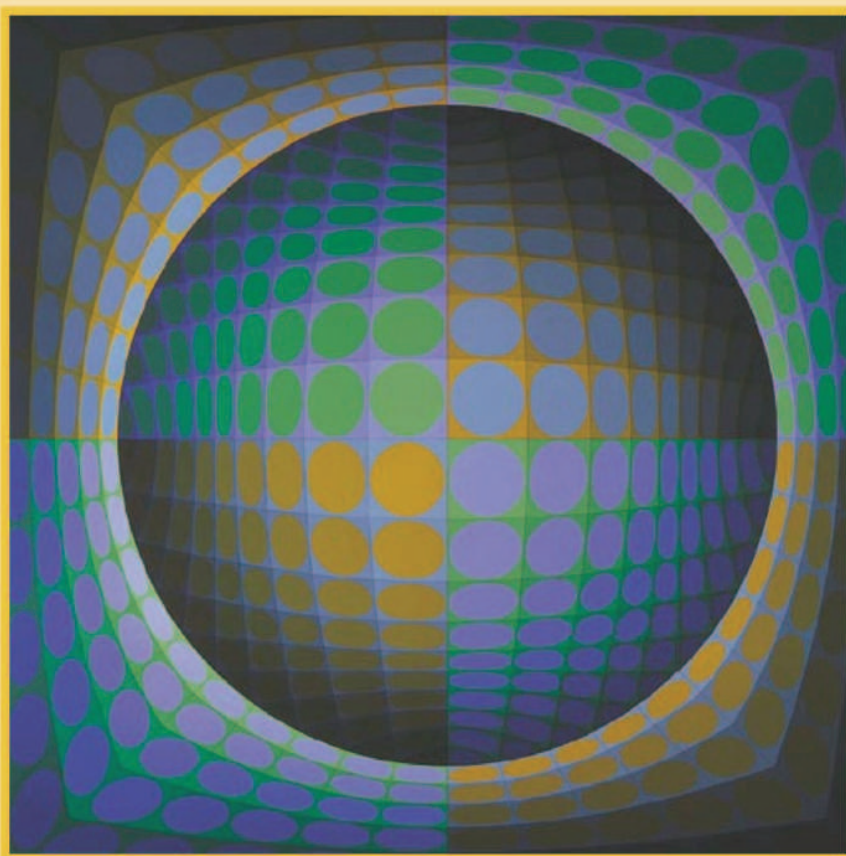


ALGORITHMS of Informatics



volume **2**

**ALGORITHMS
OF INFORMATICS**

Volume 2

**AnTonCom
Budapest, 2010**

This electronic book was prepared in the framework of project Eastern Hungarian Informatics Books Repository no. TÁMOP-4.1.2-08/1/A-2009-0046.

This electronic book appeared with the support of European Union and with the co-financing of European Social Fund.



Editor: Antal [Iványi](#)

Authors of Volume 1: László [Lovász](#) (Preface), Antal [Iványi](#) (Introduction), Zoltán [Kása](#) (Chapter 1), Zoltán [Csörnyei](#) (Chapter 2), Ulrich [Tamm](#) (Chapter 3), Péter [Gács](#) (Chapter 4), Gábor [Ivanyos](#) and Lajos [Rónyai](#) (Chapter 5), Antal [Járai](#) and Attila [Kovács](#) (Chapter 6), Jörg [Rothe](#) (Chapters 7 and 8), Csanád [Imreh](#) (Chapter 9), Ferenc [Szidarovszky](#) (Chapter 10), Zoltán [Kása](#) (Chapter 11), Aurél [Galántai](#) and András [Jeney](#) (Chapter 12),

Validators of Volume 1: Zoltán [Fülöp](#) (Chapter 1), Pál [Dömösi](#) (Chapter 2), Sándor [Fridli](#) (Chapter 3), Anna [Gál](#) (Chapter 4), Attila [Pethő](#) (Chapter 5), Lajos [Rónyai](#) (Chapter 6), János [Gonda](#) (Chapter 7), Gábor [Ivanyos](#) (Chapter 8), Béla [Vizvári](#) (Chapter 9), János [Mayer](#) (Chapter 10), András [Recski](#) (Chapter 11), Tamás [Szántai](#) (Chapter 12), Anna [Iványi](#) (Bibliography)

Authors of Volume 2: Burkhard [Englert](#), Dariusz [Kowalski](#), Gregorz [Malewicz](#), and Alexander [Shvartsman](#) (Chapter 13), Tibor [Gyires](#) (Chapter 14), Claudia [Fohry](#) and Antal [Iványi](#) (Chapter 15), Eberhard [Zehendner](#) (Chapter 16), Ádám [Balogh](#) and Antal [Iványi](#) (Chapter 17), János [Demetrovics](#) and Attila [Sali](#) (Chapters 18 and 19), Attila [Kiss](#) (Chapter 20), István [Miklós](#) (Chapter 21), László [Szirmay-Kalos](#) (Chapter 22), Ingo [Althöfer](#) and Stefan [Schwarz](#) (Chapter 23)

