

When the token cannot proceed (because no buffer of level i is free in the next node), it "merges" with another stuck token in the same node (A_i and B_i). One of them is promoted to proceed in buffers of level i+1, and is placed in buffer E_{i+1}. Thus a token in level i represents 2^i packets. Thus, the higher the level, the less congested the buffers are. The other token remains locked in the node at level i, until it receives permission to continue (after the first token has reached its destination) in level $i + 1$. However, the two buffers A_i and B_i remain locked until the second token has reached its destination, and a second permit arrives. The task of freeing buffers is tricky. It would seem the permit messages would themselves use buffers, and again deadlock may arise. An attempt to solve this problem by the same strategy may result in increasing the communication complexity (a stretch in communication overhead of $\log |V|$). To overcome the problem presented above, the releasing paradigm is relaxed. A permit is now viewed as a "general purpose currency", or check, rather than a dedicated process. So it is no longer required that a level i permit generated by some token T frees the particular token T' with whom T was merged. The permit can be used to free any other token that happens to be locked at level i on the permit's way.

The check approach implies the following modifications. Instead of releasing seperate tokens, we have counters Debt_e on every edge e, for every level. These counters specify "how much freeing work to be done in direction of e". Simularly we have counters (credits) specifying "how much freeing work" we are permitted to do on every level. We also have a counter on an "internal link", which is used for credits to cross level boundaries. Every credit on level l is worth 2 credits on level $l - 1$. If a credit is send over the internal link at level 0 at some node, a new packet may be introduced in the network by that node.

When a process has more than 0 credits, it first tries to free packets stuck in its node, then sends the remaining credits across a *fairly selected* link e with $\text{Debt}_e > 0$.

4.1.2 Complexity and Correctness

The analysis given in the article is quite involved so for brevity only the results will be stated here.

Proposition 4.1 *At no time does the algorithm require a buffer of level larger than d.*

Proof. This follows from the fact that one token in level i represents 2^i packets and that at most 2^d packets are underway in the network at any one time. □

Lemma 4.2 *At any time, if two tokens wish to perform a merge operation at level i, then the elevator buffer E_{i+1} is empty.*

Proof. If E_{i+1} were not empty, buffers A_i and B_i would still be "locked" and at most one message could occupy one of them. □

Lemma 4.3 *At any moment, eventually every node is allowed to enter a new packet into the network.*

Lemma 4.4 *The communication overhead of the hierarchical algorithm is O(1).*

Theorem 4.5 *The algorithm can be extended so that with the same communication overhead and number of buffers, every packed is delivered within $O(n^2 \log n)$ of the time it entered the network.*

4.2 Resource Management

In this section the *Resource Controller* is introduced. A Resource Controller deals with the problem of controlling the total amount of resources a distributed algorithm consumes.

4.2.1 Definition of a Resource Controller

A Resource Controller is a distributed algorithm that executes concurrently with an algorithm whose resource consumption is being controlled, the *controlled algorithm.* Each unit of resource used by the controlled algorithm must first be authorised by the Resource Controller. A Resource Controller with parameters M and W satisfies the following requirements:

1. At most M resources are authorised for use.
2. The controller terminates by producing a signal at a distinguished node if and only if a request for a resource cannot be authorised anymore.
3. If the controller terminates then at least $M - W$ requests were authorised.

The following assumptions are made for simplicity :

- The controlled algorithm is a single initiator distributed algorithm with new nodes dynamically joining the algorithm.
- The graph induced by the participating nodes is a tree rooted at the initiator. Every node v knows the length of its path to the root: $D(v)$.

4.2.2 Basic Controller

In this section a Resource Controller called the *Basic Controller* is decribed. The Basic Controller operates under the assumption that an upper bound U on the number of nodes is known. Each node v is assigned a level $l_v = \max\{i : 2^i | D(v)\}$ and a supervisor Super(v), which is either the root of the spanning tree, or the closest ancestor of v whose level is $l_v + 1$, whichever is closer.

Lemma 4.6 *1. The number of nonempty levels is at most* $\log U$.

2. If v is at level l then the path from v to Super(v) has either 2^{l_v} or $3 * 2^{l_v}$ links.

3. The number of supervisors at level l is at most $U/2^{l-1}$.

Proof. The first two parts follow from the definition. To prove the third part, for each supervisor v at level l, consider a path from u to v where v is the supervisor of u. Clearly, these paths are disjoint and each contains at least 2^{l-1} vertices. □

Algorithm Overview

Every node has a procedure *Resource-Request*, every node except the root has a process called *Delivery Process*. The root has the *Root Process*.

Each node has two *bins* where permits can be stored, one managed by the Delivery Process, and one for the Resource-Request procedure. The bin capacity of the Delivery Process located at node v, denoted Cap(v), is max(1, Cap(Super(v))/2). The bin capacity of the Resource Request procedure is max(1, Cap(w) where $l_w = 0$). To ensure that at most W resources are wasted, the capacities are chosen in such a way that the sum of all bins is at most W.

Initially all bins are empty except the root-bin, which contains M permits. When a procedure Resource Request is called at some node v, it issues a permit if it has one in its bin. Otherwise, it asks the Delivery Process at v to relay the request to replenish the bin to the nearest node at level 0 up the spanning tree.

In general, if the bin of the Delivery Process at node v is empty when it receives a request, it sends a request to the Delivery Process of its supervisor for enough permits to replenish its bin. When it has filled its own bin, it sends the permits previously requested. Each node on the path from v to Super(v) serves as a relay for communication between v and Super(v).

The Root Process is slightly different. If its bin is empty, it terminates the algorithm and signals all nodes.

Analysis of the Basic Controller

Lemma 4.7 *The message complexity of the Basic Controller is* $O(\frac{M}{W} U \log^2 U)$.

Proof. Follows from the observations that
(1) the number of messages is bounded by the number of permit messages; (2) the total number of permit messages received at level l_w is at most $M/\text{Cap}(w)$ and these messages travel at most $3 \cdot 2^{l_w}$ links;
and (3) there are at most $\log U + 1$ levels. □

Lemma 4.8 *At least $M - W$ resource requests are satisfied if any request is blocked.*

Proof. Consider the moment the last permit message is received. A request was blocked, so the root-bin is empty. Now every permit has either been used, or is stored in a bin. But the total bin size is at most W, so $M - W$ permits have been used. □

Corollary 4.9 *The bit message complexity of the Basic Controller is* $O(\frac{M}{W} U \log^2 U \log\log U)$ *and memory requirements are* $O(\log\log U)$ *per link.*

4.2.3 Main Controller

In the section above the Basic Controller was defined, which assumed an a-priori knowledge of an upper bound on the number of participating nodes. The *Main Controller* defined below relaxes that assumption and its complexity depends only on the actual number of participating nodes.

The basic idea is to run two Basic Controllers concurrently. One, the *Controller-N* monitors and controls the number of participating nodes. The second controller, called *Controller-R* monitors and controls the resource consumption. Thus it is required that a new node that joins the protocol will not participate in it until it gets a permit from Controller-N.

The algorithm proceeds in iterations. Assume the number of participating nodes is V_i at the beginning of every iteration i. The number of nodes that are allowed to join during iteration i is at most twice the number at the beginning of the iteration, so both Basic Controllers assume an upper bound of $3V_i$ nodes. To enforce this, (M, W) of Controller-N is set to $(3V_i, V_i)$ at the beginning of every iteration. Thus Controller-N terminates when the number of nodes has at least doubled. When Controller-N terminates at the root, it executes a "broadcast and echo" to collect the contents of all bins (both Controller-N and Controller-R bins). When the echo terminates, V_{i+1} and the number of unused resources M_{i+1} are known at the root. If $M_{i+1} \leq W$ the algorithm terminates, otherwise both controllers are restarted with bin sizes corresponding to the new upper bound $3V_{i+1}$, and with Controller-R using (M_{i+1}, W) as its parameters.

Analysis of the Main Controller

Correctness follows from the correctness of the Basic Controller and from the fact that at any point we maintain a constant approximation to the number of participating nodes.

Theorem 4.10 *The message complexity of the Main Controller is* $O(\frac{M}{W} V \log^2 V)$ *and the bit compexity is* $O(\frac{M}{W} V \log^2 V \log\log V)$.

Proof. The message complexity of the Main Controller is the sum of the message complexity of Controller-N and Controller-R. In iteration i, these complexities are $O(V_i \log^2 V_i)$ and $O(\frac{M}{W} V_i \log^2 V_i)$. By definition $M \geq M_i \geq W$. Moreover, for each i, $3V_{i-1} \geq V_i \geq 2V_{i-1}$. The bound follows. □

Chapter 5

Distributed Data Structuring

Written by *H. Baan.*

Efficient algorithms depend on the use of efficient data structrures. This is also the case in distributed systems, and in fact distributed data structures are widely used in distributed systems. In many cases such structures are not constructed for the use in a particular algorithm, but for the use as autonomous building blocks of various permanent storage and retrieval mechanisms, such as distributed dictionaries, name servers in communication networks, bulletin boards, resource allocation managers etc. The common function of all these mechanisms is supplying to the potential users the facilities for storing and retrieving the information throughout the system.

Peleg [Pel90] studies the problem of designing, implementing and operating a data structure in a distributed system from a complexity oriented point of view. This is done by the dictionary data structure for which different distributed solutions are tailored.

5.1 Modeling

The models used for the distributed system and the distributed data structure, are given below.

5.1.1 Distributed System

The model that is used for the distributed system is the well known asynchronous point-to-point network with $G = (V, E)$ and $n = |V|$. The length (ie. the number of edges) of a shortest path between u and w in G is denoted $dist_G(u, w)$. The degree of a vertex v is denoted $deg_G(v)$ and the diameter of the graph is denoted $D(G)$.

Furthermore, the system is assumed to have 1) *broadcast*, 2) *convergecast* (ie. the possibility for a processor to collect information of all other processors in the system) and 3) *routing* capability. These operations are assumed to be implemented with the following time and communication complexity measures:

	T_{op}	C_{op}
broadcast	$O(D(G))$	$O(n)$
convergecast	$O(D(G))$	$O(n)$
routing from u to w	$O(dist_G(u,w))$	$O(dist_G(u,w))$

5.1.2 Distributed Data Strutcture

All data items X have the form $X = (Key_X, Record_X)$, and come from the domain $\mathcal{X}$. Each data item uses δ bits of memory, and all keys Key_X come from the same domain $\mathcal{K}$ which is an ordered set.

The number of distinct data items in the structure is m, M is the total amount of memory bits used in the system (this can be more than $m\delta$ if duplicates of data items exitst) and $\hat{M}$ denotes the maximum memory usage at a single site taken over all sites. It is assumed that $m \geq n$, and in most cases it will be that $m \gg n$.

Because it can not be expected that a single site is able to hold all data items, some balancing of memory usage over the sites is required. The optimal ratio of memory balancing is $Load = \lfloor \frac{m}{n} \rfloor$. Redundancy is measured by ρ_{min} and ρ_{max}, which are respectivily the minimum and maximum number of times a data item is stored in the structure.

The supported operations on the (dictionary) data structure are:

- $Find(Key, v)$: Find the data X such that $Key_X = Key$, and return $Record_X$ to the querying processor v.
- $Insert(X)$: Inserts data item X in the structure. Note however that it is required to be a *safe* operation; if there is already a data item Y in the structure with $Key_Y = Key_X$, the new data item is NOT stored, and a failure message is returned.
- $Delete(Key)$: Deletes a data item X with $Key_X = Key$ from the structure.

5.1.3 Complexity measures

The complexity measures for the above described operations Op are T_{Op} (for the time complexity) and C_{Op} (for the communication complexity). In the following it holds that $C_{Op} = T_{Op}\delta$ for every bound. This stems from the fact that in every implementation a message travels via a path only, and it follows that the communication complexity is the time complexity times the message size (which is in the order of δ). So in the following it suffices to give the time complexity only.

5.2 Some Basic Bounds

With the given characterizations of the distributed system/data structure, the following basic bounds can be derived:

Lemma 5.1 *Consider a distributed data structure supporting the $Insert(X)$ operation. If the insert function f_I[1] defined by the insertion protocol depends only on the inserting processor v and the key of X, then $\hat{M} = \Omega(m\delta)$, assuming sufficiently large $\mathcal{K}$.*

Proof. Making $\mathcal{K}$ sufficiently large, you can always take the data items to be such that all data elements are stored on one processor (just take just those elements that will be placed on a certain processor to be the data elements in the structure). Note however that this is very unlikely to occur if, for instance, uniform hashing is used. □

[1] f_I denotes the function on which the insert operation is based (hashing for instance).

Lemma 5.1 shows us that "oblivious" insert functions should not be chosen, but that things like the current number of data items stored must be taken into account.

Lemma 5.2 *For every graph G and for every data structure supporting safe Insert operations on G, $T_{Ins} = \Omega(D(G))$.*

Proof. For every data item it is possible that it has to be stored at a location at distance $D(G)$. □

Introducing redundancy should improve the bounds on the *Find* operation, as is shown by the following lemma.

Lemma 5.3 *For every graph G and for every data structure supporting the Find operation on G, $T_{Find} \cdot \rho_{min} = \Omega(D(G))$.*

Proof. Lemma 5.3 can be proved by showing where data items reside on on a path of length $D(G)$. □

Above lemmas hold for types of structures in which there are explicit "copies" of data items. If you consider structures in which data may be coded in sophisticated ways or stored implicitly lemma 5.3 has to be reformulated as follows.

Lemma 5.4 (Lemma 5.3) *For every graph G and for every data structure supporting the Find operation on G, $T_{Find} \cdot \frac{M}{m\delta} = \Omega(D(G))$.*

The lemmas direct us towards designing a compact, memory-balanced structure with complexity $O(D(G))$ for all three instruction types.

5.3 Solutions

The following solutions for the distributed dictionary data structure are presented.

CENTRAL structure

In this structure (which is clearly not memory balanced), all data items are stored on one central processor.

Because this processor can be at maximum distance $D(G)$ the complexity bounds for every operation Op are as follows.

Proposition 5.5 $T_{Op} = O(D(G))$. *The memory requirements are $M = \hat{M} = O(m\delta)$.*

2-3-TREE structure

This structure uses a "virtual" 2-3-TREE (ie. the nodes in it do not refer to actual nodes in the system) for the storage of the data. Each data item is stored at the nodes of the tree, and the tree is dynamic in nature.

All operations on the structure are guided through a central coordinator r. This coordinator holds a pointer to the root of the tree, and performs the modifications by simulating the "standard" 2-3-TREE algorithm. An additional substructure LIST, in which the free vertices of the system are stored, is used to maintain the 2-3-TREE.

Because each of the m nodes of the 2-3-TREE is stored at arbitrary locations, every pointer traversal may require $O(D(G))$ messages. Each instruction requires $O(\log m)$ basic manipulation operations on the tree or LIST structure. The complexities of the structure are therefore as follows.

Proposition 5.6 $T_{Op} = O(D(G)\log m)$. *The memory requirements of the resulting structure satisfy* $M = O(m\delta + n\log n)$, $\hat{M} = O(m\delta/n + \log n)$. *The structure is therefore memory-balanced if* $m = \Omega(n\log n)$.

BIN structure

The BIN structure is based on a "flat" B-tree; one where there exist only two levels namely a central vertex r serving as directory, and a collection of bins $B_1, \ldots, B_p$ (each maintained in some vertex of the system). This structure, like the 2-3-TREE structure, also stores the free vertices of the system in a LIST substructure.

In the BIN structure, each bin B_i stores a consecutive segment of k_i data items, $1 \leq k_i \leq 2Load+6$. For every bin r stores the current number of elements k_i, the lower bound $Low(B_i)$ (not for bin 1), and the identity of the vertex at which the bin is located.

