Contents

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 $C = (q_0, \ldots, q_{n-1})$ 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): $C_0, a_1, C_1, a_2, C_2, \ldots$. 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 processors 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 ≤ j ≤ n − 1, j ≠ 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: <*adopt*>, <*approved*> or <*rejected*>. 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 <*adopt*> 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 ≤ k ≤ n
 1   initialisation
 2       parent ← NIL
 3       children ← ∅
 4       other ← ∅

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

11   process message <approved> that has arrived on link j
12       children ← children ∪ {j}
13       if children ∪ other = neighbours \ {parent}
14         then terminate
```

```
15  process message <rejected> that has arrived on link j
16      other ← other ∪ {j}
17      if children ∪ other = neighbours \ {parent}
18          then terminate
    Extra code for the designated processor p_r
19  if parent = NIL
20      then parent ← NONE
21           send <adopt> to all links in neighbours
```

Let us outline how the algorithm works. The designated processor sends an $<$*adopt*$>$ 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 $<$*adopt*$>$ 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 $<$*approved*$>$ message to that link, and forwards an $<$*adopt*$>$ message to every other link. However, when a processor processes message $<$*adopt*$>$ again, then the processor responds with a $<$*rejected*$>$ message, because the *parent* variable is no longer NIL.

When a processor processes message $<$*approved*$>$, 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 $<$*rejected*$>$, 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 <*adopt*> 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 <*adopt*> 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 <*adopt*> 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 <*adopt*> 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 <*adopt*> 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 <*rejected*> or <*approved*> messages from all its neighbours other than the one to which *parent* link leads. A processor receives such messages only in response to <*adopt*> messages that the processor sends. At time t_n, there is a processor that still has not even sent <*adopt*> 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 <*adopt*> message to all

its neighbours other than the one to which *parent* link leads. Whenever a processor receives <*adopt*> message, the processor responds with <*rejected*> or <*approved*>, even if the processor has already terminated. Hence, eventually, each processor will receive either <*rejected*> or <*approved*> message on each link to which the processor has sent <*adopt*> 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, and

2. in any execution, once a processor has assigned a value to its *leader* variable, the variable remains unchanged.

Ring model. We study the leader election problem on a special type of network—the ring. Formally, the graph G that models a distributed system consists of n nodes that form a simple cycle; no other edges exist in the graph. The two links adjacent to a processor are labeled ***CW*** (**C**lock-**W**ise) and ***CCW*** (**C**ounter **C**lock-**W**ise). Processors agree on the orientation of the ring i.e., if a message is passed on in CW direction n times, then it visits all n processors and comes back to the one that initially sent the message; same for CCW direction. Each processor has a unique ***identifier*** that is a natural number, i.e., the identifier of each processor is different from the identifier of any other processor; the identifiers do not have to be consecutive numbers $1, \ldots, n$. Initially, no processor knows the identifier of any other processor. Also processors do not know the size n of the ring.

13.3.2. The leader election algorithm

BULLY elects a leader among asynchronous processors $p_1, \ldots, p_n$. Identifiers of processors are used by the algorithm in a crucial way. Briefly speaking, each processor tries to become the leader, the processor that has the largest identifier among all processors blocks the attempts of other processors, declares itself to be the leader, and forces others to declare themselves not to be leaders.

Let us begin with a simpler version of the algorithm to exemplify some of the ideas of the algorithm. Suppose that each processor sends a message around the ring containing the identifier of the processor. Any processor passes on such message *only* if the identifier that the message carries is strictly larger than the identifier of the processor. Thus the message sent by the processor that has the largest identifier among the processors of the ring, will always be passed on, and so it will eventually travel around the ring and come back to the processor that initially sent it. The processor can detect that such message has come back, because no other processor sends a message with this identifier (identifiers are distinct). We observe that, no other message will make it all around the ring, because the processor with the largest identifier will not pass it on. We could say that the processor with the largest identifier "swallows" these messages that carry smaller identifiers. Then the processor becomes the leader and sends a special message around the ring forcing all others to decide not to be leaders. The algorithm has $\Theta(n^2)$ message complexity, because each processor induces at most n messages, and the leader induces n extra messages; and one can assign identifiers to processors and delay processors and messages in such a way that the messages sent by a constant fraction of n processors are passed on around the ring for a constant fraction of n hops. The algorithm can be improved so as to reduce message complexity to $O(n \lg n)$, and such improved algorithm will be presented in the remainder of the section.

The key idea of the BULLY algorithm is to make sure that not too many mes-

sages travel far, which will ensure $O(n \lg n)$ message complexity. Specifically, the activity of any processor is divided into phases. At the beginning of a phase, a processor sends "probe" messages in both directions: CW and CCW. These messages carry the identifier of the sender and a certain "time-to-live" value that limits the number of hops that each message can make. The probe message may be passed on by a processor provided that the identifier carried by the message is larger than the identifier of the processor. When the message reaches the limit, and has not been swallowed, then it is "bounced back". Hence when the initial sender receives two bounced back messages, each from each direction, then the processor is certain that there is no processor with larger identifier up until the limit in CW nor CCW directions, because otherwise such processor would swallow a probe message. Only then does the processor enter the next phase through sending probe messages again, this time with the time-to-live value increased by a factor, in an attempt to find if there is no processor with a larger identifier in twice as large neighbourhood. As a result, a probe message that the processor sends will make many hops only when there is no processor with larger identifier in a large neighbourhood of the processor. Therefore, fewer and fewer processors send messages that can travel longer and longer distances. Consequently, as we will soon argue in detail, message complexity of the algorithm is $O(n \lg n)$.

We detail the BULLY algorithm. Each processor has five local variables. The variable *id* stores the unique identifier of the processor. The variable *leader* stores TRUE when the processor decides to be the leader, and FALSE when it decides not to be the leader. The remaining three variables are used for bookkeeping: *asleep* determines if the processor has ever sent a $<probe,id,0,0>$ message that carries the identifier *id* of the processor. Any processor may send $<probe,id,phase,2^{phase-1}>$ message in both directions (CW and CCW) for different values of *phase*. Each time a message is sent, a $<reply, id, phase>$ message may be sent back to the processor. The variables $CWreplied$ and $CCWreplied$ are used to remember whether the replies have already been processed the processor.

The code of each processor is composed of five segments. The first segment (lines 1–5) initialises the local variables of the processor. The second segment (lines 6–8) can only be executed when the local variable *asleep* is TRUE. The remaining three segments (lines 9–17, 1–26, and 27–31) describe the actions that the processor takes when it processes each of the three types of messages: $<probe, ids, phase, ttl>$, $<reply, ids, phase>$ and $<terminate>$ respectively. The messages carry parameters *ids*, *phase* and *ttl* that are natural numbers.

We now describe how the algorithm works. Recall that we assume that the local variables of each processor have been initialised before time 0 of the global clock. Each processor eventually sends a $<probe,id,0,0>$ message carrying the identifier *id* of the processor. At that time we say that the processor *enters* phase number zero. In general, when a processor sends a message $<probe,id,phase,2^{phase}-1>$, we say that the processor *enters* phase number *phase*. Message $<probe,id,0,0>$ is never sent again because FALSE is assigned to *asleep* in line 7. It may happen that by the time this message is sent, some other messages have already been processed by the processor.

When a processor processes message $<probe, ids, phase, ttl>$ that has arrived on

link CW (the link leading in the clock-wise direction), then the actions depend on the relationship between the parameter *ids* and the identifier *id* of the processor. If *ids* is smaller than *id*, then the processor does nothing else (the processor swallows the message). If *ids* is equal to *id* and processor has not yet decided, then, as we shall see, the probe message that the processor sent has circulated around the entire ring. Then the processor sends a <*terminate*> message, decides to be the leader, and terminates (the processor may still process messages after termination). If *ids* is larger than *id*, then actions of the processor depend on the value of the parameter *ttl* (time-to-live). When the value is strictly larger than zero, then the processor passes on the probe message with *ttl* decreased by one. If, however, the value of *ttl* is already zero, then the processor sends back (in the CW direction) a reply message. Symmetric actions are executed when the <*probe*, *ids*, *phase*, *ttl*> message has arrived on link CCW, in the sense that the directions of sending messages are respectively reversed – see the code for details.

Bully