Validators of Volume 2: István [Majzik](#) (Chapter 13), János [Sztrik](#) (Chapter 14), Dezső [Sima](#) (Chapters 15 and 16), László [Varga](#) (Chapter 17), Attila [Kiss](#) (Chapters 18 and 19), András [Benczúr](#) (Chapter 20), István [Katsányi](#) (Chapter 21), János [Vida](#) (Chapter 22), Tamás [Szántai](#) (Chapter 23), Anna [Iványi](#) (Bibliography)

Cover art: Victor Vasarely, *Dirac*, 1978. With the permission of [Museum of Fine Arts](#), Budapest. The used film is due to [GOMA](#) ZRt.

Cover design by Antal [Iványi](#)

© 2010 AnTonCom Infokommunikációs Kft.

Homepage: <http://www.antoncom.hu/>

Contents

IV. COMPUTER NETWORKS	591
13. Distributed Algorithms	592
13.1. Message passing systems and algorithms	593
13.1.1. Modeling message passing systems	593
13.1.2. Asynchronous systems	593
13.1.3. Synchronous systems	594
13.2. Basic algorithms	595
13.2.1. Broadcast	595
13.2.2. Construction of a spanning tree	596
13.3. Ring algorithms	600
13.3.1. The leader election problem	600
13.3.2. The leader election algorithm	601
13.3.3. Analysis of the leader election algorithm	604
13.4. Fault-tolerant consensus	607
13.4.1. The consensus problem	607
13.4.2. Consensus with crash failures	608
13.4.3. Consensus with Byzantine failures	609
13.4.4. Lower bound on the ratio of faulty processors	610
13.4.5. A polynomial algorithm	610
13.4.6. Impossibility in asynchronous systems	611
13.5. Logical time, causality, and consistent state	612
13.5.1. Logical time	613
13.5.2. Causality	614
13.5.3. Consistent state	617
13.6. Communication services	619
13.6.1. Properties of broadcast services	619
13.6.2. Ordered broadcast services	621
13.6.3. Multicast services	625
13.7. Rumor collection algorithms	626
13.7.1. Rumor collection problem and requirements	626
13.7.2. Efficient gossip algorithms	627
13.8. Mutual exclusion in shared memory	634

- 13.8.1. Shared memory systems 634
- 13.8.2. The mutual exclusion problem 634
- 13.8.3. Mutual exclusion using powerful primitives 635
- 13.8.4. Mutual exclusion using read/write registers 636
- 13.8.5. Lamport’s fast mutual exclusion algorithm 640
- 14. Network Simulation 644**
- 14.1. Types of simulation 644
- 14.2. The need for communications network modelling and simulation 645
- 14.3. Types of communications networks, modelling constructs 647
- 14.4. Performance targets for simulation purposes 649
- 14.5. Traffic characterisation 652
- 14.6. Simulation modelling systems 660
- 14.6.1. Data collection tools and network analysers 660
- 14.6.2. Model specification 660
- 14.6.3. Data collection and simulation 660
- 14.6.4. Analysis 661
- 14.6.5. Network Analysers 662
- 14.6.6. Sniffer 669
- 14.7. Model Development Life Cycle (MDLC) 669
- 14.8. Modelling of traffic burstiness 675
- 14.8.1. Model parameters 680
- 14.8.2. Implementation of the Hurst parameter 681
- 14.8.3. Validation of the baseline model 683
- 14.8.4. Consequences of traffic burstiness 686
- 14.8.5. Conclusion 690
- 14.9. Appendix A 690
- 14.9.1. Measurements for link utilisation 690
- 14.9.2. Measurements for message delays 690
- 15. Parallel Computations 703**
- 15.1. Parallel architectures 705
- 15.1.1. SIMD architectures 705
- 15.1.2. Symmetric multiprocessors 706
- 15.1.3. Cache-coherent NUMA architectures: 707
- 15.1.4. Non-cache-coherent NUMA architectures: 707
- 15.1.5. No remote memory access architectures 708
- 15.1.6. Clusters 708
- 15.1.7. Grids 708
- 15.2. Performance in practice 709
- 15.3. Parallel programming 713
- 15.3.1. MPI programming 714
- 15.3.2. OpenMP programming 717
- 15.3.3. Other programming models 719
- 15.4. Computational models 720
- 15.4.1. PRAM 720
- 15.4.2. BSP, LogP and QSM 721