A bin is said to be *near-full*, if it stores $2Load + 2 \leq k_i \leq 2Load + 6$ items. And in order to guarantee balanced distribution the following invariant should be met:

At any given time, for any $1 \leq i < p$, either B_i or $B_i + 1$ are near-full.

Load changes affect the invariant, so action has to be taken to maintain the invariant of the structure. The structure is maintained so that if a bin has to be split or merged, the number of items in that bin is set to $2Load + 4$. Repeated changes of the *Load* between two values l and $l+1$ cannot affect bins that do not otherwise modify their data. In other words, each balancing operation can be charged to some specific operation carried out in the bin in consideration. Consequently, the update costs due to *Load* changes can be amortized over previous operations, resulting in an additional amortized cost of $O(D(G)\delta)$ per operation. The amortized worst-case complexities thus become as in the following proposition.

Proposition 5.7 $T_{Op} = O(D(G))$. *The memory requirements satisfy* $M = O(m\delta)$, $\hat{M} = O(m\delta/n + n\delta$. *So it is memory balanced whenever* $m = \Omega(n^2)$. *Furthermore, for* $m = \Omega(n^{1+1/k})$ *it is possible to construct a memory balanced k-BIN variant (ie. a k-level B-tree) with replacing* $D(G)$ *with* $kD(G)$ *in the complexity bounds.*

By performing the burst of activity due to a *Load* change in the "background" one can achieve "actual amortization" instead of the normal "virtual amortization".

Proposition 5.8 *It is possible to implement the BIN dictionary with the same bounds on the individual (rather than amortized) worst-case complexities of the operations.*

DIST-BIN structure

The DIST-BIN structure is essentially the same as the BIN structure, but without a specific center r; every processor is a center on its own holding all the information previously stored at the one center r. By amortizing (again either "virtual" or "actual") the extra cost due to ditributing the center over all operations the following holds.

Proposition 5.9 *It is possible (with some difficulty) to implement the (centerless) DIST-BIN structure with the same complexity bounds as the BIN dictonary. This can be done by amortizing the extra cost due to ditributing over all operations.*

CLUSTERED structure

Replication is used for 1) fault-tolerance, 2) reducing acces cost by placing the data closer to its potential users and 3) work load balancing.

In the CLUSTERED structure, the system is divided into "clusters", each of which contains all data items. Inside every cluster the DIST-BIN structure is used. The *Find* instruction is carried out by inside the cluster, whereas the *Insert* and *Delete* instructions are carried out in all clusters. The following is presented without proof:

Lemma 5.10 *For every graph G and every $C \subseteq V$, there exists an (efficiently constructible) partition $\mathcal{S}$ satisfying $|\mathcal{S}| \leq |C|$, $D(\mathcal{S}) \leq 3D(C)$ and $size(\mathcal{S}) \stackrel{\text{def}}{=} \min\{|S| \mid S \in \mathcal{S}\} \geq D(C)$.*

Corollary 5.11 *Using a partition constructed as in lemma 5.10 based on any set of centers C, it is possible to implement the CLUSTERED dictionary (assuming $m = \Omega(n^2)$), with $T_{Insert} = T_{Delete} = O(D(G))$, $T_{Find} = O(D(C))$, $C_{Insert} = C_{Delete} = O(n\delta)$ and $C_{Find} = O(D(C)\delta)$. The memory requirements of the resulting structure satisfy $\rho_{\max} = |C|$, $M = O(|C|m\delta)$ and $\hat{M} = O(m\delta/D(C))$.*

5.4 Incorporating Concurrent Accesses

Previous structures were designed for sequential operations only. However, an essential requirement from a practical point of view is that it support multiple simultaneous accesses (of both queries and modifications). In database theory a typical approach to ensure correctness, is to try to guarantee the *serializability* of executions for a given sequence of operations. For general data structures we may actually need a stronger consistency property called "order preserving serializability" (in short: if in the actual computation an instruction I is completed before another instruction I' starts, then this order is preserved in the serialization).

In structures that have a center this can be enforced by the center itself simply by handling the requests in order of arrival, starting the next request only after the previous one has completed. If the structure is not centralized, problems due to inconsistent views of the system may arise. However (without proof):

Proposition 5.12 *It is possible to implement the concurrent version of the CLUSTERED dictionary with the same bounds on communication complexity and memory requirements as the sequential version.*

Part II

Algorithms for Fault-Tolerance

Chapter 6

FAULT TOL∃RANCE

Written by *B. Olivier.*

Fault tolerance is a concept that arises naturally from the practical use of distributed systems. In a distributed system many processors are cooperating. If one of them fails we do not want the whole system to collapse. We can have various kind of failures in a system: messages may get lost or corrupted, or a processor may crash or act maliciously. In this chapter we will show the impossibility of distributed consensus in the presence of one faulty process (fail-stop) in an asynchronous setting with (not necessarily FIFO) fair message-passing. The original proof is due to Fischer, Lynch, and Paterson [FLP85].

6.1 Preliminaries

We consider deterministic processes with crash failures and message passing. In the crash model, a process fails by stopping, that is, the process behaves correctly up to a certain point and then halts completely. A message consist of a destination (process identity) and the message contents. A process can perform **send**(p, m) and **receive**$_p(m)$ operations. Operation **send**(p, m) puts a message m addressed to process p in the bag of messages. Operation **receive**$_p(m)$ may retrieve a message m addressed to process p from the bag, or may return $\emptyset$. All non-faulty processes will eventually receive all messages addressed to them.

We consider an interleaving semantics, where a *step* (state transition) consists of a (receive) *event* by a process followed by zero or more send actions by the same process. A *run* is a sequence of steps starting from some global state. A *schedule* is a sequence of *events*. An event can be *applied* to some global state if the corresponding message (to receive) is in the bag in this global state. This is generalized to schedules in the obvious way.

We will use d, e, f for events, s, t for steps, non-capital Greek symbols for schedules, p, q for processes and A, B, C, D for global states. A global state (or configuration) consists of the bag of messages in transit and the local states of all processes; see Fig. 6.1. By determinism of the individual processes a step is determined by the starting state and the event. This also determines the state resulting from the step. Therefore we say that $C \xrightarrow{e}$ is the state obtained by applying event e to state C. This can be generalized to schedules denoted as $C \xrightarrow{\sigma}$. If there exists a schedule σ such that $C \xrightarrow{\sigma} = D$ (also denoted as $C \xrightarrow{\sigma} D$) we say that D is *reachable* from C. If σ_1 and σ_2 are schedules, we will denote their concatenation by $\sigma_1 \circ \sigma_2$, i.e., $C \xrightarrow{\sigma_1 \circ \sigma_2} = C \xrightarrow{\sigma_2} \xrightarrow{\sigma_1}$.

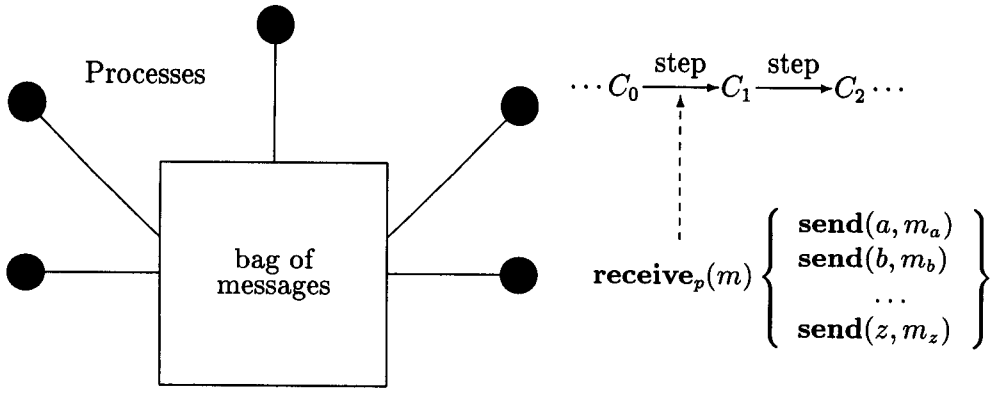


Figure 6.1: CONFIGURATION AND STEPS.

Definition 6.1 (Consensus) *Consider a network with $n > 1$ processes such that each process has a 0 or 1 as input(bit). The output of a consensus protocol is a state in which all processes have written the same value (0 or 1) in their* write-once *output variable, which has initial value $\perp$. In order to avoid trivial solutions, it is required that some initial state can reach a decision value 0 and some initial state can reach 1.*

In the absence of faulty processes the solution is easy. Let all processes send their input plus identity to all other processes. By defining a function on this vector of values all processes can decide a value in unison.

Now we will consider the problem in the presence of one faulty process in the crash model, i.e. it may stop, no longer sending or receiving messages and not making any decision.

Definition 6.2 (Decision state) *A state C has a decision value $v \in \{0, 1\}$ if some process has written v in its output variable. C is called v-decided.*

In a correct consensus protocol all (non-faulty) processes will eventually write the same value.

Definition 6.3 (TC1FTP) *A* totally correct 1-fault-tolerant protocol *(TC1FTP) has no state with more than one decision value reachable from any initial state. In every run with at most one faulty process, from any initial state, some decision state is reached. Some initial state can reach decision value 0 and some initial state can reach 1.*

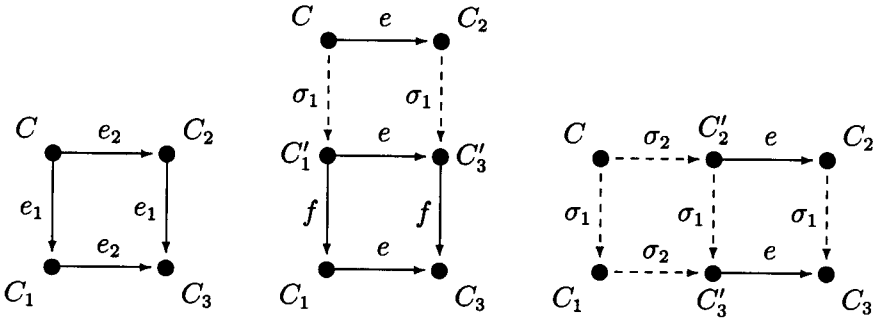
6.2 Non-existence of TC1FTP

Definition 6.4 *Two schedules σ_1 and σ_2 are* process-disjoint *if no process has an event in both σ_1 and σ_2.*

Lemma 6.5 (Commutative schedules) *If σ_1 and σ_2 can both be applied to state C and they are process-disjoint, then they commute, i.e. $C^{\sigma_1 \circ \sigma_2} = C^{\sigma_2 \circ \sigma_1}$.*

Proof. First consider the case that $\sigma_1 = e_1$ and $\sigma_2 = e_2$ (the first picture below). Let p_1 be the process of event e_1 and p_2 of e_2, and m_1 and m_2 the messages received in the events.

Since e_2 concerns a different message than e_1 (process-disjointness), m_2 is still in the bag in state $C_1 = C\xrightarrow{e_1}$ and therefore e_2 is applicable at C_1. For the same reason e_1 is applicable at $C_2 = C\xrightarrow{e_2}$. Since p_2 can't read or write the local state of p_1, process p_1 will change its state in C_2 the same way as in state C and send the same messages. Similarly for process p_2 in states C_1 and C. Since no other process takes a step, all process have the same local state in $C\xrightarrow{e_1 \circ e_2}$ as they have in $C\xrightarrow{e_2 \circ e_1}$. The message bag in $C\xrightarrow{e_1 \circ e_2}$ contains the messages in C, minus m_1 and m_2, plus the messages sent in e_1 and e_2; this is the same in $C\xrightarrow{e_2 \circ e_1}$. Consequently, $C\xrightarrow{e_1 \circ e_2} = C\xrightarrow{e_2 \circ e_1} = C_3$.



Now we will prove with induction to the length of σ_1 that $C\xrightarrow{\sigma_1 \circ e} = C\xrightarrow{e \circ \sigma_1}$ if e and σ_1 are process-disjoint. For $\sigma_1 = \emptyset$ (the empty sequence) it is trivial. By induction hypothesis $C\xrightarrow{\sigma_1 \circ e} = C\xrightarrow{e \circ \sigma_1} = C'_3$, therefore the top square of the second picture above commutes. The bottom square commutes because of the first picture and because e and f are process-disjoint, and therefore the whole diagram commutes.

Finally we prove with induction to the length of σ_2 that $C\xrightarrow{\sigma_1 \circ \sigma_2} = C\xrightarrow{\sigma_2 \circ \sigma_1}$. If $\sigma_2 = \emptyset$ it is again trivial. By induction hypothesis $C\xrightarrow{\sigma_1 \circ \sigma_2} = C\xrightarrow{\sigma_2 \circ \sigma_1} = C'_3$ (commutation of the left square in the third picture above). The right square commutes because the second diagram commutes and because e and σ_1 are process-disjoint. Therefore the third diagram commutes. □

Definition 6.6 (Valency) *A state C is* bivalent *if there exist σ_0 and σ_1 such that $C\xrightarrow{\sigma_0}$ is 0-decided and $C\xrightarrow{\sigma_1}$ is 1-decided. A state C is x-*valent *if for all (applicable) σ, $C\xrightarrow{\sigma}$ is not $\overline{x}$-decided (where $\overline{x} = 1 - x$).*

Note that if C is both 0-valent and 1-valent then no schedule will reach a decision from state C.

In the sequel we will assume the existence of a TC1FT-protocol and conclude with a contradiction. This contradiction is achieved by constructing an infinite run in which only bivalent states occur, consuming all messages and no process failing. The following lemma is the base of the induction.

Lemma 6.7 *Some initial state is bivalent for a TC1FTP.*

Proof. By contradiction, so s'pose not. All initial states are therefore either 0-valent or 1-valent (and all will eventually get decided). By definition of consensus there is an initial state 0-valent and an other 1-valent. We can construct a graph on all initial states such that two adjacent states differ only in the input bit of *one* process. It follows that there must

exist two adjacent states A and B such that A is 0-valent and B is 1-valent. Let A and B differ only in the input bit of process p. Since one process is allowed to fail, we can delay steps of process p until a decision is reached. Since process p has not taken any step in such a schedule, it is applicable to both A and B, implying the same decision is reached. This contradicts the difference in valency. □

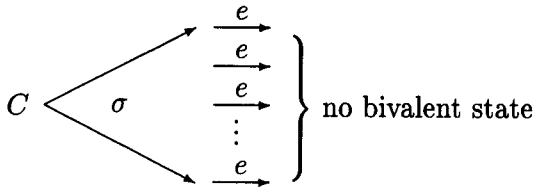
The following lemma is the induction step. It states that from any bivalent state C we can find an applicable schedule $\sigma \circ e$ to another bivalent state D, consuming the message of event e that was in the bag in state C. In this way we can consume all messages while staying bivalent. Beware: there are three claims proved within the proof of Lemma 6.8.

Lemma 6.8 $(\forall C, e)(C$ *is bivalent* $\wedge$ e *is applicable at* $C \Rightarrow$
$(\exists \sigma, D)(D$ *is bivalent* $\wedge$ $C \xrightarrow{\sigma} \xrightarrow{e} D))$

Proof. By contradiction, so s'pose not. Let C and e be arbitrary such that C is bivalent and e is applicable at C and

$$(\forall \sigma, D)(C \xrightarrow{\sigma} \xrightarrow{e} D \Rightarrow D \text{ is not bivalent})$$

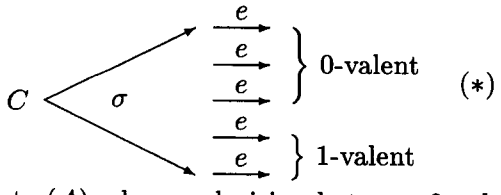
as in the picture below.