```
    Code for any processor p_k, 1 ≤ k ≤ n
 1  initialisation
 2      asleep ← TRUE
 3      CWreplied ← FALSE
 4      CCWreplied ← FALSE
 5      leader ← NIL

 6  if asleep
 7    then asleep ←FALSE
 8         send <probe,id,0,0> to links CW and CCW

 9  process message <probe, ids, phase, ttl> that has arrived
                    on link CW (resp. CCW)
10                  if id = ids and leader = NIL
11                    then send <terminate> to link CCW
12                         leader ← TRUE
13                         terminate
14                  if ids > id and ttl > 0
15                    then send < probe,ids,phase,ttl − 1 >
                      to link CCW (resp. CW)
16                  if ids > id and ttl = 0
17                    then send <reply,ids,phase> to link CW (resp. CCW)
```

```
18 process message <reply,ids,phase> that has arrived on link CW (resp. CCW)
19     if id ≠ ids
20         then send <reply,ids,phase> to link CCW (resp. CW)
21         else  CWreplied ← TRUE (resp. CCWreplied)
22     if CWreplied and CCWreplied
23         then CWreplied ← FALSE
24              CCWreplied ← FALSE
25              send <probe,id,phase+1,2^(phase+1) − 1>
                to links CW and CCW

26 process message <terminate> that has arrived on link CW
27     if leader NIL
28         then send <terminate> to link CCW
29              leader ← FALSE
30              terminate
```

When a processor processes message $<reply, ids, phase>$ that has arrived on link CW, then the processor first checks if *ids* is different from the identifier *id* of the processor. If so, the processor merely passes on the message. However, if $ids = id$, then the processor records the fact that a reply has been received from direction CW, by assigning TRUE to *CWreplied*. Next the processor checks if both *CWreplied* and *CCWreplied* variables are *true*. If so, the processor has received replies from both directions. Then the processor assigns *false* to both variables. Next the processor sends a probe message. This message carries the identifier *id* of the processor, the next phase number $phase + 1$, and an increased time-to-live parameter $2^{phase+1} - 1$. Symmetric actions are executed when <*reply,ids,phase*> has arrived on link CCW.

The last type of message that a processor can process is <*terminate*>. The processor checks if it has already decided to be or not to be the leader. When no decision has been made so far, the processor passes on the <*terminate*> message and decides not to be the leader. This message eventually reaches a processor that has already decided, and then the message is no longer passed on.

13.3.3. Analysis of the leader election algorithm

We begin the analysis by showing that the algorithm BULLY solves the leader election problem.

Theorem 13.8 BULLY *solves the leader election problem on any ring with asynchronous processors.*

Proof We need to show that the two conditions listed at the beginning of the section are satisfied. The key idea that simplifies the argument is to focus on one processor. Consider the processor p_i with maximum *id* among all processors in the ring. This processor eventually executes lines 6–8. Then the processor sends <*probe,id*,0,0> messages in CW and CCW directions. Note that whenever the processor sends $<probe,id,phase,2^{phase} - 1>$ messages, each such message is always passed on by

other processors, until the *ttl* parameter of the message drops down to zero, or the message travels around the entire ring and arrives at p_i. If the message never arrives at p_i, then a processor eventually receives the probe message with *ttl* equal to zero, and the processor sends a response back to p_i. Then, eventually p_i receives messages <*reply,id,phase*> from each directions, and enters phase number $phase + 1$ by sending probe messages <*probe,id,phase*+1,$2^{phase+1} - 1$> in both directions. These messages carry a larger time-to-live value compared to the value from the previous phase number *phase*. Since the ring is finite, eventually *ttl* becomes so large that processor p_i receives a probe message that carries the identifier of p_i. Note that p_i will eventually receive two such messages. The first time when p_i processes such message, the processor sends a <*terminate*> message and terminates as the leader. The second time when p_i processes such message, lines 11–13 are not executed, because variable *leader* is no longer NIL. Note that no other processor p_j can execute lines 11–13, because a probe message originated at p_j cannot travel around the entire ring, since p_i is on the way, and p_i would swallow the message; and since identifiers are distinct, no other processor sends a probe message that carries the identifier of processor p_j. Thus no processor other than p_i can assign TRUE to its *leader* variable. Any processor other than p_i will receive the <*terminate*> message, assign FALSE to its *leader* variable, and pass on the message. Finally, the <*terminate*> message will arrive at p_i, and p_i will not pass it anymore. The argument presented thus far ensures that eventually exactly one processor assigns TRUE to its *leader* variable, all other processors assign FALSE to their *leader* variables, and once a processor has assigned a value to its *leader* variable, the variable remains unchanged. ■

Our next task is to give an upper bound on the number of messages sent by the algorithm. The subsequent lemma shows that the number of processors that can enter a phase decays exponentially as the phase number increases.

Lemma 13.9 *Given a ring of size n, the number k of processors that enter phase number $i \geq 0$ is at most $n/2^{i-1}$.*

Proof There are exactly n processors that enter phase number $i = 0$, because each processor eventually sends <*probe,id*,0,0> message. The bound stated in the lemma says that the number of processors that enter phase 0 is at most $2n$, so the bound evidently holds for $i = 0$. Let us consider any of the remaining cases i.e., let us assume that $i \geq 1$. Suppose that a processor p_j enters phase number i, and so by definition it sends message <*probe,id,i*,$2^i - 1$>. In order for a processor to send such message, each of the two probe messages <*probe,id,i*-1,$2^{i-1} - 1$> that the processor sent in the previous phase in both directions must have made 2^{i-1} hops always arriving at a processor with strictly lower identifier than the identifier of p_j (because otherwise, if a probe message arrives at a processor with strictly larger or the same identifier, than the message is swallowed, and so a reply message is not generated, and consequently p_j cannot enter phase number i). As a result, if a processor enters phase number i, then there is no other processor 2^{i-1} hops away in both directions that can ever enter the phase. Suppose that there are $k \geq 1$ processors that enter phase i. We can associate with each such processor p_j, the 2^{i-1} consecutive processors that follow p_j in the CW direction. This association assigns 2^{i-1} distinct processors to each of