15.4.3. Mesh, hypercube and butterfly	722
15.5. Performance in theory	724
15.6. PRAM algorithms	728
15.6.1. Prefix	729
15.6.2. Ranking	735
15.6.3. Merge	737
15.6.4. Selection	741
15.6.5. Sorting	746
15.7. Mesh algorithms	749
15.7.1. Prefix on chain	749
15.7.2. Prefix on square	750
16. Systolic Systems	754
16.1. Basic concepts of systolic systems	755
16.1.1. An introductory example: matrix product	755
16.1.2. Problem parameters and array parameters	756
16.1.3. Space coordinates	757
16.1.4. Serialising generic operators	758
16.1.5. Assignment-free notation	759
16.1.6. Elementary operations	760
16.1.7. Discrete timesteps	760
16.1.8. External and internal communication	761
16.1.9. Pipelining	763
16.2. Space-time transformation and systolic arrays	764
16.2.1. Further example: matrix product	764
16.2.2. The space-time transformation as a global view	765
16.2.3. Parametric space coordinates	767
16.2.4. Symbolically deriving the running time	770
16.2.5. How to unravel the communication topology	770
16.2.6. Inferring the structure of the cells	771
16.3. Input/output schemes	773
16.3.1. From data structure indices to iteration vectors	774
16.3.2. Snapshots of data structures	775
16.3.3. Superposition of input/output schemes	776
16.3.4. Data rates induced by space-time transformations	777
16.3.5. Input/output expansion	777
16.3.6. Coping with stationary variables	778
16.3.7. Interleaving of calculations	779
16.4. Control	781
16.4.1. Cells without control	781
16.4.2. Global control	782
16.4.3. Local control	783
16.4.4. Distributed control	786
16.4.5. The cell program as a local view	790
16.5. Linear systolic arrays	794
16.5.1. Matrix-vector product	794
16.5.2. Sorting algorithms	795

16.5.3. Lower triangular linear equation systems	796
V. DATA BASES	798
17. Memory Management	799
17.1. Partitioning	799
17.1.1. Fixed partitions	800
17.1.2. Dynamic partitions	806
17.2. Page replacement algorithms	813
17.2.1. Static page replacement	815
17.2.2. Dynamic paging	822
17.3. Anomalies	824
17.3.1. Page replacement	825
17.3.2. Scheduling with lists	826
17.3.3. Parallel processing with interleaved memory	833
17.3.4. Avoiding the anomaly	837
17.4. Optimal file packing	837
17.4.1. Approximation algorithms	838
17.4.2. Optimal algorithms	841
17.4.3. Shortening of lists (SL)	842
17.4.4. Upper and lower estimations (ULE)	842
17.4.5. Pairwise comparison of the algorithms	843
17.4.6. The error of approximate algorithms	845
18. Relational Data Base Design	850
18.1. Functional dependencies	851
18.1.1. Armstrong-axioms	851
18.1.2. Closures	852
18.1.3. Minimal cover	855
18.1.4. Keys	857
18.2. Decomposition of relational schemata	859
18.2.1. Lossless join	860
18.2.2. Checking the lossless join property	860
18.2.3. Dependency preserving decompositions	864
18.2.4. Normal forms	867
18.2.5. Multivalued dependencies	872
18.3. Generalised dependencies	878
18.3.1. Join dependencies	878
18.3.2. Branching dependencies	879
19. Query Rewriting in Relational Databases	883
19.1. Queries	883
19.1.1. Conjunctive queries	885
19.1.2. Extensions	890
19.1.3. Complexity of query containment	898
19.2. Views	902
19.2.1. View as a result of a query	902
19.3. Query rewriting	905

19.3.1. Motivation	905
19.3.2. Complexity problems of query rewriting	910
19.3.3. Practical algorithms	913
20. Semi-structured Databases	932
20.1. Semi-structured data and XML	932
20.2. Schemas and simulations	934
20.3. Queries and indexes	939
20.4. Stable partitions and the PT-algorithm	945
20.5. $A(k)$ -indexes	952
20.6. $D(k)$ - and $M(k)$ -indexes	954
20.7. Branching queries	961
20.8. Index refresh	965
VI. APPLICATIONS	972
21. Bioinformatics	973
21.1. Algorithms on sequences	973
21.1.1. Distances of two sequences using linear gap penalty	973
21.1.2. Dynamic programming with arbitrary gap function	976
21.1.3. Gotoh algorithm for affine gap penalty	977
21.1.4. Concave gap penalty	977
21.1.5. Similarity of two sequences, the Smith-Waterman algorithm	980
21.1.6. Multiple sequence alignment	981
21.1.7. Memory-reduction with the Hirschberg algorithm	983
21.1.8. Memory-reduction with corner-cutting	984
21.2. Algorithms on trees	986
21.2.1. The small parsimony problem	986
21.2.2. The Felsenstein algorithm	987
21.3. Algorithms on stochastic grammars	989
21.3.1. Hidden Markov Models	989
21.3.2. Stochastic context-free grammars	991
21.4. Comparing structures	994
21.4.1. Aligning labelled, rooted trees	994
21.4.2. Co-emission probability of two HMMs	995
21.5. Distance based algorithms for constructing evolutionary trees	997
21.5.1. Clustering algorithms	998
21.5.2. Neighbour joining	1001
21.6. Miscellaneous topics	1005
21.6.1. Genome rearrangement	1006
21.6.2. Shotgun sequencing	1007
22. Computer Graphics	1012
22.1. Fundamentals of analytic geometry	1012
22.1.1. Cartesian coordinate system	1013
22.2. Description of point sets with equations	1013
22.2.1. Solids	1014
22.2.2. Surfaces	1014