Claim 6.9 $(\forall x \in \{0,1\})(\exists \sigma, D)(C \xrightarrow{\sigma} \xrightarrow{e} D \wedge D$ *is* x*-valent*)

Proof. By contradiction, so s'pose that for some x there exists no such σ and D. By *fairness* any run of the protocol eventually includes event e. So if every run is not bivalent nor x-valent, every run must be $\overline{x}$-valent, implying C is not bivalent. Contradiction. □

By the previous claim we have the following situation



The next claim isolates a state (A) where a decision between 0-valent and 1-valent is made.

Claim 6.10 $(\exists \sigma, d, A, B, x \in \{0,1\})(e \notin \sigma \wedge$
A *is bivalent* $\wedge C \xrightarrow{\sigma} A \wedge$
$A \xrightarrow{e}$ *is* x*-valent* $\wedge$
$A \xrightarrow{d} B \xrightarrow{e}$ *is* $\overline{x}$*-valent*)

Proof. By contradiction, so s'pose not, i.e.

$(\forall \sigma, d, A, B, x \in \{0,1\})(e \notin \sigma \wedge C \xrightarrow{\sigma} A \wedge A$ is bivalent $\Rightarrow$
$(A \xrightarrow{e}$ is x-valent $\Leftrightarrow A \xrightarrow{d} B \xrightarrow{e}$ is x-valent$))$ (†)

Note that we have used that $\neg\overline{x}$ – valent $\equiv x$ – valent for schedules ending with event e. This is correct by $(*)$, because any run containing e must either lead to a 0-valent or a 1-valent state.

Let $\sigma = d_1, \ldots, d_m$ such that $C \xrightarrow{\sigma} \xrightarrow{e}$ is x-valent. Let $\sigma_i = d_1, \ldots, d_i$, where $1 \leq i \leq m$. We will now show that $C \xrightarrow{e}$ is x-valent.

We have either $C \xrightarrow{\sigma}$ is bivalent or x-valent. In the latter case there must exist an $1 \leq i < m$ such that $C \xrightarrow{\sigma_i}$ is bivalent and $C \xrightarrow{\sigma_i} \xrightarrow{d_{i+1}}$ is x-valent, because C is bivalent and somewhere the run must turn to x-valent. Since $C \xrightarrow{\sigma_i}$ is bivalent and $C \xrightarrow{\sigma_i} \xrightarrow{d_{i+1}} \xrightarrow{e}$ is x-valent, it follows by the s'pose not, that $C \xrightarrow{\sigma_i} \xrightarrow{e}$ is x-valent. Concluding that in both cases there exists an i such that $C \xrightarrow{\sigma_i}$ is bivalent and $C \xrightarrow{\sigma_i} \xrightarrow{e}$ is x-valent.

From $C \xrightarrow{\sigma_i}$ is bivalent it follows that for any j such that $1 \leq j \leq i$, $C \xrightarrow{\sigma_j}$ is bivalent. Now we show by induction that for all $1 \leq j \leq i$ we have $C \xrightarrow{\sigma_j} \xrightarrow{e}$ is x-valent. For the case $j = i$ it follows from the existence of σ_i. So suppose by induction hypothesis that $C \xrightarrow{\sigma_j} \xrightarrow{d_{j+1}} \xrightarrow{e}$ is x-valent. By $C \xrightarrow{\sigma_j}$ is bivalent and s'pose not it follows that $C \xrightarrow{\sigma_j} \xrightarrow{e}$ is x-valent.

Now from the case $j = 0$ we conclude $C \xrightarrow{e}$ is x-valent. However by $(*)$ we have σ_v (for both $v = 0$ and $v = 1$) such that $C \xrightarrow{\sigma_v}$ is v-valent, implying that $C \xrightarrow{e}$ would be both 0- and 1-valent, contradicting the existence of TC1FTP. □

By the previous claim we may fix

$\sigma : e \notin \sigma$
$A : C \xrightarrow{\sigma} A \xrightarrow{e} x$-valent
$B : A \xrightarrow{d} B \xrightarrow{e} \overline{x}$-valent

Next we claim that the decision made in state A, is made by a single process.

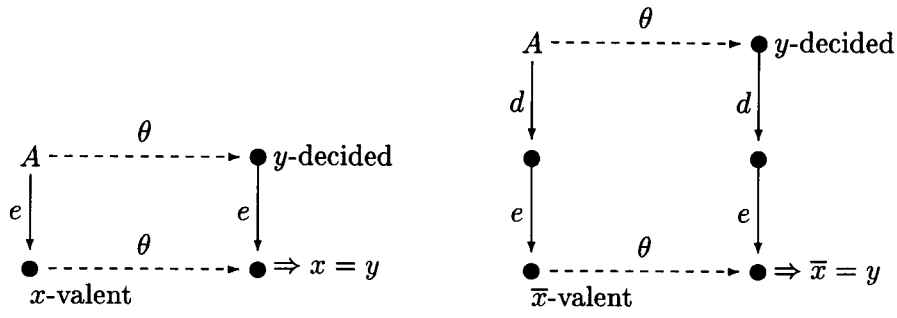
Claim 6.11 *d and e occur at the same process.*

Proof. By contradiction, so s'pose not, then the following diagram would commute

$$\begin{array}{ccc} A & \xrightarrow{e} & x\text{-valent} \\ {\scriptstyle d}\downarrow & & \downarrow{\scriptstyle d} \\ B & \xrightarrow{e} & \overline{x}\text{-valent} \end{array}$$

implying that $A \xrightarrow{e} \circ \xrightarrow{d}$ is both x-valent and $\overline{x}$-valent. This contradicts the fact that any schedule of TC1FTP eventually decides. □

Let p be the process at which d and e occur. We let process p postpone its decision in state A until the other processors have reached some decision y, following a schedule θ. By the definition of TC1FTP such a schedule must exist. Let $A \xrightarrow{\theta}$ be y-valent. Since θ does not contain a step of p, the following diagrams commute



Because any schedule must eventually reach a decision, the left diagram implies that $x = y$ and the right diagram implies $\overline{x} = y$, contradiction. □

Theorem 6.12 *TC1FTP is impossible.*

Proof. Construct an infinite run of bivalent states in which every message is received. By Lemma 6.7 we can find a bivalent initial state. By Lemma 6.8, if no processor fails, then for any bivalent state, some other bivalent state is reachable, without any process failing, in a way that consumes the message of our choice. We let this message be the *oldest* in the bag. By repeating this forever all messages are eventually received. Note that Lemma 6.8 actually requires the selection of an event, but even if the bag is empty it is possible to select a process receiving $\emptyset$. □

Exercise 6.1

1. *Why does the proof still hold if we assume point-to-point transmission, i.e. in an atomic step a processor can send to at most one processor?*
2. *Why does it also hold if we (also) assume separate receive and send steps, i.e. in an atomic step a processor cannot both receive and send?*

Exercise 6.2 *Examine the proof and find the places where we used the following assumptions about the execution model*

1. *Processes are asynchronous.*
2. *Message can be delayed arbitrary long.*

Dolev, Dwork, and Stockmeyer [DDS87] showed that if the messages are delivered in order, then up to n faults can be tolerated. It is also shown that if processes are synchronous and messages are delivered within some time bound (not necessarily in order), then it is again possible to tolerate up to n faults; see Chapter 8.

Chapter 7

Fault Tolerant Algorithms

Written by *D. Alstein.*

Given the impossibility result by Fischer, Lynch and Paterson (Chapter 6), the question arises what problems *can* be solved in an asynchronous environment. The three papers that are discussed in this chapter answer the question, each in a different way.

Itai *et al.* [IKWZ90] study the Leader Election problem. Basically, this problem is not solvable in a fault-tolerant way by the results of Moran and Wolfstahl [MW87]. Itai *et al.* restrict the problem by requiring that failures only occur *before* the algorithm is started. The paper, discussed in Section 7.1, presents an algorithm that tolerates fail-stop failures, provided that less than half of the processors fail.

Attiya *et al.* [ABND+90] introduce the Renaming Problem: all processors have a name, taken from a large set; find new names such that the new names are taken from a smaller name space. The new names must be unique, with the optional extra requirement that the order among the names is preserved. The renaming problem imposes a weaker coordination between the processes than the consensus problem, and is solvable; see Section 7.2.

Bracha and Toueg [BT85] give *probabilistic* algorithms for Consensus, discussed in Section 7.3. Unlike Fischer, Lynch and Paterson, they do not require that correct processors terminate within a finite number of steps, but merely that the probability that an execution takes more than k steps approaches zero as k increases. Their second algorithm can even tolerate malicious ("Byzantine") failures, if less than a third of the processors are faulty.

All algorithms in this chapter are made for asynchronous systems, the communication is assumed to be fault-free, but not necessarily FIFO, and unless stated otherwise fail-stop failures are considered.

7.1 Fault Tolerant Election in Cliques

The leader election algorithm of Itai *et al.* [IKWZ90] is based on a non-resilient algorithm by Humblet [Hum84]. Crash-robust election algorihms are impossible; Hublet's algorithm was adapted to be robust against *initially dead processes*, meaning that processes can crash, but only *before* the execution of the algorithm.

7.1.1 Humblet's algorithm

In Humblet's algorithm, a candidate for leadership is called a *king*. Each king tries to annex other processors. If successful, the annexed processor becomes a *subject*, and is added to the *domain* of the king. Initially, the domain contains only the processor itself. The *size* of a domain is the number of processors that it contains. The decision about whether a processor becomes subject to another is based on its domain size: the processor with the largest domain wins. Processor identities are used to break ties in case the domains are of equal size.

Communication takes place by sending *tokens*. There is one token for each processor. A token contains, apart from the processor's id and domain size, the *type* of the token. This type can either be **join**, **accept** or **reject**. A **join** is a request from the owner to another processor to join its domain. If the receiving processor is a subject, it relays the token to its master. Otherwise, it replies with an **accept** (if its size is smaller), or a **reject**. If it sends an **accept**, it becomes subject, and remembers which link connects it to its master. The owner of the token, upon receiving an **accept** increases its size by 1 and attempts a new annexation by sending the token to another processor. Upon receiving a **reject**, it destroys the token. The algorithm terminates when the domain size of one of the processors becomes equal to n.[1]

7.1.2 Modifications

In order to obtain a t-resilient algorithm, Itai *et al.* have modified Humblet's algorithm. Each processor now owns $t+1$ tokens, which it sends to different processors at the start of the algorithm. Note however that not all processors need to do this. A certain number k of processors start spontaneously by sending the tokens; the others join the algorithm upon receiving a token. At least one of the $t+1$ tokens remains, but there may be more than one token left. Provisions must be made to process them in parallel.

Suppose a processor receives a **reject** as response to a **join**. It may have increased its size in the mean time. In that case, it resends the token to the destination with its new size (it enters a *battle*). While waiting for the answer to the second token, it suspends processing all other incoming tokens. The adversary eventually returns the token (reporting its possibly increased size), after which the battle is over. The sender then processes the suspended tokens.

7.1.3 Performance

The maximum number of failures that can be tolerated is $\lfloor n/2 \rfloor$. Its message complexity is $O(n \log k + n + kt)$, which is proved to be optimal.

7.2 Renaming in an Asynchronous Environment

The Renaming Problem concerns the shrinking of the processor name space. Suppose the n processors all have a unique name, taken from a large (possibly infinite), totally ordered set. We want to give each processor a new name, taken from a smaller set of size N. Two variants of the problem are studied: the *Uniqueness* problem only requires that the new names are unique; the more difficult *Order-preserving* problem requires that the ordering of the processors by their new names is the same as when they are ordered by their old names.

[1] Actually, it is sufficient if processors terminate when their size has become larger than $n/2$.

```
0.  V ← {p}
1.  /* broadcast new set V */
    send V to every other processor
    c ← 1
2.  receive a vector V'
    if V' ⊂ V then
        goto 2          /* V' contains old information: ignore */
    if V' − V ≠ ∅ then
        V ← V ∪ V'      /* V' contains new information: update V */
        goto 1          /* restart */
    if V' = V then
        c ← c + 1       /* received identical copy */
        if c < n − t then goto 2 else goto 3
3.  /* V is a stable vector: decide */
    new name is the pair (|V|, <rank of old name in V >)
4.  /* echo stage: help other processors reach a stable vector */
    repeat forever
        receive a vector V'
        V ← V ∪ V'
        send V to every other processor
```

Algorithm 7.1: THE SIMPLE RENAMING ALGORITHM.

Initially, processors do not know the names of the other processors. Otherwise, a trivial solution to both problems exists, with $N = n$: the processor with the lowest old name chooses 1 as its new name, the next one chooses 2, etc.

The rationale behind studying the Renaming Problem is that smaller names can be useful to reduce message size (since destination names are often part of the header of message packets).

7.2.1 Algorithms

Attiya *et al.* [ABND+90] present three algorithms. Two solve the Uniqueness problem (a simple but inefficient one, and a more complex one), the third algorithm is a solution to the Order-preserving problem.

The basis of the three algorithms is that every processor keeps a *vector* containing its view of the system (i.e. the old names plus possibly extra data, depending on the algorithm). The processors continuously exchange their vectors. A processor's vector is called *stable* if it has received that same vector from $n - t - 1$ other processors. If its vector is stable, it chooses its new name. If a received vector is not equal to its own, it uses a partial ordering on the vectors. In this ordering, a "larger" vector means that it contains more information. If the received vector is larger, the processor updates its own vector with the new information. A smaller vector is ignored.

The simple renaming algorithm is presented as Alg. 7.1. (The rank of a name in V is its position within the ordering of the names in V.)

Proposition 7.1 *The set of vectors that a certain processor holds during an execution of the*

algorithm, is totally ordered with respect to the subset relation.

Lemma 7.2 *The set of stable vectors in an execution is totally ordered with respect to the subset relation.*

Proof. Suppose two stable vectors V and V' are incomparable (i.e. neither $V' \subseteq V$ nor $V \subseteq V'$). Because these are stable vectors, their owners have received at least $n - t$ copies from other processors. Since $n \leq 2t + 1$, there must have been a processor that sent V and V', at different stages of its execution. This contradicts the previous proposition. □

Theorem 7.3 *The new names chosen by the processors are distinct.*

Proof. If two processors chose the same name, their vectors were of equal size. By the previous lemma, these vectors must have been equal, otherwise they would be incomparable. But if the vector sizes are equal, the second part of the name (the rank of the processor's old name in the vector) ensures that the new names are different, which is a contradiction. □

Theorem 7.4 *Every non-faulty processor eventually obtains a stable vector.*

Proof. Let p be a non-faulty processor, and let V be its vector after the last update (note that a vector can be updated at most $n-1$ times). Let τ denote the time of the update. After this update, p sends V to all other processors. Let p' be any non-faulty processor receiving V, and let V' be its own vector at that moment. Since p' sent V' to p when it updated its vector to V', and because V is the last update that p does, it must be that $V' \subseteq V$.

If $V' = V$, then the vector V' that p' sent to p arrives after τ. If, on the other hand, $V' \subset V$, then p' updates its vector to V and sends this to all other processors. In both cases p receives V from p' after time τ. All in all, p receives at least $n - t$ copies of V, thus V eventually becomes stable. □

Note that in this algorithm the partial ordering on vectors is simply the subset relation. For the other algorithms, the contents of the vectors, the ordering relation and the method for choosing the new name are more complex.

7.2.2 Performance and lower bounds

Attiya *et al.* show that the Uniqueness problem is only solvable if $N \geq n + 1$; this is again a consequence of the results of Moran and Wolfstahl [MW87]. The simple algorithm has $N = (n - t/2)(t + 1)$, the second $N = n + t$. For the Order-preserving problem it is shown that $N \geq 2^t(n - t + 1) - 1$ is necessary, and this bound is met by the order preserving algorithm. Furthermore, it is proved that a majority of the processors must be correct: there is no solution to either problem if $n \leq 2t$.