the k processors. So there must be at least $k + k \cdot 2^{i-1}$ distinct processor in the ring. Hence $k(1 + 2^{i-1}) \leq n$, and so we can weaken this bound by dropping 1, and conclude that $k \cdot 2^{i-1} \leq n$, as desired. ∎

Theorem 13.10 *The algorithm* BULLY *has $O(n \lg n)$ message complexity, where n is the size of the ring.*

Proof Note that any processor in phase i, sends messages that are intended to travel 2^i away and back in each direction (CW and CCW). This contributes at most $4 \cdot 2^i$ messages per processor that enters phase number i. The contribution may be smaller than $4 \cdot 2^i$ if a probe message gets swallowed on the way away from the processor. Lemma 13.9 provides an upper bound on the number of processors that enter phase number k. What is the highest phase that a processor can ever enter? The number k of processors that can be in phase i is at most $n/2^{i-1}$. So when $n/2^{i-1} < 1$, then there can be no processor that ever enters phase i. Thus no processor can enter any phase beyond phase number $h = 1 + \lceil \log_2 n \rceil$, because $n < 2^{(h+1)-1}$. Finally, a single processor sends one termination message that travels around the ring once. So for the total number of messages sent by the algorithm we get the

$$n + \sum_{i=0}^{1+\lceil \log_2 n \rceil} \left(n/2^{i-1} \cdot 4 \cdot 2^i \right) = n + \sum_{i=0}^{1+\lceil \log_2 n \rceil} 8n = O(n \lg n)$$

upper bound. ∎

Burns furthermore showed that the asynchronous leader election algorithm is asymptotically optimal: Any uniform algorithm solving the leader election problem in an asynchronous ring must send the number of messages at least proportional to $n \lg n$.

Theorem 13.11 *Any uniform algorithm for electing a leader in an asynchronous ring sends $\Omega(n \lg n)$ messages.*

The proof, for any algorithm, is based on constructing certain executions of the algorithm on rings of size $n/2$. Then two rings of size $n/2$ are pasted together in such a way that the constructed executions on the smaller rings are combined, and $\Theta(n)$ additional messages are received. This construction strategy yields the desired logarithmic multiplicative overhead.

Exercises

13.3-1 Show that the simplified BULLY algorithm has $\Omega(n^2)$ message complexity, by appropriately assigning identifiers to processors on a ring of size n, and by determining how to delay processors and messages.

13.3-2 Show that the algorithm BULLY has $\Omega(n \lg n)$ message complexity.

13.4. Fault-tolerant consensus

The algorithms presented so far are based on the assumption that the system on which they run is reliable. Here we present selected algorithms for unreliable distributed systems, where the active (or correct) processors need to coordinate their activities based on common decisions.

It is inherently difficult for processors to reach agreement in a distributed setting prone to failures. Consider the deceptively simple problem of two failure-free processors attempting to agree on a common bit using a communication medium where messages may be lost. This problem is known as the ***two generals problem.*** Here two generals must coordinate an attack using couriers that may be destroyed by the enemy. It turns out that it is not possible to solve this problem using a finite number of messages. We prove this fact by contradiction. Assume that there is a protocol used by processors A and B involving a finite number of messages. Let us consider such a protocol that uses the smallest number of messages, say k messages. Assume without loss of generality that the last k^{th} message is sent from A to B. Since this final message is not acknowledged by B, A must determine the decision value whether or not B receives this message. Since the message may be lost, B must determine the decision value without receiving this final message. But now both A and B decide on a common value without needing the k^{th} message. In other words, there is a protocol that uses only $k-1$ messages for the problem. But this contradicts the assumption that k is the smallest number of messages needed to solve the problem.

In the rest of this section we consider agreement problems where the communication medium is reliable, but where the processors are subject to two types of failures: ***crash failures,*** where a processor stops and does not perform any further actions, and ***Byzantine failures,*** where a processor may exhibit arbitrary, or even malicious, behaviour as the result of the failure.

The algorithms presented deal with the so called ***consensus problem,*** first introduced by Lamport, Pease, and Shostak. The consensus problem is a fundamental coordination problem that requires processors to agree on a common output, based on their possibly conflicting inputs.

13.4.1. The consensus problem

We consider a system in which each processor p_i has a special state component x_i, called the ***input*** and y_i, called the ***output*** (also called the ***decision***). The variable x_i initially holds a value from some well ordered set of possible inputs and y_i is undefined. Once an assignment to y_i has been made, it is irreversible. Any solution to the consensus problem must guarantee:

- **Termination:** In every admissible execution, y_i is eventually assigned a value, for every nonfaulty processor p_i.

- **Agreement:** In every execution, if y_i and y_j are assigned, then $y_i = y_j$, for all nonfaulty processors p_i and p_j. That is nonfaulty processors do not decide on conflicting values.

- **Validity:** In every execution, if for some value v, $x_i = v$ for all processors p_i, and if y_i is assigned for some nonfaulty processor p_i, then $y_i = v$. That is, if all processors have the same input value, then any value decided upon must be that common input.

Note that in the case of crash failures this validity condition is equivalent to requiring that every nonfaulty decision value is the input of some processor. Once a processor crashes it is of no interest to the algorithm, and no requirements are put on its decision.

We begin by presenting a simple algorithm for consensus in a synchronous message passing system with crash failures.

13.4.2. Consensus with crash failures

Since the system is synchronous, an execution of the system consists of a series of rounds. Each round consists of the delivery of all messages, followed by one computation event for every processor. The set of faulty processors can be different in different executions, that is, it is not known in advance. Let F be a subset of at most f processors, the faulty processors. Each round contains exactly one computation event for the processors not in F and at most one computation event for every processor in F. Moreover, if a processor in F does not have a computation event in some round, it does not have such an event in any further round. In the last round in which a faulty processor has a computation event, an arbitrary subset of its outgoing messages are delivered.

Consensus-with-Crash-Failures