22.2.3.	Curves	1015
22.2.4.	Normal vectors	1016
22.2.5.	Curve modelling	1017
22.2.6.	Surface modelling	1022
22.2.7.	Solid modelling with blobs	1023
22.2.8.	Constructive solid geometry	1024
22.3.	Geometry processing and tessellation algorithms	1026
22.3.1.	Polygon and polyhedron	1026
22.3.2.	Vectorization of parametric curves	1027
22.3.3.	Tessellation of simple polygons	1027
22.3.4.	Tessellation of parametric surfaces	1029
22.3.5.	Subdivision curves and meshes	1031
22.3.6.	Tessellation of implicit surfaces	1033
22.4.	Containment algorithms	1035
22.4.1.	Point containment test	1035
22.4.2.	Polyhedron-polyhedron collision detection	1039
22.4.3.	Clipping algorithms	1040
22.5.	Translation, distortion, geometric transformations	1044
22.5.1.	Projective geometry and homogeneous coordinates	1045
22.5.2.	Homogeneous linear transformations	1049
22.6.	Rendering with ray tracing	1052
22.6.1.	Ray surface intersection calculation	1054
22.6.2.	Speeding up the intersection calculation	1056
22.7.	Incremental rendering	1070
22.7.1.	Camera transformation	1071
22.7.2.	Normalizing transformation	1073
22.7.3.	Perspective transformation	1074
22.7.4.	Clipping in homogeneous coordinates	1076
22.7.5.	Viewport transformation	1077
22.7.6.	Rasterization algorithms	1078
22.7.7.	Incremental visibility algorithms	1084
23.	Human-Computer Interaction	1093
23.1.	Multiple-choice systems	1093
23.1.1.	Examples of multiple-choice systems	1094
23.2.	Generating multiple candidate solutions	1097
23.2.1.	Generating candidate solutions with heuristics	1097
23.2.2.	Penalty method with exact algorithms	1100
23.2.3.	The linear programming - penalty method	1108
23.2.4.	Penalty method with heuristics	1112
23.3.	More algorithms for interactive problem solving	1113
23.3.1.	Anytime algorithms	1114
23.3.2.	Interactive evolution and generative design	1115
23.3.3.	Successive fixing	1115
23.3.4.	Interactive multicriteria decision making	1115

23.3.5. Miscellaneous	1116
Bibliography	1118
Index	1129
Name Index	1140

IV. COMPUTER NETWORKS

13. Distributed Algorithms

We define a distributed system as a collection of individual computing devices that can communicate with each other. This definition is very broad, it includes anything, from a VLSI chip, to a tightly coupled multiprocessor, to a local area cluster of workstations, to the Internet. Here we focus on more loosely coupled systems. In a distributed system as we view it, each processor has its semi-independent agenda, but for various reasons, such as sharing of resources, availability, and fault-tolerance, processors need to coordinate their actions.

Distributed systems are highly desirable, but it is notoriously difficult to construct efficient distributed algorithms that perform well in realistic system settings. These difficulties are not just of a more practical nature, they are also fundamental in nature. In particular, many of the difficulties are introduced by the three factors of: asynchrony, limited local knowledge, and failures. Asynchrony means that global time may not be available, and that both absolute and relative times at which events take place at individual computing devices can often not be known precisely. Moreover, each computing device can only be aware of the information it receives, it has therefore an inherently local view of the global status of the system. Finally, computing devices and network components may fail independently, so that some remain functional while others do not.

We will begin by describing the models used to analyse distributed systems in the message-passing model of computation. We present and analyze selected distributed algorithms based on these models. We include a discussion of fault-tolerance in distributed systems and consider several algorithms for reaching agreement in the messages-passing models for settings prone to failures. Given that global time is often unavailable in distributed systems, we present approaches for providing logical time that allows one to reason about causality and consistent states in distributed systems. Moving on to more advanced topics, we present a spectrum of broadcast services often considered in distributed systems and present algorithms implementing these services. We also present advanced algorithms for rumor gathering algorithms. Finally, we also consider the mutual exclusion problem in the shared-memory model of distributed computation.