7.3 Asynchronous Consensus and Broadcast Protocols

Fischer, Lynch and Paterson proved that, in an asynchronous environment with certain properties, Consensus is impossible if only one processor may crash. Somewhat surprisingly, a lot more is possible if the termination condition is slightly weakened. The Termination requirement, demanding that correct processors terminate within a bounded number of steps, is

replaced by the probabilistic Convergence requirement: the probability that a correct process does not decide within k steps approaches zero as $k \to \infty$.

Bracha and Toueg [BT85] present protocols for two models: in the first, processor failures are fail-stop, and in the second model faulty processor may have malicious (Byzantine) behavior. The first algorithm tolerates a minority of faulty processors ($t \leq n/2$), the second is correct as long as less than a third of the processors are faulty ($t \leq n/3$). Both bounds are shown to be sharp, i.e. there is no algorithm that is resilient to more failures.

7.3.1 Algorithm for fail-stop failures

Every processor keeps two variables, *value* which holds the processor's current (binary) estimate of the decision, and *cardinality* which counts the number of processors that hold the same *value*. The *value* is initially set to the processor's initial value, and the *cardinality* to one.

The algorithm operates in *rounds*. In each round, a processor sends its a message containing its value, cardinality and round number, to all other processors. Then it waits until it has received $n - t$ messages carrying the same round number. A message with a value i and a cardinality $\geq n/2$ is called a *witness* for i. After receiving the messages, the processor examines them. If there is a witness for some i, it changes its value to i. Otherwise, it sets its cardinality to the number of 0-values or the number of 1-values received, whichever is larger. It also changes its value to 0 or 1, accordingly. If there are more than t witnesses for i in this round, the processor decides i. Otherwise, it goes on to the next round.

The nondeterminism in this algorithm stems from the fact that, if a processor receives more than $n - t$ messages in a certain round, it is free to choose which ones it accepts. However, Bracha and Toueg assume a *fair scheduler*. This means that for any pair i, j of (non-faulty) processors and for any round, there is a non-zero probability that the message sent by i to j in that round will be accepted by j.

Note that the number of rounds is not fixed. If all initial values are the same, all processors terminate in two rounds. If the initial values are distributed equally and the actual number of failures is less than t, there may be many rounds. However, given a certain subset S of $n - t$ correct processors, there is a non-zero probability that all processors from S only consider messages from other processors in S, for three consecutive rounds. If this happens, the protocol terminates.

7.3.2 Algorithm for malicious failures

The algorithm for the malicious case operates in a similar fashion. At the start of each round, a processor sends its state plus the round number to all other processors and then waits for similar messages from other processors. Because of possible malicious behavior, it can not simply process these messages. Instead, it must *echo* them, i.e., resend them to the other processors. It *accepts* the message if it receives more than $(n + t)/2$ identical *echo* messages.

As in the previous algorithm, the processor waits until it has accepted $n - t$ messages. It changes its *value* to the majority of the accepted messages. If this majority is larger than $(n + t)/2$, the processor decides on this value.

Chapter 8

Conditions for solvability

Written by *D. Rezania.*

In Chapter 6 we saw that in a completely *asynchronous* system no consensus protocol is 1-resilient, that is, even one failure cannot be tolerated. In reading the proof of this result, one sees that three types of asynchrony are used:

1. Processor asynchrony allows processors to "go to sleep" for arbitrary long finite amounts of time while other processors continue to run.

2. Communication asynchrony precludes an apriori bound on message delivery time.

3. Message order asynchrony allows messages to be delivered in an order different from the order in which they were sent.

The goal of this chapter is to understand whether all three types of asynchrony are needed simultaneously to obtain the impossibility result. Dolev, Dwork, and Stockmeyer [DDS87] show that they are not, and we see a proof of the impossibility of a 1-resilient consensus protocol even if the processors operate in lockstep synchrony.

Chandra, Hadzilacos, and Toueg [CHT92] propose a technique to circumvent such impossibility results, which broadens the applicability of the asynchronous model of computing.

8.1 Definitions

A *consensus protocol* is a system of $N (N \geq 2)$ processors $P = \{p_1, ..., p_N\}$, each with a special *initial bit.* The processors are modeled as finite state machines with state set Z. There are two special *initial states*, z_0 and z_1. The *consensus problem* is defined (see [FLP85, CHT92] in terms of to primitives *propose(v)* and *decide(v)*, where v is a value drawn from a set of possible initial values. The *Consensus* problem is specified as follows:

- *Validity:* If a correct process decides v, then v was proposed by some process.

- *Agreement:* All correct processes that decide, decide the same value.

- *Termination:* Every correct process eventually decides some value.

In this formulation, all processes are aware that they must each propose and decide a value. Chandra, Hadzilacos, and Toueg [CHT92] solve a stronger version of Consensus, called *Spontaneous Consensus*, in which processes may or may not wish to propose an initial value. Spontaneous Consensus requires that *if* all correct processors eventually propose a value, *then* Termination, Validity, and Agreement are satisfied. If some correct process never proposes a value, then there are no requirements on whether processes decide, or on what values they decide. Clearly, any solution to Spontaneous Consensus is also a solution to Consensus.

8.2 The Principal Boundaries

In this section we will see the results presented by Dolev *et al.* [DDS87] on the minimum synchronism needed for distributed consensus.

In the case of synchronous message order, each processor buffer is modeled as a FIFO queue of messages, and each process p is specified by a *state transition function* δ_p, and a *sending function* β_p where

$$\delta_p : Z \times M^* \to Z,$$

$$\beta_p : Z \times M^* \to \{B \subseteq P \times M | B \text{ is finite}\}.$$

If transmission is *point-to-point*, then $|\beta_p(z, \mu)| \leq 1$ for every p, z and μ. If transmission is *broadcast*, then p can send messages to any number of processors in one step.

In the case of asynchronous message order, each buffer is modeled as an unordered set of messages. We study the dependency of the robustness of systems on synchrony by varying the system synchrony according to the following five parameters. In each case, the property being favorable defines a stronger system model than the property being unfavorable.

		Favorable (F)	Unfavorable (U)
1	processors	synchronous	asynchronous
2	communication	synchronous	asynchronous
3	message order	synchronous	asynchronous
4	transmission	broadcast	point-to-point
5	receive/send	atomic	separate

Like Fischer *et al.* [FLP85], we first show that there is a bivalent initial configuration.

Lemma 8.1 ([FLP85]) *For any choice of system parameters and any $t \geq 1$, if a protocol is t-resilient and solves the nontrivial consensus problem, then the protocol has a bivalent initial configuration.*

Lemma 8.2 ([FLP85]) *Suppose that processors and communication are both asynchronous. Let C be a bivalent configuration and let $e = (p, m)$ be an event applicable to C. Let $\mathcal{L}$ be the set of configurations reachable from C without applying e and let $\mathcal{D} = \{e(E) | E \in \mathcal{L}\}$. $\mathcal{D}$ contains a bivalent configuration.*

The main result of Fischer, Lynch, and Paterson [FLP85] is that in the model with processors, communication, and message order all asynchronous (and the other system parameters favorable), there is no 1-resilient protocol for the nontrivial consensus problem. The following theorem strengthens this by allowing synchronous, even lockstep, processors.

Theorem 8.3 *In the model with asynchronous communication and asynchronous message order (and the other three parameters favorable), there is no 1-resilient nontrivial consensus protocol. Moreover, this is true if processors are lockstep synchronous.*

Proof. First introduce the notation of a "failure step" as an expositional convenience. The corresponding event is denoted $(p, \dagger)$; the dagger denotes death. The next configuration $(p, \dagger)(C)$ is identical to C. To the definition of applicable schedule σ, add the conditions that if a processor takes a failure step, then all of its subsequent steps are failure steps, and if τ is a subsequence of σ in which some processor takes $\Phi + 1$ steps, then every processor takes at least one step in τ (possibly a failure step). In other words processors are forced to keep taking steps, but allowed to fail by taking failure steps from some point on. A C is *ff-bivalent* if there are configurations D_0 and D_1 reachable from C by failure-free runs such that D_v has decision value v for $v = 0, 1$. The following two lemmas are essential to the proof:

Lemma 8.4 ([DDS87]) *Let C be a configuration reachable from some initial configuration by a failure free run. Then C is ff-bivalent iff C is bivalent.*

The following lemma shows that if C is a bivalent configuration, and p is a processor, then there is a schedule σ such that $\sigma(C)$ is bivalent and p takes a step in σ. Moreover, if p's buffer is nonempty in C, then for any message m in p's buffer, there is a σ in which p receives m.

Lemma 8.5 ([DDS87]) *Let C be a bivalent configuration reachable from some initial configuration by a failure-free run, and let p be a processor. If buff(p,C) $\neq \emptyset$, let m be an arbitrary message in buff(p, C), or let $m = \emptyset$ if buff(p,C) $= \emptyset$. Let $\mathcal{L}$ be the set off configurations reachable from C by any failure-free run in which the event $e = (p, m)$ is not applied, and let*

$$\mathcal{D} = \{e(E) | E \in \mathcal{L} \text{ and } e \text{ is applicable to } E\}.$$

Then $\mathcal{D}$ contains a bivalent configuration.

Definition 8.6 *A processor p is* nonfaulty *in an infinite run if it takes infinitely many steps and is* faulty *otherwise. For $0 \leq t \leq N$, an infinite run is a* t-admissible run from I *if*

1. *the associated schedule is applicable to I;*
2. *at most t processors are faulty;*
3. *all messages sent to nonfaulty processors are eventually received.*

Using lemmas 8.4 and 8.5, construct an infinite t-admissible run that is not deciding as follows. Let B_1 be an initial bivalent configuration. In general, if B_i is bivalent, let $p = p_j$ where $j \equiv i \pmod N$ and let $B_{i+1} = \sigma(B_i)$, where σ is obtained from lemma 8.5. Moreover, if p's buffer is nonempty in B_i, let p receive a message that has been in the buffer for the longest time. The resulting infinite run is 0-admissible. It is not a deciding run because, by partial correctness, a bivalent configuration has no decision value. □

Theorem 8.7 *If communication is synchronous, transmission is broadcast, and receive/send is atomic (and other two parameters are unfavorable), there is an n-resilient strong consensus protocol.*

Proof. (Outline) According to protocol for p_i, P_i first broadcasts its name and initial value (i, x_i), and then attempts to receive messages for 2Δ of its own steps (where communication is Δ-synchronous). If it receives a message "Decide v", it decides v. If it receives (j, x_j) from other processor, it remembers x_j and attempts to receive for 2Δ more steps. If at some point it has run for 2Δ steps without receiving any messages, it sees whether all initial values received from other processors are the same as its own initial value. If so it decides on this common value v; if not it decides $v = 0$. In either case it broadcasts "Decide v".

Termination of the protocol is obvious. The proof is concluded by showing that if the initial bits are not all the same, there is no run with two different decision values [DDS87]. □

The following two theorems show the effect of replacing broadcast transmission by point-to-point transmission.

Theorem 8.8 ([DDS87]) *If model of theorem 8.7 is weakened by having point-to-point transmission, then there is a 1-resilient strong consensus protocol.*

Theorem 8.9 ([DDS87]) *Assume $N \geq 3$. If model of theorem 8.7 is weakened by having point-to-point transmission, then there is no 2-resilient nontrivial consensus protocol. Moreover, this is true even if message order is synchronous and communication is immediate.*

The following theorem is well known. A processor can tell if another has failed by using "time-outs". Consensus can be reached by simplifying any algorithm that reaches Byzantine agreement.

Theorem 8.10 *If processors and communication are both synchronous (and the three other parameters are unfavorable), then there is an N-resilient consensus protocol.*

Theorem 8.11 *If message order is synchronous, and transmission is broadcast (and other three parameters are unfavorable), there is an N-resilient strong consensus protocol.*

Proof. The first step of each processor is to broadcast its initial value. It then attempts to receive and decides on the first value received. Since message order is synchronous, the first value broadcasted will be decided by all. □

8.3 Failure Detectors: Model and Properties

A *failure detector* is defined to be a distributed oracle – each process has its own local failure detector module, and the failure detector modules of two different processes need not agree on the list of processes that are suspected to have crashed. Let $Q = \{p_1, p_2, ..., p_n\}$. We define a distributed failure detector D as the vector $D = \langle D_{p1}, D_{p2}, ..., D_{pn} \rangle$, where D_p, the failure detector module at process p, is a function from time and the set of all runs to 2^Q. $D_p(t, \sigma)$ is the set of processes that are suspected to have crashed by p's failure detector module at time t in run σ. After a process crashes we do not care what its failure detector module indicates.

Failure detectors are characterized by the *completeness* and *accuracy* properties. We consider two types of completeness properties: *strong completeness* (i.e., eventually every process that crashes is permanently suspected by *every* correct process), and *weak completeness* (i.e., eventually every process that crashes is permanently suspected by *some* correct process).

We also define two types of accuracy properties: *strong accuracy* (i.e., correct processes are never suspected), and *weak accuracy* (i.e., some correct process is never suspected).

The following three classes of failure detectors are defined:

- $\mathcal{P}$, the set of *perfect failure detectors* that satisfy the strong completeness and the strong accuracy properties.
- $\mathcal{S}$, the set of *strong failure detectors* that satisfy the strong completeness and the weak accuracy properties.
- $\mathcal{W}$, the set of *weak failure detectors* that satisfy the weak completeness and the weak accuracy properties.

8.3.1 Solving Spontaneous Consensus

To solve spontaneous consensus, it is first shown how to convert any given weak failure detector $W \in \mathcal{W}$ to a strong failure detector $S \in \mathcal{S}$. The algorithm works as follows: every process p reliably broadcasts the names of the processes that it suspects according its failure detector module, W_p. Every process q that delivers such names, adds them to its list of suspects S_q. The resulting failure detector S satisfies strong completeness and weak accuracy.

Then they give an algorithm that solves the spontaneous consensus in asynchronous systems by using any given strong failure detector. The algorithm runs through three phases. In phase 1, processes execute n asynchronous rounds, during which they broadcast and relay their initial values. At the end of each asynchronous round, each process p waits until, for each process q, either p receives a message from q or q is suspected to have crashed by p's failure detector module S_p. By the end of phase 2, correct processes agree on a vector based on the initial values of all processes. It is shown that this vector contains the initial value of at least one process. In phase 3, correct processes decide the first non-trivial component of this vector.

8.3.2 Solution to Atomic Broadcast

Formally, *atomic broadcast* is a spontaneous reliable broadcast that satisfies *total order*, i.e., if two correct process p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'. Total order and agreement ensure that all correct processes deliver the same sequence of messages. Atomic broadcast is perhaps the most powerful communication paradigm for fault-tolerant distributed computing.

Chandra *et al.* [CHT92] first shown how to solve atomic broadcast by reducing it to spontaneous consensus. The algorithm works by using repeated executions of spontaneous consensus. Processes disambiguate between these independent executions by tagging a counter k to all the messages pertaining to the k^{th} execution of spontaneous consensus. Informally, the k^{th} execution of spontaneous consensus is used to agree on the k^{th} batch of messages to be atomically delivered.

When a process intends to atomically broadcast a message m, it broadcasts m, using spontaneous reliable broadcast. When a process p delivers m, it adds m to the set $ab\_broadcast_p$, which contains all the messages submitted for atomic broadcast that p is currently aware of. When p atomically delivers a message m, it adds m to the set $ab\_delivered_p$. Thus $ab\_broadcast_p - ab\_delivered_p$ is the set of messages that were submitted for atomic broadcast but not yet atomically delivered by p. When this set is not empty, p proposes it as the next batch of messages to be atomically delivered. The k^{th} batch of messages that p atomically delivers is the set of messages agreed upon by the k^{th} execution of spontaneous consensus, minus those messages that p has already atomically delivered.