```
   Code for processor p_i, 0 ≤ i ≤ n − 1.
   Initially V = {x}
   round k, 1 ≤ k ≤ f + 1
1  send {v ∈ V : p_i has not already sent v} to all processors
2  receive S_j from p_j, 0 ≤ j ≤ n − 1, j ≠ i
3  V ← V ∪ ⋃_{j=0}^{n−1} S_j
4  if k = f + 1
5     then y ← min(V)
```

In the previous algorithm, which is based on an algorithm by Dolev and Strong, each processor maintains a set of the values it knows to exist in the system. Initially, the set contains only its own input. In later rounds the processor updates its set by joining it with the sets received from other processors. It then broadcasts any new additions to the set of all processors. This continues for $f + 1$ rounds, where f is the maximum number of processors that can fail. At this point, the processor decides on the smallest value in its set of values.

To prove the correctness of this algorithm we first notice that the algorithm requires exactly $f + 1$ rounds. This implies termination. Moreover the validity con-

dition is clearly satisfied since the decision value is the input of some processor. It remains to show that the agreement condition holds. We prove the following lemma:

Lemma 13.12 *In every execution at the end of round $f+1$, $V_i = V_j$, for every two nonfaulty processors p_i and p_j.*

Proof We prove the claim by showing that if $x \in V_i$ at the end of round $f+1$ then $x \in V_j$ at the end of round $f+1$.

Let r be the first round in which x is added to V_i for any nonfaulty processor p_i. If x is initially in V_i let $r = 0$. If $r \leq f$ then, in round $r+1 \leq f+1$ p_i sends x to each p_j, causing p_j to add x to V_j, if not already present.

Otherwise, suppose $r = f+1$ and let p_j be a nonfaulty processor that receives x for the first time in round $f+1$. Then there must be a chain of $f+1$ processors $p_{i_1}, \ldots p_{i_{f+1}}$ that transfers the value x to p_j. Hence p_{i_1} sends x to p_{i_2} in round one etc. until $p_{i_{f+1}}$ sends x to p_j in round $f+1$. But then $p_{i_1}, \ldots, p_{i_{f+1}}$ is a chain of $f+1$ processors. Hence at least one of them, say p_{i_k} must be nonfaulty. Hence p_{i_k} adds x to its set in round $k-1 < r$, contradicting the minimality of r. ■

This lemma together with the before mentioned observations hence implies the following theorem.

Theorem 13.13 *The previous consensus algorithm solves the consensus problem in the presence of f crash failures in a message passing system in $f+1$ rounds.*

The following theorem was first proved by Fischer and Lynch for Byzantine failures. Dolev and Strong later extended it to crash failures. The theorem shows that the previous algorithm, assuming the given model, is optimal.

Theorem 13.14 *There is no algorithm which solves the consensus problem in less than $f+1$ rounds in the presence of f crash failures, if $n \geq f+2$.*

What if failures are not benign? That is can the consensus problem be solved in the presence of *Byzantine* failures? And if so, how?

13.4.3. Consensus with Byzantine failures

In a computation step of a faulty processor in the Byzantine model, the new state of the processor and the message sent are completely unconstrained. As in the reliable case, every processor takes a computation step in every round and every message sent is delivered in that round. Hence a faulty processor can behave arbitrarily and even maliciously. For example, it could send different messages to different processors. It can even appear that the faulty processors coordinate with each other. A faulty processor can also mimic the behaviour of a crashed processor by failing to send any messages from some point on.

In this case, the definition of the consensus problem is the same as in the message passing model with crash failures. The validity condition in this model, however, is not equivalent with requiring that every nonfaulty decision value is the input of some processor. Like in the crash case, no conditions are put on the output of faulty processors.

13.4.4. Lower bound on the ratio of faulty processors

Pease, Shostak and Lamport first proved the following theorem.

Theorem 13.15 *In a system with n processors and f Byzantine processors, there is no algorithm which solves the consensus problem if $n \leq 3f$.*

13.4.5. A polynomial algorithm

The following algorithm uses messages of constant size, takes $2(f+1)$ rounds, and assumes that $n > 4f$. It was presented by Berman and Garay.

This consensus algorithm for Byzantine failures contains $f+1$ phases, each taking two rounds. Each processor has a preferred decision for each phase, initially its input value. At the first round of each phase, processors send their preferences to each other. Let v_i^k be the majority value in the set of values received by processor p_i at the end of the first round of phase k. If no majority exists, a default value $v_\perp$ is used. In the second round of the phase processor p_k, called the *king* of the phase, sends its majority value v_k^k to all processors. If p_i receives more than $n/2+f$ copies of v_i^k (in the first round of the phase) then it sets its preference for the next phase to be v_i^k; otherwise it sets its preference to the phase kings preference, v_k^k received in the second round of the phase. After $f+1$ phases, the processor decides on its preference. Each processor maintains a local array *pref* with n entries.

We prove correctness using the following lemmas. Termination is immediate. We next note the persistence of agreement:

Lemma 13.16 *If all nonfaulty processors prefer v at the beginning of phase k, then they all prefer v at the end of phase k, for all k, $1 \leq k \leq f+1$.*

Proof Since all nonfaulty processors prefer v at the beginning of phase k, they all receive at least $n-f$ copies of v (including their own) in the first round of phase k. Since $n > 4f$, $n - f > n/2 + f$, implying that all nonfaulty processors will prefer v at the end of phase k. ∎

CONSENSUS-WITH-BYZANTINE-FAILURES

```
    Code for processor p_i, 0 ≤ i ≤ n − 1.
    Initially pref[j] = v_⊥, for any j ≠ i
    round 2k − 1, 1 ≤ k ≤ f + 1
1   send ⟨pref[i]⟩ to all processors
2   receive ⟨v_j⟩ from p_j and assign to pref[j], for all 0 ≤ j ≤ n − 1, j ≠ i
3   let maj be the majority value of pref[0],...,pref[n − 1](v_⊥ if none)
4   let mult be the multiplicity of maj
```