13.1. Message passing systems and algorithms

We present our first model of distributed computation, for message passing systems without failures. We consider both synchronous and asynchronous systems and present selected algorithms for message passing systems with arbitrary network topology, and both synchronous and asynchronous settings.

13.1.1. Modeling message passing systems

In a message passing system, processors communicate by sending messages over communication channels, where each channel provides a bidirectional connection between two specific processors. We call the pattern of connections described by the channels, the *topology* of the system. This topology is represented by an undirected graph, where each node represents a processor, and an edge is present between two nodes if and only if there is a channel between the two processors represented by the nodes. The collection of channels is also called the *network*. An algorithm for such a message passing system with a specific topology consists of a local program for each processor in the system. This local program provides the ability to the processor to perform local computations, to send and receive messages from each of its neighbours in the given topology.

Each processor in the system is modeled as a possibly infinite state machine. A *configuration* is a vector $C = (q_0, \dots, q_{n-1})$ where each q_i is the state of a processor p_i . Activities that can take place in the system are modeled as *events* (or *actions*) that describe indivisible system operations. Examples of events include local computation events and delivery events where a processor receives a message. The behaviour of the system over time is modeled as an *execution*, a (finite or infinite) sequence of configurations (C_i) alternating with events (a_i): $C_0, a_1, C_1, a_2, C_2, \dots$. Executions must satisfy a variety of conditions that are used to represent the correctness properties, depending on the system being modeled. These conditions can be classified as either safety or liveness conditions. A *safety condition* for a system is a condition that must hold in every finite prefix of any execution of the system. Informally it states that nothing *bad* has happened yet. A *liveness condition* is a condition that must hold a certain (possibly infinite) number of times. Informally it states that eventually something *good* must happen. An important liveness condition is *fairness*, which requires that an (infinite) execution contains infinitely many actions by a processor, unless after some configuration no actions are enabled at that processor.

13.1.2. Asynchronous systems

We say that a system is *asynchronous* if there is no fixed upper bound on how long it takes for a message to be delivered or how much time elapses between consecutive steps of a processor. An obvious example of such an asynchronous system is the Internet. In an implementation of a distributed system there are often upper bounds on message delays and processor step times. But since these upper bounds are often very large and can change over time, it is often desirable to develop an algorithm

that is independent of any timing parameters, that is, an asynchronous algorithm.

In the asynchronous model we say that an execution is *admissible* if each processor has an infinite number of computation events, and every message sent is eventually delivered. The first of these requirements models the fact that processors do not fail. (It does not mean that a processor's local program contains an infinite loop. An algorithm can still terminate by having a transition function not change a processor's state after a certain point.)

We assume that each processor's set of states includes a subset of *terminated* states. Once a processor enters such a state it remains in it. The algorithm has *terminated* if all processors are in terminated states and no messages are in transit.

The *message complexity* of an algorithm in the asynchronous model is the maximum over all admissible executions of the algorithm, of the total number of (point-to-point) messages sent.

A *timed execution* is an execution that has a nonnegative real number associated with each event, the *time* at which the event occurs. To measure the *time complexity* of an asynchronous algorithm we first assume that the maximum message delay in any execution is one unit of time. Hence the *time complexity* is the maximum time until termination among all timed admissible executions in which every message delay is at most one. Intuitively this can be viewed as taking any execution of the algorithm and normalising it in such a way that the longest message delay becomes one unit of time.

13.1.3. Synchronous systems

In the synchronous model processors execute in lock-step. The execution is partitioned into rounds so that every processor can send a message to each neighbour, the messages are delivered, and every processor computes based on the messages just received. This model is very convenient for designing algorithms. Algorithms designed in this model can in many cases be automatically simulated to work in other, more realistic timing models.

In the synchronous model we say that an execution is admissible if it is infinite. From the round structure it follows then that every processor takes an infinite number of computation steps and that every message sent is eventually delivered. Hence in a synchronous system with no failures, once a (deterministic) algorithm has been fixed, the only relevant aspect determining an execution that can change is the initial configuration. On the other hand in an asynchronous system, there can be many different executions of the same algorithm, even with the same initial configuration and no failures, since here the interleaving of processor steps, and the message delays, are not fixed.

The notion of *terminated states* and the *termination* of the algorithm is defined in the same way as in the asynchronous model.

The *message complexity* of an algorithm in the synchronous model is the maximum over all admissible executions of the algorithm, of the total number of messages sent.

To measure time in a synchronous system we simply count the number of rounds until termination. Hence the *time complexity* of an algorithm in the synchronous

model is the maximum number of rounds in any admissible execution of the algorithm until the algorithm has terminated.