Part III

Stabilizing Algorithms

Chapter 9

Stabilization: Mutual Exclusion

Written by *M. P. Bodlaender.*

In this chapter the problem of designing self-stabilizing algorithms is discussed. Self-stabilizing algorithms have the ability to recover from temporary failures that can corrupt memory, but cannot corrupt the program code. A self-stabilizing algorithm in a corrupted state will eventually converge towards a correct state, which is defined by some predicate. An example of the use of a self-stabilizing algorithms is that it is unnecessary to initialize variables for such an algorithm. Self-stabilizing algorithms are in a sense simular to neural networks which by 'learning' also converge to some correct state.

9.1 A Formal Specification

It is assumed that processes communicate by reading each others local states. Because of this assumption there are no messages "in transit". This simplification of the classic distributed system leads to the following model:

Definition 9.1 *A system is a tuple $(C, \rightarrow)$, where C is a set of configurations and $\rightarrow$ a binary relation on C.*

$\alpha \rightarrow \beta$ means that the system can go from configuration α to configuration β in a single step.

Definition 9.2 *In a system $S = (C, \rightarrow)$ a configuration β is reachable from configuration α, denoted $\alpha \rightarrow^* \beta$ if there is a sequence $(\alpha_0, ..., \alpha_k)$ of configurations, such that $\alpha = \alpha_0, \beta = \alpha_k$ and $\alpha_i \rightarrow \alpha_{i+1}$ for all i.*

A computation is a sequence of configurations, where every configuration is reachable in a single transition from the last. Only maximal computations, which are infinitely long or end in a configuration without a possible transition, are taken into account. The correct behaviour of a system is now defined as a predicate on the set of all possible computations.

Definition 9.3 *A specification of S is a subset of all possible computations.*

A system stabilizes to a specification if in every computation eventually a configuration is reached from where only computations according to the specification are possible.

Definition 9.4 *System S stabilizes to specification F if for all possible computations $C = (\alpha_0, \alpha_1, ...)$ holds $\exists i : \forall C' = (\alpha_i, ...) : C' \in F$*

9.2 Examples

The question that remains is: are there (non-trivial) self-stabilizing algorithms? To answer to this question and to give the reader some feeling as to how self-stabilizing algorithms might be constructed, the problem of mutual exclusion in a distributed network is solved using self-stabilizing algorithms. The mutual exclusion problem is that of all processes in the network, at most one has at any one time a "permit" to perform some action, but eventually all processes will get a permit infinitely often.

9.2.1 A solution with K-state machines

As a start, the mutual exclusion problem is tackled by allowing that not all processes in the network run the same algorithm. This is called the non-uniform exclusion problem. Furthermore, the network topology is constricted to rings. Communication between adjacent processes is implemented by allowing processes to "read" their neighbors internal state.

The algorithm

This solution, which was first presented by Dijkstra [Dij82], contains one process (process P_0) which differs from the others. The algorithm works on a uni-directional ring with N processes. Every process has K possible states and arithmetic performed on states is modulo K. The state of process P_i is denoted S_i. If a process $P_{i\neq 0}$ has a permit and performs a step, it changes its state to S_{i-1}. If process P_0 performs a step it changes its state to $S_{N-1}+1$. A process has a permit if the following holds:
Process $P_{i\neq 0}$ has a permit if $S_i \neq S_{i-1}$.
Process P_0 has a permit if $S_0 = S_{N-1}$.

Proposition 9.5 *In every configuration at least one process has a permit.*

Proof. Suppose P_1 through P_{N-1} have no permit. Then S_0 ... S_{N-1} are all equal, so $S_0 = S_{N-1}$, thus process P_0 has a permit. □

Note that a process $P_{i\neq 0}$ with a permit changes its state so it loses its permit: $S_i = S_{i-1}$. Because of proposition 9.7 the system is deadlock free. P_0 changes its state to $S_{N-1}+1$ so P_0 also loses its permit if it performs a step.

Proposition 9.6 *No state change increases the number of permits in the network.*

Proof. As the permit of every process P_i depends only on its own state S_i and its neighbors state S_{i-1}, changing the state of P_i will at most add one permit to the network. The process P_i also loses its own permit. □

Proposition 9.7 *Let γ be a configuration with one permit, and $\gamma \rightarrow \delta$. Then δ is also a configuration with one permit.*

Proof. This proposition essentially states that once in a legal state, the system will remain in a legal state. The proof follows from a combination of the last two propositions. □

Proposition 9.8 *Every computation beginning in a configuration with one permit contains an infinite number of transitions for every process.*

Proof. It can be seen from the possible transitions that in a configuration with exactly one permit, that permit travels in one direction around the ring, ensuring that all processes get the permit infinitely often. □

The propositions above prove that once in a correct state, the system will remain in a correct state. In the following lemma it is proven that in a bounded time such a correct state is actually reached.

Lemma 9.9 *If $K \geq N$ then every computation reaches a configuration with exactly one permit in a bounded number of transitions.*

Proof. First we prove that in a bounded number of steps P_0 reaches an unique state, thus $\forall i > 0 : S_i \neq S_0$.

Suppose there are some P_i such that $S_i = S_0$. It can be easily seen that P_0 will infinitely get a permit, as permits travel only in one direction along the ring. This ensures that there can only be a finite number of transitions in the ring if P_0 does not get a permit. The claim now follows directly from proposition 9.6 which ensures deadlock-freedom.

Every time P_0 makes a transition, S_0 becomes $S_{N-1} + 1$. As $K \geq N$, it is certain that at some time P_0 reaches an unique state, as $S_1 ... S_{N-1}$ only change to match the value of their neighbor, thus not bringing new values into the ring.

This can be proven by looking at some state $S_i = x$ for which there is no process in the ring with state $x + 1$. There will be such a process as $K \geq N$ and P_0 is not unique. Now either in finite time $S_0 = x + 1$ or the value x is overwritten and absent from the ring. If it is overwritten repeat the process. At most $N - 1$ values can be overwritten because no new values are introduced in the ring, so in finite time P_0 will be in an unique state.

Now P_0 is in a unique state, it will not get a permit before $S_1 = S_2 = ... = S_{N-1} = S_0$. After that, there is exactly one permit in the system. □

Theorem 9.10 *The algorithm described above is self-stabilizing.*

Proof. With lemma 9.10 it is shown that in a bounded number of steps there is exactly one permit. Proposition 9.8 ensures that it system will remain in such a correct state. Proposition 9.9 ensures that the permit will travel around the ring. □

9.2.2 A solution with 3-state machines

In the previous section a solution using K-state machines was used, where $K \geq N$. Here a solution first published by Dijkstra [Dij86] is presented which uses 3-state machines. The algorithm uses communication in two directions over the ring.

The algorithm

Processes P_0 and P_{N-1} use different algorithms than processes $P_1 ... P_{N-2}$. In the following lines all arithmetic is modulo 3 as there are only 3 different states. The transitions are as follows:

P_0 : if $S_1 = S_0 + 1$ then $S_0 = S_0 + 2$
$P_{0<i<N-1}$: if $S_{i-1} = S_i + 1$ or $S_{i+1} = S_i + 1$ then $S_i = S_i + 1$
P_{N-1} : if $S_0 = S_{N-2}$ and $S_0 + 1 \neq S_{N-1}$ then $S_{N-1} = S_0 + 1$

Proposition 9.11 *There is at least one permit in the ring.*

Proof. Suppose no process in the ring holds a permit. Then $S_1 = S_2 = .. = S_{N-2}$ holds because of the second transition rule. From the first and the second transition rule follows $S_0 = S_1$. Now because $S_0 = S_{N-2}$ and the third transition rule it must be shown that $S_0 + 1 = S_{N-1}$. But according to transition rule two $S_{N-1} \neq S_{N-2} + 1$, so $S_0 + 1 \neq S_{N-1}$. This is a contradiction so there is at least one permit. □

Theorem 9.12 *The algorithm is self-stabilizing.*

Proof. This follows from the fact that permissions are bounced back and forth along the ring. Thus two permissions traveling in the ring will (if not cancelled already) eventually bounce into each other. The rules for guiding permissions are created such that if two permissions meet, one of them is cancelled and the other continues. This construction guarantees that if there is ≥ 1 permit, eventually there will be exactly one. Proposition 9.11 ensures that there will at least be one permit in the ring. □

9.2.3 A uniform solution

The solutions of Dijkstra in the previous sections rely heavily on the special top and bottom processes to prevent the system from deadlock. Burns and Pachl [BP88] gave a uniform solution on a directed ring.

Their algorithm only works on rings with a prime size, as it can be proven that no deterministic mutual exclusion algorithms can be constructed on a uniform ring with non-prime size.

Theorem 9.13 *On a uniform ring of non-prime size, no deterministic mutual exclusion algorithm can be constructed.*

Proof. It is possible to partition the ring in K identical sections of length N/K, where the i^{th} processes of every section are exactly equal. This is called a *symmetric configuration* Now it is possible to construct an execution of the algorithm where corresponding processes take their steps concurrently, thus never breaking the symmetry. $S_i = S_{i+N/K} = S_{i+2N/K} = ... = S_{i+(K-1)N/K}$ holds in the entire computation, so if some process P_i decides it has a permit, so will processes $P_{i+j\times N/K}(0 \leq j < K)$. □

Note how this proof is entirely based on the possibility of partitioning the ring into $K > 1$ equal sections.

9.2.4 Self-stabilization in tree structures

In this section a solution will be presented for the case that the system has the structure of a tree. The algorithm presented here was proposed by Kruijer [Kru79]. The requirement that only one process has a permit at any one time is changed to the requirement that no process which holds a permit has an ancestor which holds a permit. For example if the root holds a permit, no other process will hold a permit, but if the root has no permit all its children can have permits.

The algorithm

The tree T is represented by N processes numbered 1 to N. The structure is characterized by an array sup$[1:N]$ where sup$[i]$ is the index of the father of i (0 for the root). The state of process i consists of two variables $s[i] \in \{0,..,K-1\}$ for some arbitrary $K > 3$ and boolean $eq[i]$. A process i can make two state changes:

(a) if $eq[i]$ =false and test(i) =true then $eq[i]$ =true
(b) if $eq[i]$ =true and $s[i] \neq s[\text{sup}[i]]$
then if sup$[i]$=0 then $s[i] = s[i] + 1$ else $s[i] = s[\text{sup}[i]]$
$eq[i]$ =false

In (a) a boolean procedure *test* is used which returns true if i is a leaf of T. If i is not a leaf, test returns true if all processes p with sup$[p]$=i have $s[p] = s[i]$ and $eq[p] = eq[i]$ and test returns false otherwise.

Definition 9.14 *A state is a* perfect *state if* $s[1] = s[2] = ... = s[N]$ *and* $eq[1] = eq[2] = ... = eq[N]$.

The legitimate states of the system are now defined as the perfect states and all states reachable from the perfect states.

Analysis

The correctness proof first shows that there is at least one permit, using simular arguments as in the previous proofs in this chapter.

It then proves that the system will converge within a bounded number of steps to a perfect state by first ensuring that in any computation an infinite number of moves of the root of T must occur. This follows from the fact that all sons of the root of T can take at most 3 steps if the root remains constant. Using induction it follows that there can only be a finite number of steps if the root remains constant.

Now the root of T will eventually perform step (a) later followed by step (b). It can be shown that the next time the root is allowed to perform a step, the system is in a perfect state.

Chapter 10

Stabilizing Graph Algorithms[1]

Written by *G. Tel.*

In this chapter three stabilizing graph algorithms are discussed. The desired postcondition of such an algorithm is expressed as a condition ψ relating the variables of the processes to the structure of the graph (and possibly the inputs available at the nodes). We say system S stabilizes to ψ if a set of configurations (global states) $\mathcal{L}$ exists such that

1. $\mathcal{L}$ is closed and $\gamma \in \mathcal{L}$ implies $\psi(\gamma)$.

2. every computation reaches a configuration of $\mathcal{L}$.

The algorithm may terminate, but this is not always the case. Section 10.1 presents a ring orientation algorithm that may eventually go through an infinite sequence of different (but oriented) configurations. Section 10.2 presents an algorithm to find a maximal matching, in which legitimate configurations are terminal. Section 10.3 presents an algorithm to compute a leader and spanning tree; this algorithm terminates after establishing its postcondition.

10.1 Ring Orientation

In the ring orientation problem we consider an undirected ring of N processes, where each process has labeled one of its links with *succ* and the other with *pred*. Process p's label of the link pq is called l_{pq}, and no global consistency of the labels is initially assumed. The goal of an orientation algorithm is to compute a consistent sense of direction in the ring.

The orientation problem requires to reach the postcondition ψ, defined by "for every edge pq, l_{pq} is *succ* if and only if l_{qp} is *pred*". It was shown by Israeli and Jalfon [IJ90] that no deterministic uniform ring orientation algorithm exists if the ring size is even, and also that no solution is possible in the state reading models. In this section we present Israeli and Jalfon's algorithm, assuming the link register model and the availability of identities to break symmetry on links.

The algorithm, given as Alg. 10.1, uses, besides the link registers holding *pred* or *succ*, a variable s with values S, R, and I. In one step, process p considers its state, the state of neighbor q and the link registers, and changes state if one of the five guards is *true*. The processes circulate tokens, setting their successor to the direction of the last forwarded token;

[1]This chapter is taken from Section 15.3 of [Tel94].

var s_p : (S, R, I) ;

(1) $\{state_p = I \wedge s_q = S \wedge l_{qp} = succ\}$
$s_p := R$; **if** $l_{pq} = succ$ **then** *flip*

(2) $\{s_p = S \wedge l_{pq} = succ \wedge s_q = R \wedge l_{qp} = pred\}$
$s_p := I$

(3) $\{s_p = R \wedge l_{pq} = pred \wedge \neg(s_q = S \wedge l_{qp} = succ)\}$
$s_p := S$

(4) $\{s_p = s_q = S \wedge l_{pq} = l_{qp} = succ \wedge p < q\}$
$s_p := R$; *flip*

(5) $\{s_p = s_q = I \wedge l_{pq} = l_{qp} = succ \wedge p < q\}$
$s_p := S$

procedure *flip*: reverse *succ* and *pred*

Algorithm 10.1: RING ORIENTATION ALGORITHM.

if two tokens meet, one is eliminated. Eventually all remaining tokens will travel in the same direction, which causes the ring to be oriented, but then the remaining tokens will keep circulating. The S state of a process indicates that it wants to pass a token to its current successor, while the R state indicates that a process has accepted a token from a neighbor, which is now its predecessor.

In action (1), if q wants to transmit to the idle p, p accepts the token and makes q its predecessor. In action (2), p observes that its successor has accepted the token it wants to transmit, and becomes idle. In action (3), p observes that q, from which it has accepted a token, has become idle, and starts transmitting the token to its successor. The last two actions concern the symmetric situations where (4) two tokens traversing in different directions meet, and (5) two idle processes are inconsistently oriented. In action (4), p accepts q's token (implicitly destroying its own) if it's identity is the smaller. In action (5), p generates a token (to "overrule" q's orientation), again if its identity is smaller.

The effect of each action is visualized in Fig. 10.2, where the state and orientation of each process is compactly represented with an arrow in the direction of its successor.