```
    round 2k, 1 ≤ k ≤ f + 1
5   if i = k
6      then send ⟨maj⟩ to all processors
7   receive ⟨king-maj⟩ from p_k (v_⊥ if none)
8   if mult > n/2 + f
9      then pref[i] ← maj
10     else  pref[i] ← king − maj
11  if k = f + 1
12     then y ←pref[i]
```

This implies the validity condition: If they all start with the same input v they will continue to prefer v and finally decide on v in phase $f+1$. Agreement is achieved by the king breaking ties. Since each phase has a different king and there are $f+1$ phases, at least one round has a nonfaulty king.

Lemma 13.17 *Let g be a phase whose king p_g is nonfaulty. Then all nonfaulty processors finish phase g with the same preference.*

Proof Suppose all nonfaulty processors use the majority value received from the king for their preference. Since the king is nonfaulty, it sends the same message and hence all the nonfaulty preferences are the same.

Suppose a nonfaulty processor p_i uses its own majority value v for its preference. Thus p_i receives more than $n/2 + f$ messages for v in the first round of phase g. Hence every processor, including p_g receives more than $n/2$ messages for v in the first round of phase g and sets its majority value to v. Hence every nonfaulty processor has v for its preference. ∎

Hence at phase $g+1$ all processors have the same preference and by Lemma 13.16 they will decide on the same value at the end of the algorithm. Hence the algorithm has the agreement property and solves consensus.

Theorem 13.18 *There exists an algorithm for n processors which solves the consensus problem in the presence of f Byzantine failures within $2(f+1)$ rounds using constant size messages, if $n > 4f$.*

13.4.6. Impossibility in asynchronous systems

As shown before, the consensus problem can be solved in synchronous systems in the presence of both crash (benign) and Byzantine (severe) failures. What about asynchronous systems? Under the assumption that the communication system is completely reliable, and the only possible failures are caused by unreliable processors, it can be shown that if the system is completely asynchronous then there is no consensus algorithm even in the presence of only a single processor failure. The result holds even if the processors only fail by crashing. The impossibility proof relies heavily on the system being asynchronous. This result was first shown in a

breakthrough paper by Fischer, Lynch and Paterson. It is one of the most influential results in distributed computing.

The impossibility holds for both shared memory systems if only read/write registers are used, and for message passing systems. The proof first shows it for shared memory systems. The result for message passing systems can then be obtained through simulation.

Theorem 13.19 *There is no consensus algorithm for a read/write asynchronous shared memory system that can tolerate even a single crash failure.*

And through simulation the following assertion can be shown.

Theorem 13.20 *There is no algorithm for solving the consensus problem in an asynchronous message passing system with n processors, one of which may fail by crashing.*

Note that these results do not mean that consensus can never be solved in asynchronous systems. Rather the results mean that there are no algorithms that guarantee termination, agreement, and validity, in all executions. It is reasonable to assume that agreement and validity are essential, that is, if a consensus algorithm terminates, then agreement and validity are guaranteed. In fact there are efficient and useful algorithms for the consensus problem that are not guaranteed to terminate in all executions. In practice this is often sufficient because the special conditions that cause non-termination may be quite rare. Additionally, since in many real systems one can make some timing assumption, it may not be necessary to provide a solution for asynchronous consensus.

Exercises

13.4-1 Prove the correctness of algorithm Consensus-Crash.

13.4-2 Prove the correctness of the consensus algorithm in the presence of Byzantine failures.

13.4-3 Prove Theorem 13.20.

13.5. Logical time, causality, and consistent state

In a distributed system it is often useful to compute a global state that consists of the states of all processors. Having access to the global can allows us to reason about the system properties that depend on all processors, for example to be able to detect a deadlock. One may attempt to compute global state by stopping all processors, and then gathering their states to a central location. Such a method is will-suited for many distributed systems that must continue computation at all times. This section discusses how one can compute global state that is quite intuitive, yet consistent, in a precise sense. We first discuss a distributed algorithm that imposes a global order on instructions of processors. This algorithm creates the illusion of a global clock available to processors. Then we introduce the notion of one instruction causally affecting other instruction, and an algorithm for computing which instruction affects which. The notion turns out to be very useful in defining a consistent global state of

distributed system. We close the section with distributed algorithms that compute a consistent global state of distributed system.

13.5.1. Logical time

The design of distributed algorithms is easier when processors have access to (Newtonian) global clock, because then each event that occurs in the distributed system can be labeled with the reading of the clock, processors agree on the ordering of any events, and this consensus can be used by algorithms to make decisions. However, construction of a global clock is difficult. There exist algorithms that approximate the ideal global clock by periodically synchronising drifting local hardware clocks. However, it is possible to totally order events without using hardware clocks. This idea is called the ***logical clock.***

Recall that an execution is an interleaving of instructions of the n programs. Each instruction can be either a computational step of a processor, or sending a message, or receiving a message. Any instruction is performed at a distinct point of global time. However, the reading of the global clock is not available to processors. Our goal is to assign values of the logical clock to each instruction, so that these values appear to be readings of the global clock. That is, it possible to postpone or advance the instants when instructions are executed in such a way, that each instruction x that has been assigned a value t_x of the logical clock, is executed exactly at the instant t_x of the global clock, and that the resulting execution is a valid one, in the sense that it can actually occur when the algorithm is run with the modified delays.

The Logical-Clock algorithm assigns logical time to each instruction. Each processor has a local variable called *counter*. This variable is initially zero and it gets incremented every time processor executes an instruction. Specifically, when a processor executes any instruction other than sending or receiving a message, the variable *counter* gets incremented by one. When a processor sends a message, it increments the variable by one, and attaches the resulting value to the message. When a processor receives a message, then the processor retrieves the value attached to the message, then calculates the maximum of the value and the current value of *counter*, increments the maximum by one, and assigns the result to the *counter* variable. Note that every time instruction is executed, the value of *counter* is incremented by at least one, and so it grows as processor keeps on executing instructions. The value of logical time assigned to instruction x is defined as the pair $(counter, id)$, where *counter* is the value of the variable *counter* right after the instruction has been executed, and *id* is the identifier of the processor. The values of logical time form a total order, where pairs are compared lexicographically. This logical time is also called Lamport time. We define t_x to be a quotient $counter + 1/(id + 1)$, which is an equivalent way to represent the pair.

Remark 13.21 *For any execution, logical time satisfies three conditions:*
(i) if an instruction x is performed by a processor before an instruction y is performed by the same processor, then the logical time of x is strictly smaller than that of y,
(ii) any two distinct instructions of any two processors get assigned different logical times,

(iii) if instruction x sends a message and instruction y receives this message, then the logical time of x is strictly smaller than that of y.