13.2. Basic algorithms

We begin with some simple examples of algorithms in the message passing model.

13.2.1. Broadcast

We start with a simple algorithm SPANNING-TREE-BROADCAST for the (single message) broadcast problem, assuming that a spanning tree of the network graph with n nodes (processors) is already given. Later, we will remove this assumption. A processor p_i wishes to send a message M to all other processors. The spanning tree rooted at p_i is maintained in a distributed fashion: Each processor has a distinguished channel that leads to its *parent* in the tree as well as a set of channels that lead to its *children* in the tree. The root p_i sends the message M on all channels leading to its children. When a processor receives the message on a channel from its parent, it sends M on all channels leading to its children.

SPANNING-TREE-BROADCAST

Initially M is in transit from p_i to all its children in the spanning tree.

Code for p_i :

```

1   upon receiving no message: // first computation event by  $p_i$ 
2   terminate
```

Code for p_j , $0 \leq j \leq n - 1$, $j \neq i$:

```

3   upon receiving  $M$  from parent:
4   send  $M$  to all children
5   terminate
```

The algorithm SPANNING-TREE-BROADCAST is correct whether the system is synchronous or asynchronous. Moreover, the message and time complexities are the same in both models.

Using simple inductive arguments we will first prove a lemma that shows that by the end of round t , the message M reaches all processors at distance t (or less) from p_r in the spanning tree.

Lemma 13.1 *In every admissible execution of the broadcast algorithm in the synchronous model, every processor at distance t from p_r in the spanning tree receives the message M in round t .*

Proof We proceed by induction on the distance t of a processor from p_r . First let $t = 1$. It follows from the algorithm that each child of p_r receives the message in round 1.

Assume that each processor at distance $t - 1$ received the message M in round

$t - 1$. We need to show that each processor p_t at distance t receives the message in round t . Let p_s be the parent of p_t in the spanning tree. Since p_s is at distance $t - 1$ from p_r , by the induction hypothesis, p_s received M in round $t - 1$. By the algorithm, p_t will hence receive M in round t . ■

By Lemma 13.1 the time complexity of the broadcast algorithm is d , where d is the depth of the spanning tree. Now since d is at most $n - 1$ (when the spanning tree is a chain) we have:

Theorem 13.2 *There is a synchronous broadcast algorithm for n processors with message complexity $n - 1$ and time complexity d , when a rooted spanning tree with depth d is known in advance.*

We now move to an asynchronous system and apply a similar analysis.

Lemma 13.3 *In every admissible execution of the broadcast algorithm in the asynchronous model, every processor at distance t from p_r in the spanning tree receives the message M by time t .*

Proof We proceed by induction on the distance t of a processor from p_r . First let $t = 1$. It follows from the algorithm that M is initially in transit to each processor p_i at distance 1 from p_r . By the definition of time complexity for the asynchronous model, p_i receives M by time 1.

Assume that each processor at distance $t - 1$ received the message M at time $t - 1$. We need to show that each processor p_t at distance t receives the message by time t . Let p_s be the parent of p_t in the spanning tree. Since p_s is at distance $t - 1$ from p_r , by the induction hypothesis, p_s sends M to p_t when it receives M at time $t - 1$. By the algorithm, p_t will hence receive M by time t . ■

We immediately obtain:

Theorem 13.4 *There is an asynchronous broadcast algorithm for n processors with message complexity $n - 1$ and time complexity d , when a rooted spanning tree with depth d is known in advance.*

13.2.2. Construction of a spanning tree

The asynchronous algorithm called FLOOD, discussed next, constructs a spanning tree rooted at a designated processor p_r . The algorithm is similar to the Depth First Search (DFS) algorithm. However, unlike DFS where there is just one processor with “global knowledge” about the graph, in the FLOOD algorithm, each processor has “local knowledge” about the graph, processors coordinate their work by exchanging messages, and processors and messages may get delayed arbitrarily. This makes the design and analysis of FLOOD algorithm challenging, because we need to show that the algorithm indeed constructs a spanning tree despite conspiratorial selection of these delays.

Algorithm description. Each processor has four local variables. The links adjacent to a processor are identified with distinct numbers starting from 1 and stored in a local variable called *neighbours*. We will say that the *spanning tree has been constructed*, when the variable *parent* stores the identifier of the link leading to the parent of the processor in the spanning tree, except that this variable is NONE for the designated processor p_r ; *children* is a set of identifiers of the links leading to the children processors in the tree; and *other* is a set of identifiers of all other links. So the knowledge about the spanning tree may be “distributed” across processors.