Call a configuration legitimate if and only if it is oriented, i.e., all arrows point in the

Action no.	q	p		p
(1)	$\vec{S}$	I	$\longrightarrow$	$\vec{R}$
(2)	$\overleftarrow{R}$	$\overleftarrow{S}$	$\longrightarrow$	$\overleftarrow{I}$
(3)	$\neg\ \vec{S}$	$\vec{R}$	$\longrightarrow$	$\vec{S}$
(4)	$\vec{S}$	$\overleftarrow{S}$	$\longrightarrow$	$\vec{R}$ if $p < q$
(5)	$\vec{I}$	$\overleftarrow{I}$	$\longrightarrow$	$\overleftarrow{S}$ if $p < q$

Figure 10.2: EFFECT OF ACTIONS OF ALG. 10.1.

same direction.

Lemma 10.1 *$\mathcal{L}$ is closed and $\gamma \in \mathcal{L}$ implies $\psi(\gamma)$.*

Proof. Legitimate configurations are oriented by the choice of $\mathcal{L}$, which immediately implies the second part. Further, only steps of types (1), (2), and (3) occur, because steps (4) and (5) are only enabled when there are differently oriented neighbors. Steps (2) and (3) never flip the orientation of a process, and neither does (1) if p is oriented in the same direction as q, which showns closedness of $\mathcal{L}$. □

Proposition 10.2 *A terminal configuration is legitimate.*

Proof. Let γ be terminal; no process is in state R in γ, because if $s_p = R$ then either p is enabled (by action (3)) or its predecessor is enabled (by action (2)).

If γ is not oriented there are two processes pointing at each other, i.e., p and q with $l_{pq} = l_{qp} = succ$. If one of p and q is idle action (1) is enabled, if both are idle action (5) is enabled, and if both are sending action (4) is enabled. It follows that a terminal configuration without receiving processes is oriented. □

A process *holds a token* if its state is S or its state is R and its predecessor is not in state S. A token moves from p to q in action (2), and is destroyed (created) in p in action (4) (or (5), respectively); these steps are called *token steps*. Steps (1) and (3) have no effect on tokens and are called *silent steps*. After step (1), the next step of the same process is step (3), and after step (3) the next step is step (4) or step (2), which bounds the number of silent steps linearly in the number of token steps.

Theorem 10.3 *The protocol converges to a legitimate state.*

Proof. Token creation occurs in a pair $\vec{I}\,\overleftarrow{I}$, and such a pair is not formed in any of the five actions, so a token is generated only in an initially idle process as the first step of that process. This implies that the overall number of tokens that exists during an execution is bounded by N.

If a token exists in a configuration during the execution and this token has already moved k times, then a sequence of $k+1$ processes (ending in the one holding the token) is consistently oriented. All these processes have forwarded the token and adopted its direction, and no tokens have been created "behind" the token that could have disturbed the orientation again. So if one token moves $N-1$ times all processes are oriented equally, and the configuration is legitimate.

Because there are no more than N different tokens during the execution, a legitimate configuration is reached after at most N^2 token steps. □

10.2 Maximal Matchings

A *matching* in a graph is a set of edges such that no node of the graph is incident to more than one of the edges. The matching is *maximal* if it cannot be extended with more edges of the graph, and a maximal matching of a graph can be constructed in time linear in the number of edges. Each edge is considered in turn, and included in the matching if it is not incident with any edge already in the matching. This algorithm is, however, inherently sequential, and not suited for distributed execution; we shall now present a stabilizing algorithm for constructing maximal matchings, proposed by Hsu and Huang [HH92].

var $pref_p$: $Neigh_p \cup \{nil\}$;

$\mathbf{M}_p$: $\{pref_p = nil \ \wedge \ pref_q = p\}$
$pref_p := q$

$\mathbf{S}_p$: $\{pref_p = nil \ \wedge \ \forall r \in Neigh_p : pref_r \neq p \ \wedge \ pref_q = nil\}$
$pref_p := q$

$\mathbf{U}_p$: $\{pref_p = q \ \wedge \ pref_q \neq p \wedge pref_q \neq nil\}$
$pref_p := nil$

Algorithm 10.3: THE MAXIMAL MATCHING ALGORITHM.

The Matching Algorithm. Process p has one variable $pref_p$, whose value belongs to $Neigh_p \cup \{nil\}$, representing p's *preferred neighbor*. If $pref_p = q$, p has selected its neighbor q to become matched with, i.e., to include edge pq in the matching; $pref_p = nil$ if p has not selected a matching partner. We distinguish five cases depending on p's preference and that of its neighbors. If p has selected q, then p is waiting (for q) if q did not make a selection yet, matched if q selected p, and chaining if q selected a neighbor other then p. If p has not made a selection, p is dead if all neighbors of p are matched and free if there is an unmatched neighbor. Formally:

$$\begin{array}{lcl} wait(p) & \equiv & pref_p = q \in Neigh_p \wedge pref_q = nil \\ match(p) & \equiv & pref_p = q \in Neigh_p \wedge pref_q = p \\ chain(p) & \equiv & pref_p = q \in Neigh_p \wedge pref_q = r \in Neigh_q \wedge r \neq p \\ dead(p) & \equiv & pref_p = nil \wedge \forall q \in Neigh_p : match(q) \\ free(p) & \equiv & pref_p = nil \wedge \exists q \in Neigh_p : \neg match(q) \end{array}$$

The required postcondition for the algorithm is

$$\psi \equiv \forall p : (match(p) \vee dead(p)).$$

Proposition 10.4 *If ψ holds, the set $M = \{(p, pref_p) : pref_p \neq nil\}$ is a maximal matching.*

Proof. If there is an edge $pq \in M$, then $pref_p = q$ or $pref_q = p$ by the definition of M; but because q is not waiting or chaining, the latter implies $pref_p = q$ as well. It follows that at most one edge incident to p belongs to M, hence M is a matching.

To show the maximality, assume $M \cup \{pq\}$, where $pq \notin M$, is also a matching; then M contains no edge incident to p, which implies $dead(p) \vee free(p)$, and hence (by ψ) $dead(p)$. But $dead(p)$ implies $match(q)$, so M contains an edge incident to q, contradicting the possible extension of M with pq. □

The algorithm is described for the state read-all model and consists of three actions for each process p; see Alg. 10.3. Process p matches with neighbor q (action $\mathbf{M}_p$) if p is free and q has selected p. If p is free but cannot match, it selects a free neighbor if such is possible (action $\mathbf{S}_p$), and p unchains (action $\mathbf{U}_p$) if p is chaining.

Analysis of the Algorithm.

Lemma 10.5 *Configuration γ is terminal if and only if $\psi(\gamma)$.*

Proof. Action $\mathbf{M}_p$ is enabled for p only if neighbor q is waiting, action $\mathbf{S}_p$ requires that p is free, and action $\mathbf{U}_p$ that p is chaining, hence $\psi(\gamma)$ implies that no action is enabled in γ.

If ψ does not hold, there is a p such that p is chaining, waiting, or free; in a chaining p, $\mathbf{U}_p$ is enabled and if p is waiting for q, $\mathbf{M}_q$ is enabled. Finally, assume p is free and q is an unmatched neighbor. Because p is not matched, q is not dead. If q is waiting, the matching action is enabled for one of its neighbors and if q is chaining $\mathbf{U}_q$ is enabled. Finally, if q is free, p and q can select each other. Hence, if ψ does not hold, the configuration is not terminal. □

Lemma 10.6 *Alg. 10.3 reaches a terminal configuration in $O(N^2)$ steps.*

Proof. Define the norm function F by the pair $(c+f+w,\ 2c+f)$ where c, f, and w are the numbers of chaining, free, and waiting processes, respectively. We show that F decreases (in the lexicographic order) with every step; first observe that a matched or dead process remains matched or dead forever, so $c+f+w$ never increases.

Action $\mathbf{M}_p$: This action applies when p is free and its neighbor q is waiting and causes both to become matched, decreasing $c+f+w$ by 2. (The sum may decrease even further if some neighbors of p and q become dead.)

Action $\mathbf{S}_p$: This action is applicable when p is free, and causes p to become waiting, thus decreasing $2c+f$ by 1. No waiting process becomes free or chaining, because the action is applicable only if no process waits for p, and no free process becomes chaining.

Action $\mathbf{U}_p$: This action is applicable when p is chaining, and causes p to become free (if there are unmatched neighbors) or dead (if all neighbors are matched), thus decreasing $2c+f$ by at least 1. Chaining neighbors of p may become waiting, thus further decreasing c.

As $c+f+w$ is bounded by N and $2c+f$ by $2N$, there are at most $(N+1)(2N+1)$ different values of F, so each execution terminates within $2N^2+3N$ steps. □

Algorithm 10.3 is now easily seen to stabilize to ψ, by taking all configurations satisfying ψ as the legitimate ones.

10.3 Election and Spanning Tree

Afek, Kutten, and Yung [AKY90] proposed a stabilizing algorithm to compute a spanning tree on a network, with the larges process being the root. We describe the algorithm for the read-all state model (although the algorithm can also be used for the read-one model) and assuming that the network is connected. In this subsection we shall use capitalized words for process variables, and lower case words for predicates.

Process p maintains variables $Root_p$, Par_p, and Dis_p to describe the tree structure; we introduce the following predicates.

$$
\begin{array}{lcl}
root(p) & \equiv & Root_p = p \wedge Dis_p = 0 \\
child(p,\, q) & \equiv & Root_p = Root_q > p \wedge Par_p = q \in Neigh_p \wedge Dis_p = Dis_q + 1 \\
tree(p) & \equiv & root(p) \ \vee \ \exists q : child(p,\, q) \\
lmax(p) & \equiv & \forall q \in Neigh_p : Root_p \geq Root_q \\
sat(p) & \equiv & tree(p) \wedge lmax(p)
\end{array}
$$

The intended postcondition of the algorithm is ψ, defined by $\forall p : sat(p)$.

var $Root_p$, Par_p , Dis_p ; (* Describe tree structure *)
Req_p, $From_p$, To_p, Dir_p ; (* Request forwarding *)

$\mathbf{B}_p$: (* Become root *)
{ $\neg tree(p)$ }
$Root_p := p$; $Dis_p := 0$;
$Req_p := p$; $To_p := q$; $Dir_p := Ask$

$\mathbf{A}_p$: (* Ask permission to join *)
{ $tree(p) \wedge \neg lmax(p)$ }
select $q \in Neigh_p$ with maximal value of $Root_q$;
$Req_p := p$; $From_p := p$; $To_p := q$; $Dir_p := Ask$

$\mathbf{J}_p$: (* Join tree *)
{ $tree(p) \wedge \neg lmax(p) \wedge grant(To_p, p)$ }
$Par_p := q$; $Root_p := Root_q$; $Dis_p := Dis_q + 1$;
$Req_p := From_p := To_p := Dir_p := undef$

$\mathbf{C}_p$: (* Clear request variables *)
{ $sat(p) \wedge \neg\exists q : forw(p, q) \wedge \neg idle(p)$ }
$Req_p := From_p := To_p := Dir_p := undef$

$\mathbf{F}_p$: (* Forward request *)
{ $sat(p) \wedge idle(p) \wedge asks(q, p)$ }
$Req_p := Req_q$; $From_p := q$; $To_p := Par_p$

$\mathbf{G}_p$: (* Grant join request *)
{ $sat(p) \wedge root(p) \wedge forw(p, q) \wedge Dir_p = Ask$ }
$Dir_p := Grant$

$\mathbf{R}_p$: (* Relay grant *)
{ $sat(p) \wedge grant(Par_p, p) \wedge Dir_p = Ask$ }
$Dir_p := Grant$

Algorithm 10.4: THE SPANNING TREE ALGORITHM.

Lemma 10.7 *ψ implies that the edges $\{(p, q) : child(p, q)\}$ form a spanning tree with the largest process as the root.*

Proof. As $lmax(p)$ is satisfied for all p and the network is connected, all processes have the same value of *Root*. The process p_0 with minimal value of *Dis* does not have a neighbor q with $Dis_p = Dis_q + 1$, so $root(p_0)$ holds and the common value of the *Root* variables is p_0. But then, for all $p \neq p_0$, $child(p)$ holds, and, because $child(p, q) \Rightarrow (Dis_p > Dis_q)$, the *child*-relation is acyclic. The result follows. □

Description of the Algorithm. In order to establish ψ, process p with $sat(p)$ true never changes its variables describing the tree structure (see Alg. 10.4). Process p with $sat(p)$ false attempts to establish $sat(p)$ by becoming a child of its neighbor with the highest value of *Root* (action $\mathbf{J}_p$).

Call the value of $Root_q$ a *false root* if there is no process with that identity in the network; false roots may exist initially but are not created during execution. A main problem in the

design of the algorithm is to prevent processes from becoming the child of a process with a false root infinitely often. To this end, joining q's tree by p is done in three steps. First, p becomes a root (action $\mathbf{B}_p$), then asks for join permission (action $\mathbf{A}_p$), and finally joins when q grants the permission (action $\mathbf{J}_p$).

The remaining four actions ($\mathbf{C}_p$, $\mathbf{F}_p$, $\mathbf{G}_p$, and $\mathbf{R}_p$) implement the request/reply mechanism and are only executed by processes p with $sat(p)$. Variable Req_p contains the process whose join request p is currently processing; $From_p$ is the neighbor from whom p read the request; To_p is the neighbor to which p forwards it; and Dir_p (values Ask and $Grant$) indicates whether the request was granted. We define the following predicates.

$$
\begin{array}{lcl}
idle(p) & \equiv & Req_p = From_p = To_p = Dir_p = undef \\
asks(p, q) & \equiv & ((root(p) \wedge Req_p = p) \vee child(p, q)) \\
 & & \wedge To_p = q \wedge Dir_p = Ask) \\
forw(p, q) & \equiv & Req_p = Req_q \wedge From_p = q \wedge To_q = p \wedge To_p = Par_p \\
grant(p, q) & \equiv & forw(p, q) \wedge Dir_p = Grant
\end{array}
$$

Process p clears the variables for request processing (action $\mathbf{C}_p$) if it is not currently processing a request and the variables are not undefined already. Then, if p is idle but has a neighbor q with a request for p (a root that wants to join, or a child of p forwarding a request), process p may start forwarding the request (action $\mathbf{F}_p$). If p is a root and forwards a request, it will grant it (action $\mathbf{G}_p$), and if p forwards the request to its parent and the parent grants it, then p relays the grant (action $\mathbf{R}_p$).

Correctness of the Algorithm. Afek *et al.* [AKY90] argue the correctness of the algorithm by showing properties of executions using behavioral reasoning.

The request/reply mechanism ensures that only finitely often a process can join a tree with a false root, because the non-existent root does not grant requests, and only finitely many false grants exist initially. Then, no *Root* variable contains a false root forever; the process with smallest value of *Dis* containing a false root does not satisfy *tree*, and will reset *Root* to its own identity. It follows that eventually there are no false roots, and also that eventually the process with highest identity will be a root, for this is the only way to satisfy *sat* in the absence of higher *Root* values.

A node that does not belong to the tree rooted at the highest node but does have a neighbor in that tree does not satisfy the *sat* predicate and will attempt to join, while nodes in the tree never leave it. As all incorrect requests eventually disappear from the tree, addition of nodes to the tree continues until the tree spans the whole network.

Chapter 11

A Self-stabilizing extension

Written by *P.I.A. van der Put.*

Katz and Perry [KP90] investigated the possibility of extending an arbitrary program into a self-stabilizing one. The communication topology is assumed to be a strongly connected graph.

They present an extension using a self-stabilizing snapshot algorithm and a self-stabilizing reset algorithm. It is assumed that a predicate is given that recognizes all legal global states of the arbitrary (basic) program.

11.1 Model and Definitions

An asynchronous message-passing network is considered. Processes are connected by FIFO communication channels. There are no bounds on either relative process speeds, message delivery time, or channel capacities.

It is assumed that every message sent is eventually received and that every statement whose guard remains true is eventually selected for execution.