Our goal now is to argue that logical clock provides to processors the illusion of global clock. Intuitively, the reason why such an illusion can be created is that we can take any execution of a deterministic algorithm, compute the logical time t_x of each instruction x, and run the execution again delaying or speeding up processors and messages in such a way that each instruction x is executed at the instant t_x of the global clock. Thus, without access to a hardware clock or other external measurements not captured in our model, the processors cannot distinguish the reading of logical clock from the reading of a real global clock. Formally, the reason why the re-timed sequence is a valid execution that is indistinguishable from the original execution, is summarised in the subsequent corollary that follows directly from Remark 13.21.

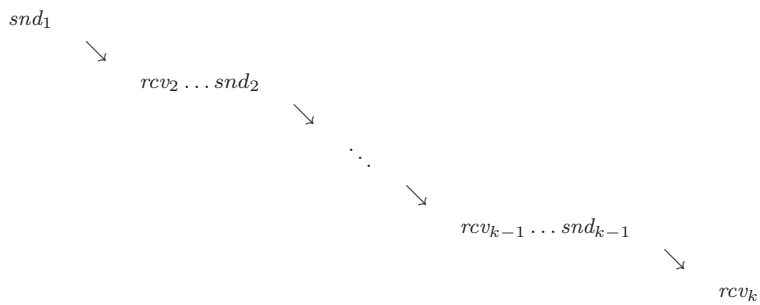
Corollary 13.22 *For any execution α, let T be the assignment of logical time to instructions, and let β be the sequence of instructions ordered by their logical time in α. Then for each processor, the subsequence of instructions executed by the processor in α is the same as the subsequence in β. Moreover, each message is received in β after it is sent in β.*

13.5.2. Causality

In a system execution, an instruction can affect another instruction by altering the state of the computation in which the second instruction executes. We say that one instruction can ***causally*** affect (or influence) another, if the information that one instruction produces can be passed on to the other instruction. Recall that in our model of distributed system, each instruction is executed at a distinct instant of global time, but processors do not have access to the reading of the global clock. Let us illustrate causality. If two instructions are executed by the same processor, then we could say that the instruction executed earlier can causally affect the instruction executed later, because it is possible that the result of executing the former instruction was used when the later instruction was executed. We stress the word possible, because in fact the later instruction may not use any information produced by the former. However, when defining causality, we simplify the problem of capturing how processors influence other processors, and focus on what is possible. If two instructions x and y are executed by two different processors, then we could say that instruction x can causally affect instruction y, when the processor that executes x sends a message when or after executing x, and the message is delivered before or during the execution of y at the other processor. It may also be the case that influence is passed on through intermediate processors or multiple instructions executed by processors, before reaching the second processor.

We will formally define the intuition that one instruction can causally affect another in terms of a relation called ***happens before,*** and that relates pairs of instructions. The relation is defined for a given execution, i.e., we fix a sequence of instructions executed by the algorithm and instances of global clock when the instructions were executed, and define which pairs of instructions are related by the

happens before relation. The relation is introduced in two steps. If instructions x and y are executed by the same processor, then we say that x *happens before* y if and only if x is executed before y. When x and y are executed by two different processors, then we say that x happens before y if and only if there is a chain of instructions and messages



for $k \geq 2$, such that snd_1 is either equal to x or is executed after x by the same processor that executes x; rcv_k is either equal to y or is executed before y by the same processor that executes y; rcv_h is executed before snd_h by the same processor, $2 \leq h < k$; and snd_h sends a message that is received by rcv_{h+1} , $1 \leq h < k$. Note that no instruction happens before itself. We write $x <_{HB} y$ when x happens before y. We omit the reference to the execution for which the relation is defined, because it will be clear from the context which execution we mean. We say that two instructions x and y are *concurrent* when neither $x <_{HB} y$ nor $y <_{HB} x$. The question stands how processors can determine if one instruction happens before another in a given execution according to our definition. This question can be answered through a generalisation of the Logical-Clock algorithm presented earlier. This generalisation is called vector clocks.

The Vector-Clocks algorithm allows processors to relate instructions, and this relation is exactly the happens before relation. Each processor p_i maintains a vector V_i of n integers. The j-th coordinate of the vector is denoted by $V_i[j]$. The vector is initialised to the zero vector $(0, \ldots, 0)$. A vector is modified each time processor executes an instruction, in a way similar to the way *counter* was modified in the Logical-Clock algorithm. Specifically, when a processor p_i executes any instruction other than sending or receiving a message, the coordinate $V_i[i]$ gets incremented by one, and other coordinates remain intact. When a processor sends a message, it increments $V_i[i]$ by one, and attaches the resulting vector V_i to the message. When a processor p_j receives a message, then the processor retrieves the vector V attached to the message, calculates coordinate-wise maximum of the current vector V_j and the vector V, except for coordinate $V_j[j]$ that gets incremented by one, and assigns the result to the variable V_j.

$V_j[j] \leftarrow V_j[j] + 1$
for all $k \in [n] \setminus \{j\}$
    $V_j[k] \leftarrow \max\{V_j[k], V[k]\}$

We label each instruction x executed by processor p_i with the value of the vector V_i right after the instruction has been executed. The label is denoted by $VT(x)$ and is called ***vector timestamp*** of instruction x. Intuitively, $VT(x)$ represents the knowledge of processor p_i about how many instructions each processor has executed at the moment when p_i has executed instruction x. This knowledge may be obsolete.

Vector timestamps can be used to order instructions that have been executed. Specifically, given two instructions x and y, and their vector timestamps $VT(x)$ and $VT(y)$, we write that $x \leq_{VT} y$ when the vector $VT(x)$ is majorised by the vector $VT(y)$ i.e., for all k, the coordinate $VT(x)[k]$ is at most the corresponding coordinate $VT(y)[k]$. We write $x <_{VT} y$ when $x \leq_{VT} y$ but $VT(x) \neq VT(y)$.

The next theorem explains that the Vector-Clocks algorithm indeed implements the happens before relation, because we can decide if two instructions happen or not before each other, just by comparing the vector timestamps of the instructions.

Theorem 13.23 *For any execution and any two instructions x and y, $x <_{HB} y$ if and only if $x <_{VT} y$.*

Proof We first show the forward implication. Suppose that $x <_{HB} y$. Hence x and y are two different instructions. If the two instructions are executed on the same processor, then x must be executed before y. Only finite number of instructions have been executed by the time y has been executed. The Vector-Clock algorithm increases a coordinate by one as it calculates vector timestamps of instructions from x until y inclusive, and no coordinate is ever decreased. Thus $x <_{VT} y$. If x and y were executed on different processors, then by the definition of happens before relation, there must be a finite chain of instructions and messages leading from x to y. But then by the Vector-Clock algorithm, the value of a coordinate of vector timestamp gets increased at each move, as we move along the chain, and so again $x <_{VT} y$.

Now we show the reverse implication. Suppose that it is not the case that $x <_{HB} y$. We consider a few subcases always concluding that it is not that case that $x <_{VT} y$. First, it could be the case that x and y are the same instruction. But then obviously vector clocks assigned to x and y are the same, and so it cannot be the case that $x <_{VT} y$. Let us, therefore, assume that x and y are different instructions. If they are executed by the same processor, then x cannot be executed before y, and so x is executed after y. Thus, by monotonicity of vector timestamps, $y <_{VT} x$, and so it is not the case that $x <_{VT} y$. The final subcase is when x and y are executed by two distinct processors p_i and p_j. Let us focus on the component i of vector clock V_i of processor p_i right after x was executed. Let its value be k. Recall that other processors can only increase the value of their components i by adopting the value sent by other processors. Hence, in order for the value of component i of processor p_j to be k or more at the moment y is executed, there must be a chain of instructions and messages that passes a value at least k, originating at processor p_i. This chain starts at x or at an instruction executed by p_i subsequent to x. But the existence of such chain would imply that x happens before y, which we assumed was not the case. So the component i of vector clock $VT(y)$ is strictly smaller than the component i of vector clock $VT(x)$. Thus it cannot be the case that $x <_{VT} y$. ∎

This theorem tells us that we can decide if two distinct instructions x and y are concurrent, by checking that it is not the case that $VT(x) < VT(y)$ nor is it the case that $VT(x) > VT(y)$.