The code of each processor is composed of segments. There is a segment (lines 1–4) that describes how local variables of a processor are initialised. Recall that the local variables are initialised that way before time 0. The next three segments (lines 5–11, 12–15 and 16–19) describe the instructions that any processor executes in response to having received a message: $\langle adopt \rangle$, $\langle approved \rangle$ or $\langle rejected \rangle$. The last segment (lines 20–22) is only included in the code of processor p_r . This segment is executed only when the local variable *parent* of processor p_r is NIL. At some point of time, it may happen that more than one segment can be executed by a processor (e.g., because the processor received $\langle adopt \rangle$ messages from two processors). Then the processor executes the segments serially, one by one (segments of any given processor are never executed concurrently). However, instructions of different processor may be arbitrarily interleaved during an execution. Every message that can be processed is eventually processed and every segment that can be executed is eventually executed (fairness).

FLOOD

Code for any processor p_k , $1 \leq k \leq n$

```

1  initialisation
2    parent  $\leftarrow$  NIL
3    children  $\leftarrow$   $\emptyset$ 
4    other  $\leftarrow$   $\emptyset$ 

5  process message  $\langle adopt \rangle$  that has arrived on link  $j$ 
6    if parent = NIL
7      then parent  $\leftarrow$   $j$ 
8          send  $\langle approved \rangle$  to link  $j$ 
9          send  $\langle adopt \rangle$  to all links in neighbours  $\setminus \{j\}$ 
10   else send  $\langle rejected \rangle$  to link  $j$ 

11 process message  $\langle approved \rangle$  that has arrived on link  $j$ 
12   children  $\leftarrow$  children  $\cup \{j\}$ 
13   if children  $\cup$  other = neighbours  $\setminus \{parent\}$ 
14     then terminate

```

```

15 process message  $\langle rejected \rangle$  that has arrived on link  $j$ 
16    $other \leftarrow other \cup \{j\}$ 
17   if  $children \cup other = neighbours \setminus \{parent\}$ 
18     then terminate
    Extra code for the designated processor  $p_r$ 
19 if  $parent = \text{NIL}$ 
20   then  $parent \leftarrow \text{NONE}$ 
21     send  $\langle adopt \rangle$  to all links in  $neighbours$ 

```

Let us outline how the algorithm works. The designated processor sends an $\langle adopt \rangle$ message to all its neighbours, and assigns NONE to the $parent$ variable (NIL and NONE are two distinguished values, different from any natural number), so that it never again sends the message to any neighbour.

When a processor processes message $\langle adopt \rangle$ for the first time, the processor assigns to its own $parent$ variable the identifier of the link on which the message has arrived, responds with an $\langle approved \rangle$ message to that link, and forwards an $\langle adopt \rangle$ message to every other link. However, when a processor processes message $\langle adopt \rangle$ again, then the processor responds with a $\langle rejected \rangle$ message, because the $parent$ variable is no longer NIL.

When a processor processes message $\langle approved \rangle$, it adds the identifier of the link on which the message has arrived to the set $children$. It may turn out that the sets $children$ and $other$ combined form identifiers of all links adjacent to the processor except for the identifier stored in the $parent$ variable. In this case the processor enters a terminating state.

When a processor processes message $\langle rejected \rangle$, the identifier of the link is added to the set $other$. Again, when the union of $children$ and $other$ is large enough, the processor enters a terminating state.

Correctness proof. We now argue that FLOOD constructs a spanning tree. The key moments in the execution of the algorithm are when any processor assigns a value to its $parent$ variable. These assignments determine the “shape” of the spanning tree. The facts that any processor eventually executes an instruction, any message is eventually delivered, and any message is eventually processed, ensure that the knowledge about these assignments spreads to neighbours. Thus the algorithm is expanding a subtree of the graph, albeit the expansion may be slow. Eventually, a spanning tree is formed. Once a spanning tree has been constructed, eventually every processor will terminate, even though some processors may have terminated even before the spanning tree has been constructed.

Lemma 13.5 *For any $1 \leq k \leq n$, there is time t_k which is the first moment when there are exactly k processors whose parent variables are not NIL, and these processors and their parent variables form a tree rooted at p_r .*

Proof We prove the statement of the lemma by induction on k . For the base case, assume that $k = 1$. Observe that processor p_r eventually assigns NONE to its $parent$

variable. Let t_1 be the moment when this assignment happens. At that time, the *parent* variable of any processor other than p_r is still NIL, because no $\langle adopt \rangle$ messages have been sent so far. Processor p_r and its *parent* variable form a tree with a single node and not arcs. Hence they form a rooted tree. Thus the inductive hypothesis holds for $k = 1$.