Definition 11.1 (Global state) *A local state of a process is an assignment of values to the local variables and the local program counter. A global state of a system is the cross product of the local states of the processes plus the contents of the channels in the system.*

Definition 11.2 (Semantics) *An execution sequence of program P is a possible infinite sequence of global states in which each element follows from its predecessor by execution of a single atomic step of P. The semantics sem(P) of P is the set of all possible execution sequences of P.*

Note that no assumption is made about the initial state of an execution sequence, except that all values are from the appropriate domain. Channels may contain arbitrary messages in the initial state.

Definition 11.3 (Legal semantics) *Normal initial states are the states in which the local program counter of each of the processes is 0 and all channels are empty. The legal semantics legsem(P) of program P is the set of execution sequences which start with a normal initial state. Every global state in a sequence from legsem(P) is defined as legal.*

Definition 11.4 (Self-stabilizing program) *Program P is self-stabilizing if each sequence in sem(P) has a non-empty suffix that is identical to a suffix of some sequence in legsem(P).*

Definition 11.5 (Extension) *Program Q is an extension of program P if for each global state in legsem(Q) there is a projection onto the variables and messages of P such that the resulting set of sequences is identical to legsem(P), upto stuttering.*

Definition 11.6 (Self-stabilizing extension) *Program Q is a self-stabilizing extension of program P if Q is an extension of P and Q is self-stabilizing.*

If P is self-stabilizing then from some point on every computation is identical to a legal one. If Q is an extension of P then Q may have more variables and thus more states than P. Note that no process can ever "know" whether the system is in a legal global state. That knowledge would be recorded in a local variable whose value may be corrupted.

11.2 The Self-stabilizing Extension (SSE)

An arbitrary basic program P is extended by imposing a self-stabilizing control program on P. The control program interleaves steps of P with steps of its own. The control program repeatedly

1. takes a global snapshot
2. tests whether the snapshot represents a legal global state
3. if not, globally resets the system

The control program is more or less a centralized algorithm. One dedicated process is distinguished. The dedicated process collects all data which comprise a global snapshot, decides and issues - if necessary - messages which result in a global reset. This explains the assumption (or requirement) that the topology is a strongly connected graph.

The self-stabilizing global (multiple) snapshot algorithm is built on the non-self-stablizing local (single) snapshot algorithm of Chandy and Lamport [CL85].

11.2.1 Local snapshot algorithm

The local snapshot algorithm is invoked by each process i upon receiving a special kind of message (called a marker). The first marker received by process i causes i to save its local portion of P's state in a variable, begin recording basic messages on each incoming channel and propagate a marker to its neighbors (via outgoing channels). Process i terminates with recording this information when i has received one marker on each of its incoming channels.

A process i which terminates with recording its local snapshot relays its recorded information to the dedicated process. The local snapshots are relayed by means of report messages. A global snapshot consists of the local snapshots of all processes in the system.

11.2.2 Global snapshot algorithm

The dedicated process induces the (other) processes to take a local snapshot by sending waves of token messages (markers). It may send such a wave spontaneously, in order to avoid deadlock which might occur as follows. In the current self-stabilizing model a process may be under the false impression that it sent a message that was never actually sent. Waiting for a response to such a message may cause deadlock.

Different global snapshots are discerned by means of an (unbounded) iteration number. Having several iteration numbers in the system at a time may result in deadlock as follows. A state with several iteration numbers may evolve into a state with no report at the dedicated process and empty channels. This deadlock is avoided by means of the aforementioned spontaneous waves of token messages (Since each wave starts with an increased iteration number it is guaranteed that eventually a state will be reached in which the dedicated process do receives matching report messages).

A process can detect whether it received a certain message before. It examines the list of process identifiers in the message. A process updates this list in a message just before sending the message. Old messages are ignored. This prevents infinite propagation of any message (i.e. one message leading to infinitely many messages).

11.2.3 Testing a global snapshot

It is assumed that a predicate exists which determines whether an obtained snapshot represents a legal global state. In general it is impossible to give such a predicate, but for some specific basic programs it is possible. This predicate is applied by the dedicated process.

11.2.4 Reset algorithm

Katz and Perry give a sketchy description of the reset algorithm. The main idea is to use the same mechanisms as in the global snapshot algorithm. The token messages in the global snapshot algorithm become vanilla token messages. The token messages in the global reset algorithm become reset token messages.

A process i which receives a reset token message resets its state to some default legal state. This default legal state is derived from some global default legal state which in turn is derived from the basic program P. It may pose a problem to find such a default legal state.

11.3 Another reset algorithm

Another self-stabilizing reset algorithm was proposed by Arora and Gouda [AG90]. Arora and Gouda investigated the possibility of extending an arbitrary program into a program with a self-stabilizing reset part. They present a self-stabilizing reset algorithm together with the "interface" with the original program (The original program is referred to as the application).

A shared memory model is used. At the end it is argued that (a variant of) the self-stabilizing reset algorithm also works in a message-passing network provided that the channels have capacity one.

11.3.1 Model and definitions

Channels may fail and subsequently be repaired. Processes exhibit fail-stop failures and subsequently come up. A process is either up or down. So the reset algorithm allows for topological changes. It is assumed that each up process i is always kept informed of the set $N.i$ of its current neighbors. It is also assumed that the network remains connected.

It is assumed that process identifiers are ordered (and unique). So at any point in time there exists one up process which has the maximal process identifier (of all up processes) in the system.

It is assumed that an (correct and fixed) upper bound on the number of "active" processes is known to all processes.

The definition of a self-stabilizing program is about the same as the definition used by Katz and Perry. The aimed effect of the reset algorithm is resuming the execution of the system from a global state reachable by some system computation from a predefined given global state.

11.3.2 Reset algorithm

The reset algorithm consists of three self-stabilizing layers. The lowest layer maintains for each process its neighbors. The middle layer maintains a spanning tree of all (up) processes in the system. The highest layer is concerned with relaying messages up and down the spanning tree.

An arbitrary (up) process requests a reset (caused by the application). The wave layer relays this request to the root and the root informs all processes of a reset and waits for the completion of the reset.

The root of the spanning tree functions somewhat like the dedicated process before. The root process may go down. Another process can become the root process. In contrast, the dedicated process is fixed and is not allowed to go down. If the root process goes down, a different spanning tree evolves.

The middle layer also maintains for each (up) process the name of the root. After self-stabilization one process is known in all the (up) processes. So the lowest and the middle layer combined form a self-stabilizing leader election algorithm.

In order to avoid communication between a process that has been reset and a process that has not been reset yet session numbers are introduced. The highest layer maintains a session number for each (up) process. When no reset is in progress all session numbers are equal. Each reset of a process is accompanied by incrementing the session number of the process. Two neighboring processes don't communicate unless they have the same session number. The reset algorithm involves unbounded session numbers. An alternative with bounded session numbers is also mentioned.

11.4 Discussion

The self-stabilizing extension is characterized by iteratively taking a global snapshot. This involves an unbounded iteration number and repeated spontaneous waves of messages. The repeated spontaneous waves of messages form an essential part in the construction of the extension. An alternative with a bounded iteration number is presented. In this bounded version it is assumed that there is a bounded number of messages in the channels initially.

The convergence time of different versions of a self-stabilizing extension are discussed (see [KP90]). In general there is no way to bound the number of messages used for a snapshot. However, the maximum number of "critical" steps that must be executed before a legal state is reached is linear in the maximal difference between iteration numbers of process 0 and any other process.

No measure is provided concerning the number of steps of the control program relative to the number of steps of the basic program. So it is hard to discuss complexity and fault tolerance of the extension.

A version of the extension may be considered which handles topological changes (i.e. failures (and repairs) of processes and channels). This introduces some more problems. Detecting the termination of a local snapshot may become tricky. And ensuring that report messages reach the dedicated node may become hard. If the dedicated node fails another node should be elected to perform as the (unique) dedicated node.

The reset algorithm of Arora and Gouda may be used as the reset part in the global snapshot algorithm of Katz and Perry. It would be nice if the root of the computed spanning tree and the dedicated process were the same. This can be obtained by requiring that the dedicated process has (and keeps) the maximal process name of the entire network.

Arora and Gouda state that a self-stabilizing global snapshot algorithm can be obtained by means of minor modifications to their reset algorithm. This implies that the resulting global snapshot algorithm would allow for topological changes. The restriction would be that in this case the channels in the network have capacity one.

Part IV

Atomicity

Chapter 12

Wait-Free Synchronization

Written by *H. R. Baan.*

A *concurrent object* is a data structure shared by concurrent processes. A *wait free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. Constructing wait-free implementations of one concurrent data object from another, is a fundamental problem of wait-free synchronization. In [Her91] a simple and general technique, based on reduction to a consensus protocol, for proving statements of the form "there is no wait-free implementation of X by Y" is presented.

It is also shown that there do exist simple universal objects from which one can construct a wait-free implementation of any object. A simple test for universality is given, showing that an object is universal in a system of n processes iff it has a consensus number of at least n. A machine architecture or programming language is computationally powerful enough to support arbitrary wait-free synchronization iff it provides a universal object as a primitive.

12.1 The Model

The model of computation consists of a collection of sequential threads of control called *processes* that communicate through shared data structures called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the only means to manipulate that object. Each process applies a sequence of operations to objects, issuing an invocation and receiving the associated response. The basic correctness for concurrent systems is *linearizability*: although operations of a concurrent process may overlap, each operation appears to take effect instantaneously at some point between its invocation and response. In particular, operations that do not overlap take effect in their "real-time" order.

12.1.1 I/O automata

Processes and objects are modelled using a simplified form of I/O automata. An I/O automaton A is a nondeterministic automaton with the following components: $States(A)$ is a set of states, including a distinguished set of starting states. $In(A)$, $Out(A)$ and $Int(A)$ are

the sets of input, output, and internal events of the automaton, respectively. $Step(A)$ is a transition relation (s', e, s), where s and s' are states, and e is an event.

If (s', e, s) is a step, e is *enabled* in s'. Input events are always enabled. An *execution fragment* of automaton A is a possibly infinite sequence $s_0, e_1, s_1, e_2, \ldots$ of alternating states and events constructed by the use of steps. An *execution* is an execution fragment where s_0 is a starting state. A *history fragment* is the subsequence of events occuring in an execution fragment, and a *history* is the subsequence occuring in an execution.

New I/O automata can be constructed by composing a set of compatible I/O automata (i.e., a set of I/O automata that share no output or internal events). Composing is done in a straightforward manner. If H is a history of a composite automaton and A is a component automaton, $H|A$ denotes the subhistory of H consisting of events of A.

12.1.2 Concurrent systems

A *concurrent system* is a set of processes and a set of objects. Processes represent sequential threads of control, and objects represent data structures shared by processes. A process P is an I/O automaton with output events $invoke(P, op, X)$, where op is an operation of object X, and input events $respond(P, res, X)$, where res is a result value. A process history is well formed if it begins with an invocation and alternates matching invocations and responses. An invocation is *pending* if it is not followed by a matching response. An object X has input events $invoke(P, op, X)$, and output events $respond(P, res, X)$.

A concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is an I/O automaton composed from processes $P_1, \ldots, P_n$ and objects $A_1, \ldots, A_m$. A history of a concurrent system is well formed if each $H|P_i$ is well formed, and a concurrent system is well formed if each of its histories is well formed.

An execution is sequential if it starts with an invocation, and alternates with matching invocations and responses. A history is sequential if it is derived from a sequential execution. If we look at sequential histories only, the behaviour of an object can be described by giving pre- and postconditions for each operation. Such specification is refered to as a sequential specification. Only *total* sequential specifications (i.e., if an object has a pending invocation, then it has a matching response enabled) are regarded.

Each history H induces a partial real-time order $\prec_H$ on its operations. *Concurrent operations* are those operations that are unrelated by $\prec_H$. If H is sequential, $\prec_H$ is total. A concurrent system is linearizable if, for each history H, there exists a sequential history S such that $\forall P_i$, $H|P_i = S|P_i$ and $\prec_H \subset \prec_S$. A concurrent object A is linearizable if for every history H of every concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, $H|A_j$ is linearizable.

Unlike related correctness conditions such as sequential consistency and strict serializability, linearizability is a *local* property: a concurrent system is linearizable iff each individual object is linearizable.

12.1.3 Implementations

An implementation of an object A is a concurrent system $\{F_1, \ldots, F_n; R\}$, where the F_i are called *front-ends*, and R the *representation* object. Informally R is the data structure that implements A, and F_i is the procedure called by process P_i to execute an operation. The external events of the implementation are just the external events of A: each $invoke(P_i, op, A)$ is an input event of F_i, and each $response(P_i, res, A)$ is an output event of F_i. The imple-

mentation has the following internal events: each input event $invoke(F_i, op, R)$ is composed with the matching output event of F_i, and each output event $respond(F_i, res, R)$ is composed with the matching input event of F_i. Front-ends share no events; they communicate indirectly through R.

Let I_j be an implementation of A_j; then I_j is *correct*, if for every history H of every system $\{P_1, \ldots, P_n; A_1, \ldots, I_j, \ldots, A_m\}$, there exists a history H' of $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, such that $H|\{P_1, \ldots, P_n\} = H'|\{P_1, \ldots, P_n\}$.

An implementation is *wait-free* if the following are true: (1) it has no history in which an invocation of P_i remains pending across an infinite number of steps of F_i, (2) if P_i has a pending invocation in state s, then there exists a history fragment starting from s, consisting entirely of events of F_i and R, that includes the response to that invocation. An implementation is *bounded wait-free* if there exists N such that there is no history in which an invocation of P_i remains pending across N steps of F_i.

R *implements* A if there exists a wait-free implementation $\{F_1, \ldots, F_n; R\}$ of A. From the definition of *implements*, it follows that this is a reflexive partial order on the universe of objects.

12.1.4 I/O automata and distributed systems

The I/O automata model can be mapped on the model normally used for distributed systems, by noticing the following resemblances.

- An automaton is a process/node;
- The set $States(A)$ is the set of states a process can have;
- The events from the sets $In(A)$ and $Out(A)$ are the *recieve* and *send* events respectively;
- The state transitions of an automaton are the state transitions of the process;
- The other characterizations of the model can be viewed analogous.

12.2 Impossibility Results

Informally, a *consensus protocol* is a system of n processes that communicate through a set of shared objects $\{X_1, \ldots, X_m\}$. The processes each start with an input value from some domain $\mathcal{D}$; they communicate with one another by applying operations to the shared objects; and they eventually agree on a common input value and halt. A consensus protocol is required to be (1) consistent: distinct processes never decide on distinct values, (2) wait-free: each process decides after a finite number of steps, and (3) valid: the common decision value is the input to some process. In other words: a *consensus object* provides a single operation:

`decide(input: value) returns(value).`

A wait-free linearizable implementation of a consensus object is called a consensus protocol. X solves n-process consensus, if there exists a consensus protocol $\{F_1, \ldots, F_n; W, X\}$, where W is a set of read/write registers, and W and X may be initialized to any state.

Definition 12.1 *The consensus number for X is the largest n for which X solves n-process consensus. If no largest n exists, the consensus number is said to be infinite.*

An immedeate consequence of this definition is that if Y implements X, and X solves n-process consensus then Y also solves n-process consensus.