13.5.3. Consistent state

The happens before relation can be used to compute a global state of distributed system, such that this state is in some sense consistent. Shortly, we will formally define the notion of consistency. Each processor executes instructions. A ***cut*** K is defined as a vector $K = (k_1, \ldots, k_n)$ of non-negative integers. Intuitively, the vector K denotes the states of processors. Formally, k_i denotes the number of instructions that processor p_i has executed. Not all cuts correspond to collections of states of distributed processors that could be considered natural or consistent. For example, if a processor p_i has received a message from p_j and we record the state of p_i in the cut by making k_i appropriately large, but make k_j so small that the cut contains the state of the sender before the moment when the message was sent, then we could say that such cut is not natural—there are instructions recorded in the cut that are causally affected by instructions that are not recorded in the cut. Such cuts we consider not consistent and so undesirable. Formally, a cut $K = (k_1, \ldots, k_n)$ is inconsistent when there are processors p_i and p_j such that the instruction number k_i of processor p_i is causally affected by an instruction subsequent to instruction number k_j of processor p_j. So in an inconsistent cut there is a message that "crosses" the cut in a backward direction. Any cut that is not inconsistent is called a ***consistent cut,***

The Consistent-Cut algorithm uses vector timestamps to find a consistent cut. We assume that each processor is given the same cut $K = (k_1, \ldots, k_n)$ as an input. Then processors must determine a consistent cut K' that is majorised by K. Each processor p_i has an infinite table $VT_i[0, 1, 2, \ldots]$ of vectors. Processor executes instructions, and stores vector timestamps in consecutive entries of the table. Specifically, entry m of the table is the vector timestamp $VT_i[m]$ of the m-th instruction executed by the processor; we define $VT_i[0]$ to be the zero vector. Processor p_i begins calculating a cut right after the moment when the processor has executed instruction number k_i. The processor determines the largest number $k'_i \geq 0$ that is at most k_i, such that the vector $VT_i[k'_i]$ is majorised by K. The vector $K' = (k'_1, \ldots, k'_n)$ that processors collectively find turns out to be a consistent cut.

Theorem 13.24 *For any cut K, the cut K' computed by the* Consistent-Cut *algorithm is a consistent cut majorised by K.*

Proof First observe that there is no need to consider entries of VT_i further than k_i. Each of these entries is not majorised by K, because the i-th coordinate of any of these vectors is strictly larger than k_i. So we can indeed focus on searching among the first k_i entries of VT_i. Let $k'_i \geq 0$ be the largest entry such that the vector $VT_i[k'_i]$ is majorised by the vector K. We know that such vector exists, because $VT_i[0]$ is a zero vector, and such vector is majorised by any cut K.

We argue that $(k'_1, \ldots, k'_n)$ is a consistent cut by way of contradiction. Suppose that the vector $(k'_1, \ldots, k'_n)$ is an inconsistent cut. Then, by definition, there are processors p_i and p_j such that there is an instruction x of processor p_i subsequent to

instruction number k'_i, such that x happens before instruction number k'_j of processor p_j. Recall that k'_i is the furthest entry of VT_i majorised by K. So entry $k'_i + 1$ is not majorised by K, and since all subsequent entries, including the one for instruction x, can have only larger coordinates, the entries are not majorised by K either. But, x happens before instruction number k'_j, so entry k'_j can only have lager coordinates than respective coordinates of the entry corresponding to x, and so $VT_j[k'_j]$ cannot be majorised by K either. This contradicts the assumption that $VT_j[k'_j]$ is majorised by K. Therefore, $(k'_1, \ldots, k'_n)$ must be a consistent cut. ■

There is a trivial algorithm for finding a consistent cut. The algorithm picks $K' = (0, \ldots, 0)$. However, the CONSISTENT-CUT algorithm is better in the sense that the consistent cut found is maximal. That this is indeed true, is left as an exercise.

There is an alternative way to find a consistent cut. The Consistent Cut algorithm requires that we attach vector timestamps to messages and remember vector timestamps for all instructions executed so far by the algorithm A which consistent cut we want to compute. This may be too costly. The algorithm called DISTRIBUTED-SNAPSHOT avoids this cost. In the algorithm, a processor initiates the calculation of consistent cut by flooding the network with a special message that acts like a sword that cuts the execution of algorithm A consistently. In order to prove that the cut is indeed consistent, we require that messages are received by the recipient in the order they were sent by the sender. Such ordering can be implemented using sequence number.

In the DISTRIBUTED-SNAPSHOT algorithm, each processor p_i has a variable called *counter* that counts the number of instructions of algorithm A executed by the processor so far. In addition the processor has a variable k_i that will store the i-th coordinate of the cut. This variable is initialised to $\perp$. Since the variables *counter* only count the instructions of algorithm A, the instructions of DISTRIBUTED-SNAPSHOT algorithm do not affect the *counter* variables. In some sense the snapshot algorithm runs in the "background". Suppose that there is exactly one processor that can decide to take a snapshot of the distributed system. Upon deciding, the processor "floods" the network with a special message <*Snapshot*>. Specifically, the processor sends the message to all its neighbours and assigns *counter* to k_i. Whenever a processor p_j receives the message and the variable k_j is still $\perp$, then the processor sends <*Snapshot*> message to all its neighbours and assigns *current* to k_j. The sending of <*Snapshot*> messages and assignment are done by the processor without executing any instruction of A (we can think of DISTRIBUTED-SNAPSHOT algorithm as an "interrupt"). The algorithm calculates a consistent cut.

Theorem 13.25 *Let for any processors p_i and p_j, the messages sent from p_i to p_j be received in the order they are sent. The* DISTRIBUTED-SNAPSHOT *algorithm eventually finds a consistent cut $(k_1, \ldots, k_n)$. The algorithm sends $O(e)$ messages, where e is the number of edges in the graph.*

Proof The fact that each variable k_i is eventually different from $\perp$ follows from our model, because we assumed that instructions are eventually executed and messages are eventually received, so the <*Snapshot*> messages will eventually reach all nodes.

Suppose that $(k_1, \ldots, k_n)$ is not a consistent cut. Then there is a processor p_j such that instruction number $k_j + 1$ or later sends a message $<M>$ other than $<Snapshot>$, and the message is received on or before a processor p_i executes instruction number k_i. So the message $<M>$ must have been sent after the message $<Snapshot>$ was sent from p_j to p_i. But messages are received in the order they are sent, so p_i processes $<Snapshot>$ before it processes $<M>$. But then message $<M>$ arrives after snapshot was taken at p_i. This is a desired contradiction. ■

Exercises

13.5-1 Show that logical time preserves the *happens before* ($<_{HB}$) relation. That is, show that if for events x and y it is the case that $x <_{HB} y$, then $LT(x) < LT(y)$, where $LT(\cdot)$ is the logical time of an event.

13.5-2 Show that any vector clock that captures concurrency between n processors must have at least n coordinates.

13.5-3 Show that the vector K' calculated by the algorithm CONSISTENT-CUT is in fact a maximal consistent cut majorised by K. That is that there is no K'' that majorises K' and is different from K', such that K'' is majorised by K.

13.6. Communication services

Among the fundamental problems in distributed systems where processors communicate by message passing are the tasks of spreading and gathering information. Many distributed algorithms for communication networks can be constructed using building blocks that implement various broadcast and multicast services. In this section we present some basic communication services in the message-passing model. Such services typically need to satisfy some quality of service requirements dealing with ordering of messages and reliability. We first focus on broadcast services, then we discuss more general multicast services.

13.6.1. Properties of broadcast services

In the broadcast problem, a selected processor p_i, called a ***source*** or a *sender*, has the message m, which must be delivered to all processors in the system (including the source). The interface of the broadcast service is specified as follows:

bc-send$_i(m, qos)$ **:** an event of processor p_i that sends a message m to all processors.