For the inductive step, suppose that $1 \leq k < n$ and that the inductive hypothesis holds for k . Consider the time t_k which is the first moment when there are exactly k processors whose *parent* variables are not *nil*. Because $k < n$, there is a non-tree processor. But the graph G is connected, so there is a non-tree processor adjacent to the tree. (For any subset T of processors, a processor p_i is *adjacent* to T if and only if there an edge in the graph G from p_i to a processor in T .) Recall that by definition, *parent* variable of such processor is NIL. By the inductive hypothesis, the k processors must have executed line 7 of their code, and so each either has already sent or will eventually send $\langle adopt \rangle$ message to all its neighbours on links other than the *parent* link. So the non-tree processors adjacent to the tree have already received or will eventually receive $\langle adopt \rangle$ messages. Eventually, each of these adjacent processors will, therefore, assign a value other than NIL to its *parent* variable. Let $t_{k+1} > t_k$ be the first moment when any processor performs such assignment, and let us denote this processor by p_i . This cannot be a tree processor, because such processor never again assigns any value to its *parent* variable. Could p_i be a non-tree processor that is not adjacent to the tree? It could not, because such processor does not have a direct link to a tree processor, so it cannot receive $\langle adopt \rangle$ directly from the tree, and so this would mean that at some time t' between t_k and t_{k+1} some other non-tree processor p_j must have sent $\langle adopt \rangle$ message to p_i , and so p_j would have to assign a value other than NIL to its *parent* variable some time after t_k but before t_{k+1} , contradicting the fact the t_{k+1} is the first such moment. Consequently, p_i is a non-tree processor adjacent to the tree, such that, at time t_{k+1} , p_i assigns to its *parent* variable the index of a link leading to a tree processor. Therefore, time t_{k+1} is the first moment when there are exactly $k + 1$ processors whose *parent* variables are not NIL, and, at that time, these processors and their *parent* variables form a tree rooted at p_r . This completes the inductive step, and the proof of the lemma. ■

Theorem 13.6 *Eventually each processor terminates, and when every processor has terminated, the subgraph induced by the parent variables forms a spanning tree rooted at p_r .*

Proof By Lemma 13.5, we know that there is a moment t_n which is the first moment when all processors and their *parent* variables form a spanning tree.

Is it possible that every processor has terminated before time t_n ? By inspecting the code, we see that a processor terminates only after it has received $\langle rejected \rangle$ or $\langle approved \rangle$ messages from all its neighbours other than the one to which *parent* link leads. A processor receives such messages only in response to $\langle adopt \rangle$ messages that the processor sends. At time t_n , there is a processor that still has not even sent $\langle adopt \rangle$ messages. Hence, not every processor has terminated by time t_n .

Will every processor eventually terminate? We notice that by time t_n , each processor either has already sent or will eventually send $\langle adopt \rangle$ message to all

its neighbours other than the one to which *parent* link leads. Whenever a processor receives $\langle adopt \rangle$ message, the processor responds with $\langle rejected \rangle$ or $\langle approved \rangle$, even if the processor has already terminated. Hence, eventually, each processor will receive either $\langle rejected \rangle$ or $\langle approved \rangle$ message on each link to which the processor has sent $\langle adopt \rangle$ message. Thus, eventually, each processor terminates. ■

We note that the fact that a processor has terminated does not mean that a spanning tree has already been constructed. In fact, it may happen that processors in a different part of the network have not even received any message, let alone terminated.

Theorem 13.7 *Message complexity of FLOOD is $O(e)$, where e is the number of edges in the graph G .*

The proof of this theorem is left as Problem 13-1.

Exercises

13.2-1 It may happen that a processor has terminated even though a processor has not even received any message. Show a simple network and how to delay message delivery and processor computation to demonstrate that this can indeed happen.

13.2-2 It may happen that a processor has terminated but may still respond to a message. Show a simple network and how to delay message delivery and processor computation to demonstrate that this can indeed happen.

13.3. Ring algorithms

One often needs to coordinate the activities of processors in a distributed system. This can frequently be simplified when there is a single processor that acts as a coordinator. Initially, the system may not have any coordinator, or an existing coordinator may fail and so another may need to be elected. This creates the problem where processors must elect exactly one among them, a *leader*. In this section we study the problem for special types of networks—rings. We will develop an asynchronous algorithm for the problem. As we shall demonstrate, the algorithm has asymptotically optimal message complexity. In the current section, we will see a distributed analogue of the well-known divide-and-conquer technique often used in sequential algorithms to keep their time complexity low. The technique used in distributed systems helps reduce the message complexity.

13.3.1. The leader election problem

The leader election problem is to elect exactly leader among a set of processors. Formally each processor has a local variable *leader* initially equal to NIL. An algorithm is said to *solve the leader election problem* if it satisfies the following conditions:

1. in any execution, exactly one processor eventually assigns TRUE to its *leader* variable, all other processors eventually assign FALSE to their *leader* variables,