Theorem 12.2 *If X has consensus number n, and Y has consensus number $m < n$, then there exists no wait-free implementation of X by Y in a system of more than m processes.*

Proof. All front-end and object automata are compatible by definition, and thus their composition is well defined. Let $\{F_1, \ldots, F_k; W, X\}$ be a consensus protocol, where $k > m$ and W is a set of read/write registers. Let $\{F'_1, \ldots, F'_k; Y\}$ be an implementation of X. It is easily checked that $\{F_1, \ldots, F_n; W, \{F'_1, \ldots, F'_k; Y\}\}$ is wait-free, and because composition is associative, it is identical to $\{F_1 \cdot F'_1, \ldots, F_n \cdot F'_n; W, Y\}$, where $F_i \cdot F'_i$ is the composition of F_i and F'_i. Since the former is a consensus protocol, so is the latter, contradicting the hypothesis that Y has consensus number m. □

For a consensus protocol the following definitions are made: A protocol state is *bivalent* if either decision value is still possible, otherwise it is *univalent.* An x-valent state is univalent with eventual decision value x. A *decision step* is an operation that caries a protocol from a bivalent to a univalent state.

Example: Atomic Read/Write registers

Atomic read/write registers can only solve 1-process consensus, as is shown below.

Theorem 12.3 *Atomic read/write registers have consensus number 1.*

Proof. Assume there exists a two-process consensus protocol implemented from atomic read/write registers.

If the processes have different input values, the validity condition implies that the initial state is bivalent (because in the initial state, every process can make its initial value the decision value). Consider the following sequential execution, starting from the initial state. In the first stage, P executes a sequence of operations until it reaches a state where the next operation will leave the protocol in a univalent state (By the assumption of wait-free consensus, P must eventually reach such a state). In the second stage, Q executes a sequence of operations until it reaches a similar state, and in successive stages, P and Q alternate sequences of operations until each is about to make a decision step. Because the protocol cannot run forever, it must eventually reach a bivalent state s in which any subsequent operation of either process is a decision step. Suppose P's operation carries the protocol to an x-valent state, and Q's operation carries the protocol to a y-valent state, where x and y are distinct. The following cases can be distinguished:

1. P's decision step is to read a shared register.

   Let s' be the state directly following this read. The protocol also has a history fragment starting from s, consisting entirely of operations of Q, yielding decision value y. Because s and s' differ only in the internal state of P, they have the same history fragment starting from s', and this is impossible because s' is x-valent.

2. Both processes write to different registers.

   The state that results if Q's write immedeatly follows P's, is the same if the writes occured in a different order, which is impossible because one state is x-valent, and the other is y-valent.

3. Both processes write to the same register.

   Let s' be the state directly following P's write; this is x-valent. There exists a history fragment starting from s' consisting only of operations of P yielding decision value x. Let s'' be the state reached if Q's write is overwritten by P's; this state is y-valent. Because s' and s'' differ only in the internal state of Q, they have the same history fragment starting from s'', and this is impossible because s'' is y-valent.

□

Example: *compare&swap* registers

With the use of an atomic *compare&swap* register, it is possible to reach consensus for an infinite number of processes. See Figures 12.1 and 12.2 for a definition of *compare&swap*, and the protocol used. Note that Figure 12.1 is just a definition of the function, and that *compare&swap* should be executed atomically.

```
compare&swap(r: register, old: value, new: value) returns(value)
  previous:=r
  if previous = old then
    r:=new
  end if
  return previous
end compare&swap
```

Figure 12.1: *compare&swap* definition

```
decide(input: value) returns(value)
  first:=compare&swap(r,bottom,input)
  if first=bottom then
    return input
  else
    return first
  end if
end decide
```

Figure 12.2: *compare&swap* protocol

Theorem 12.4 *A compare&swap register has infinite consensus number.*

Proof. In the protocol shown in Figure 12.2, the processes share a register r initialized to $\perp$ (`bottom` in the figure). Each process tries to change this value to their own; the decision value is established by the process that succeeds (which is the first to attempt a change of the value).

operation	consensus number
Read/Write registers	1
FIFO queues	2
any non trivial Read-Modify-Write	at least 2, this is exactly 2 if the operations apply functions from an interfering set F.
Atomic m-register assignment	$2m-2$
compare&swap	infinite (see Theorem 12.4).
Augmented queue[a]	infinite
Array of registers with *move*[b]	infinite
Array of registers with *memory-to-memory swap*	infinite

[a] An augmented queue is a queue with an extra **peek** operation, which looks at the first item in the queue (but does not remove it).

[b] *move* atomically copies the value of one register to another.

Table 12.1: List of consensus numbers

The protocol is wait-free because it contains no loops, it is consistent because (1) $r \neq \perp$ is a postcondition of *compare&swap*, and (2) for any $v \neq \perp$, the assertion $r = v$ is stable — once it becomes true, it remains true, and it is valid because if $r \neq \perp$, then r contains some process's input. □

More consensus numbers

In Table 12.1 a list of consensus numbers can be found (these numbers are given without proof).

12.3 Universality Results

A universal object is an object that can implement any other object. A consensus object with consensus number n is a universal object in a system of n or fewer processes. The basic idea behind this is to represent the object as a linked list, where the sequence of cells represents the sequence of operations applied to the object. An operation is "executed" by threading a new cell on the end of the list. The real execution of a statement takes place when the corresponding cell reaches the beginning of the list.

The object's behaviour may be specified by the following relation:

$$apply \subset \text{INVOC} \times \text{STATE} \times \text{STATE} \times \text{RESULT},$$

with INVOC the object's domain of invocations, RESULT its domain of results, and STATE its domain of states. The specification means that if $< p, s, s', r > \in apply$, applying operation p in state s, leaves the object in state s' and returns result value r. Note: *apply* is non-deterministic (therefore it is a relation rather than a funnction). The notation $apply(p, s)$ is used to denote an arbitrary pair $< s', r >$ such that $< p, s, s', r > \in apply$.

12.3.1 The algorithm

An object is represented by a doubly linked list of cells having the following fields:

1. *seq* is the cell's sequence nuber in the list. This field is zero if the cell is initialized, but not yet threaded onto the list, and otherwise it is positive. Sequence numbers for successive cells in the list increase by one.
2. *inv* is the invocation (operation name and values).
3. *new* is a consensus object whose value is the pair $< new.state, new.result >$. *new.state* is the object's state following the operation and *new.result* is the result value, if any. See further on for an explanation.
4. *before* is a pointer to the previous cell in the list, and is only used for free storeage management.
5. *after* is a consensus object whose value is a pointer to the next cell in the list. See further on for an explanation.

For cells c, d, $\max(c, d)$ returns the cell with the higher sequence number.

Initially the object is represented by a unique *anchor* cell with sequence number 1, holding a creation operation and an initial state.

The processes share the following data structures:

1. *Announce* is an n-element array, whose Pth element is a pointer to the cell P is currently trying to thread onto the list. Initially all elements point to the anchor cell.
2. *Head* is a n-element array, whose Pth element is a pointer to the last cell in the list that P has observed. Initially all elements point to the anchor cell.

Let $\max(head)$ denote the maximum $\max(head[1].seq, \ldots, head[n].seq)$.

The protocol for the universal construction can be found in Figure 12.3. In this figure "`v: T := e`" declares and initializes a variable v of type T to value e, and the type "`*cell`" means "pointer to cell".

The protocol of Figure 12.3 works as follows.
First P allocates and initializes a cell to represent the operation (<1>). It stores a pointer to the cell in $announce[P]$ (<2>), ensuring that if P itself does not succeed in threading its cell onto the list, some other process will. To locate a cell near the end of the list, P scans the *head* array, setting $head[P]$ to the cell with the maximal sequence number (<3>). P then enters the main loop of the protocol (<4>), which it executes until its own cell has been threaded onto the list (then its own sequence number is non zero). P chooses a process to "help" (<6>), and checks whether that process has an unthreaded cell (<7>). If so, then P will try to thread it; otherwise it tries to thread its own cell. P tries to set $head[P].after$ to point to the cell it is trying to thread (<8>). The *after* field must be a consensus object to ensure that only one process succeeds in setting it. Whether or not P succeeds, it then initializes the remaining fields of the next cell in the list. Because the operation may be non-deterministic, different processes may try to set the *new* field to different values, so this field also has to be a consensus object (<9>). The values of the other fields are computed deterministically, so they can simply be written as atomic registers (<10> and <11>). A process *threads* a cell <7> if the *decide* operation alters the value of the *after* field, and it announces a cell at <2> when it stores the cell's address in *announce*.

```
universal(what: INVOC) returns(RESULT)
  mine: cell := [ seq: 0,                                    < 1>
                  inv: what,
                  new: create(consensus_object)
                  before: null
                  after: create(consensus_object)
                ]
  announce[P] := mine                                        < 2>
  for each process Q do                                      < 3>
    head[P] := max(head[P],head[Q])
  end for
  while announce[P].seq = 0 do                               < 4>
    c: *cell := head[P]                                      < 5>
    help : *cell := announce[(c.seq mod n) + 1]              < 6>
    if help.seq = 0 then                                     < 7>
      prefer := help
    else
      prefer := announce[P]
    end if
    d := decide(c.after,prefer)                              < 8>
    decide(d.new,apply(d.inv,c.new.state))                   < 9>
    d.before := c                                            <10>
    d.seq = c.seq + 1                                        <11>
    head[P] := d                                             <12>
  end while
  head[P] := announce[P]                                     <13>
  return (announce[P].new.result)                            <14>
end universal
```

Figure 12.3: A universal construction

Theorem 12.5 *The protocol in Figure 12.3 is correct and bounded wait-free.*

The universal construction costs $O(n^3)$ memory cells to represent the object, and $O(n^3)$ worst case time to execute each operation.

```
T1 (move $100 from A to B):
Start Transaction
  Read CreditA
  CreditA := CreditA - 100
  Write CreditA
  Read CreditB
  CreditB := CreditB + 100
  Write CreditB
End Transaction
```

```
T2 (show credits of A & B):
Start Transaction
  Read CreditA
  Read CreditB
  Show Credits
End Transaction
```

Figure 12.4: The two transactions of the bank acount example

Note that allthough the protocol implements a universal object, this does not assure that everything can be implemented in a (bounded) wait-free fashion. Take for instance the bank account example (see Figure 12.4 for the two transactions involved); this example cannot be scheduled in a wait-free fashion without showing the wrong credits!

Bibliography

[AAPS87] AFEK, Y., AWERBUCH, B., PLOTKIN, S., AND SAKS, M. Local management of a global resource in a communication network. In *28th Foundations of Computation Theory* (1987), pp. 347–357.

[ABND+90] ATTIYA, H., BAR-NOY, A., DOLEV, D., PELEG, D., AND REISCHUK, R. Renaming in an asynchronous environment. *J. ACM* **37** (1990), 524–548.

[AG90] ARORA, A., AND GOUDA, M. Distributed reset. In *Foundations of Software Technology and Theoretical Computer Science* (1990), vol. 472 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 316–331.

[AKP91] AWERBUCH, B., KUTTEN, S., AND PELEG, D. Efficient deadlock-free routing. In *10th Symp. on Principles of Distributed Computing* (Montreal, 1991), pp. 177–188.

[AKY90] AFEK, Y., KUTTEN, S., AND YUNG, M. Memory-efficient self stabilizing protocols for general networks. In *4th Int. Workshop on Distributed Algorithms* (Bari, 1990), J. van Leeuwen and N. Santoro (Eds.), vol. 486 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–28.

[AP90] AWERBUCH, B., AND PELEG, D. Sparse partitions. In *31 Foundations of Computation Theory* (1990), pp. 503–513.

[AP92] AWERBUCH, B., AND PELEG, D. Routing with polynomial communication-space trade-off. *SIAM J. Discr. Math.* **5**, 2 (1992), 151–162.

[AP93] AWERBUCH, B., AND PELEG, D. Concurrent online tracking of mobile users. *J. ACM* (1993). To appear.

[AR92] AFEK, Y., AND RICKLIN, M. Sparser: A paradigm for running distributed algorithms. In *6th Int. Workshop on Distributed Algorithms* (Haifa, 1992), A. Segall and S. Zaks (Eds.), vol. 647 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–10.

[Awe85] AWERBUCH, B. Complexity of network synchronization. *J. ACM* **32** (1985), 804–823.

[BP88] BURNS, J. E., AND PACHL, J. Uniform self-stabilizing rings. In *Aegean Workshop on Computing* (1988), vol. 319 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 391–400.

[BT85] BRACHA, G., AND TOUEG, S. Asynchronous consensus and broadcast protocols. *J. ACM* **32** (1985), 824–840.

[CHT92] CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. In *11th Symp. on Principles of Distributed Computing* (Vancouver, 1992), pp. 147–158.

[CL85] CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* **3**, 1 (1985), 63–75.

[DDS87] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM* **34** (1987), 77–97.

[Dij82] DIJKSTRA, E. W. Self-stabilization in spite of distributed control. In *Selected Writing on Computing: A Personal Perspective.* Springer-Verlag, Berlin, 1982, pp. 41–46.

[Dij86] DIJKSTRA, E. W. A belated proof of self-stabilization. *Distributed Computing* **1** (1986), 5–6.

[Eck77] ECKSTEIN, D. *Parallel Processing Using Depth-First-Search and Breadth-First-Search.* PhD thesis, Dept. of Computer Science, Univ. of Iowa, Iowa City, 1977.

[FLP85] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM* **32** (1985), 374–382.

[Gal82] GALLAGER, R. G. Distributed minimum hop algorithms. Tech. Rep. LIDS-P-1175, M.I.T., Cambridge, Mass., 1982.

[GHS83] GALLAGER, R. G., HUMBLET, P. A., AND SPIRA, P. M. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst.* **5** (1983), 67–77.

[Her91] HERLIHY, M. P. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* **13**, 1 (1991), 124–149.

[HH92] HSU, S.-C., AND HUANG, S.-T. A self-stabilizing algorithm for maximal matching. *Inf. Proc. Lett.* **43**, 2 (1992), 77–81.

[Hum84] HUMBLET, P. M. Selecting a leader in a clique in $O(n \log n)$ messages. Internal memo, Lab. Inform. Decision Syst., M.I.T., 1984.

[IJ90] ISRAELI, A., AND JALFON, M. Self-stabilizing ring orientation. In *4th Int. Workshop on Distributed Algorithms* (Bari, 1990), J. van Leeuwen and N. Santoro (Eds.), vol. 486 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–14.

[IKWZ90] ITAI, A., KUTTEN, S., WOLFSTAHL, Y., AND ZAKS, S. Optimal distributed t–resilient election in complete networks. *IEEE Trans. Softw. Eng.* **SE–8**, 4 (1990), 415–420.

[KP90] KATZ, S., AND PERRY, K. J. Self-stabilizing extensions for message-passing systems. In *9th Symp. on Principles of Distributed Computing* (Quebec, 1990), pp. 91–102.

[Kru79] KRUIJER, H. S. M. Self-stabilization (in spite of distributed control) in tree-structured systems. *Inf. Proc. Lett.* **8** (1979), 91–95.

[KV88] KRANAKIS, E., AND VITÁNYI, P. M. B. Weigthed distributed match making (preliminary version). In *VLSI Algorithms and Architectures* (1988), Springer-Verlag, pp. 361–368.

[KV92] KRANAKIS, E., AND VITÁNYI, P. M. B. A note on weighted distributed match-making. *Mathematical Systems Theory* **25** (1992), 123–140.

[Lam78] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21** (1978), 558–564.

[Mae85] MAEKAWA, M. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.* **3** (1985), 145–159.

[MV88] MULLENDER, S. J., AND VITANYI, P. M. B. Distributed match-making. *Algorithmica* **3** (1988), 367–391.

[MW87] MORAN, S., AND WOLFSTAHL, Y. Extended impossibility results for asynchronous complete networks. *Inf. Proc. Lett.* **26** (1987), 145–151.

[Pel90] PELEG, D. Time-optimal leader election in general networks. *Journal of Parallel and Distributed Programming* **8**, 1 (1990), 96–99.

[Seg83] SEGALL, A. Distributed network protocols. *IEEE Trans. Information Theory* **IT–29** (1983), 23–35.

[Tel94] TEL, G. *Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, U.K., 1994.

Index