bc-recv$_i(m, j, qos)$ **:** an event of processor p_i that receives a message m sent by processor p_j.

In above definitions *qos* denotes the ***quality of service*** provided by the system. We consider two kinds of quality service:

Ordering: how the order of received messages depends on the order of messages sent by the source?

Reliability: how the set of received messages depends on the failures in the system?

The basic model of a message-passing distributed system normally does not guarantee any ordering or reliability of messaging operations. In the basic model we only assume that each pair of processors is connected by a link, and message delivery is independent on each link — the order of received messages may not be related to the order of the sent messages, and messages may be lost in the case of crashes of senders or receivers.

We present some of the most useful requirements for ordering and reliability of broadcast services. The main question we address is how to implement a stronger service on top of the weaker service, starting with the basic system model.

Variants of ordering requirements. Applying the definition of *happens before* to messages, we say that message m happens before message m' if either m and m' are sent by the same processor and m is sent before m', or the bc-recv event for m happens before the bc-send event for m'.

We identify four common broadcast services with respect to the message ordering properties:

Basic Broadcast: no order of messages is guaranteed.

Single-Source FIFO (*first-in-first-out*): messages sent by one processor are received by each processor in the same order as sent; more precisely, for all processors p_i, p_j and messages m, m', if processor p_i sends m before it sends m' then processor p_j does not receive message m' before message m.

Causal Order: messages are received in the same order as they happen; more precisely, for all messages m, m' and every processor p_i, if m happens before m' then p_i does not receive m' before m.

Total Order: the same order of received messages is preserved in each processor; more precisely, for all processors p_i, p_j and messages m, m', if processor p_i receives m before it receives m' then processor p_j does not receive message m' before message m.

It is easy to see that Causal Order implies Single-Source FIFO requirements (since the relation "happens before" for messages includes the order of messages sent by one processor), and each of the given services trivially implies Basic Broadcast. There are no additional relations between these four services. For example, there are executions that satisfy Single-Source FIFO property, but not Causal Order. Consider two processors p_0 and p_1. In the first event p_0 broadcasts message m, next processor p_1 receives m, and then p_1 broadcasts message m'. It follows that m happens before m'. But if processor p_0 receives m' before m, which may happen, then this execution violates Causal Order. Note that trivially Single-Source FIFO requirement is preserved, since each processor broadcasts only one message.

We denote by *bb* the Basic Broadcast service, by *ssf* the Single-Source FIFO, by *co* the Causal Order and by *to* the Total Order service.

Reliability requirements. In the model without failures we would like to guarantee the following properties of broadcast services:

Integrity: each message m received in event bc-recv has been sent in some bc-send event.

No-Duplicates: each processor receives a message not more than once.

Liveness: each message sent is received by all processors.

In the model with failures we define the notion of *reliable* broadcast service, which satisfies Integrity, No-Duplicates and two kinds of Liveness properties:

Nonfaulty Liveness: each message m sent by non-faulty processor p_i must be received by every non-faulty processor.

Faulty Liveness: each message sent by a faulty processor is either received by all non-faulty processors or by none of them.

We denote by *rbb* the **R**eliable **B**asic **B**roadcast service, by *rssf* the **R**eliable **S**ingle-**S**ource **F**IFO, by *rco* the **R**eliable **C**ausal **O**rder, and by *rto* the **R**eliable **T**otal **O**rder service.

13.6.2. Ordered broadcast services

We now describe implementations of algorithms for various broadcast services.

Implementing basic broadcast on top of asynchronous point-to-point messaging. The *bb* service is implemented as follows. If event bc-send$_i(m, bb)$ occurs then processor p_i sends message m via every link from p_i to p_j, where $0 \leq i \leq n-1$. If a message m comes to processor p_j then it enables event bc-recv$_j(m, i, bb)$.

To provide reliability we do the following. We build the reliable broadcast on the top of basic broadcast service. When bc-send$_i(m, rbb)$ occurs, processor p_i enables event bc-send$_i(\langle m, i\rangle, bb)$. If event bc-recv$_j(\langle m, i\rangle, k, bb)$ occurs and message-coordinate m appears for the first time then processor p_j first enables event bc-send$_j(\langle m, i\rangle, bb)$ (to inform other non-faulty processors about message m in case when processor p_i is faulty), and next enables event bc-recv$_j(m, i, rbb)$.

We prove that the above algorithm provides reliability for the basic broadcast service. First observe that Integrity and No-Duplicates properties follow directly from the fact that each processor p_j enables bc-recv$_j(m, i, rbb)$ only if message-coordinate m is received for the first time. Nonfaulty liveness is preserved since links between non-faulty processors enables events bc-recv$_j(\cdot, \cdot, bb)$ correctly. Faulty Liveness is guaranteed by the fact that if there is a non-faulty processor p_j which receives message m from the faulty source p_i, then before enabling bc-recv$_j(m, i, rbb)$ processor p_j sends message m using bc-send$_j$ event. Since p_j is non-faulty, each non-faulty processor p_k gets message m in some bc-recv$_k(\langle m, i\rangle, \cdot, bb)$ event, and then accepts it (enabling event bc-recv$_k(m, i, rbb)$) during the first such event.

Implementing single-source FIFO on top of basic broadcast service. Each processor p_i has its own counter (timestamp), initialised to 0. If event bc-send$_i(m, ssf)$ occurs then processor p_i sends message m with its current timestamp attached, using bc-send$_i(< m, timestamp >, bb)$. If an event bc-recv$_j(< m, t >,$

i, bb) occurs then processor p_j enables event bc-recv$_j(m, i, ssf)$ just after events bc-recv$_j(m_0, i, ssf)$, ..., bc-recv$_j(m_{t-1}, i, ssf)$ have been enabled, where $m_0, \ldots, m_{t-1}$ are the messages such that events bc-recv$_j(< m_0, 0 >, i, bb)$,...,bc-recv$_j(< m_{t-1}, t-1 >, i, bb)$ have been enabled.

Note that if we use reliable Basic Broadcast instead of Basic Broadcast as the background service, the above implementation of Single-Source FIFO becomes Reliable Single-Source FIFO service. We leave the proof to the reader as an exercise.

Implementing causal order and total order on the top of single-source FIFO service. We present an ordered broadcast algorithm which works in the asynchronous message-passing system providing single-source FIFO broadcast service. It uses the idea of timestamps, but in more advanced way than in the implementation of *ssf*. We denote by *cto* the service satisfying causal and total orders requirements.

Each processor p_i maintains in a local array T its own increasing counter (timestamp), and the estimated values of timestamps of other processors. Timestamps are used to mark messages before sending—if p_i is going to broadcast a message, it increases its timestamp and uses it to tag this message (lines 11-13). During the execution processor p_i estimates values of timestamps of other processors in the local vector T—if processor p_i receives a message from processor p_j with a tag t (timestamp of p_j), it puts t into $T[j]$ (lines 23–32). Processor p_i sets its current timestamp to be the maximum of the estimated timestamps in the vector T plus one (lines 24–26). After updating the timestamp processor sends an update message. Processor accepts a message m with associated timestamp t from processor j if pair (t, j) is the smallest among other received messages (line 42), and each processor has at least as large a timestamp as known by processor p_i (line 43). The details are given in the code below.

ORDERED-BROADCAST

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  initialisation
02      T[j] ← 0 for every 0 ≤ j ≤ n − 1

11  if bc-send_i(m, cto) occurs
12     then T[i] ← T[i] + 1
13          enable bc-send_i(< m, T[i] >, ssf)

21  if bc-recv_i(< m, t >, j, ssf) occurs
22     then add triple (m, t, j) to pending
23          T[j] ← t
24     if t > T[i]
25        then T[i] ← t
26             enable bc-send_i(< update, T[i] >, ssf)
```

31 **if** bc-recv$_i$($< update, t >, j, ssf$) occurs
32 **then** $T[j] \leftarrow t$

41 **if**
42 (m, t, j) is the pending triple with the smallest (t, j) and
$t \leq T[k]$ for every $0 \leq k \leq n-1$
43 **then enable** bc-recv$_i(m, j, cto)$
44 **remove** triple (m, t, j) from pending

Ordered-Broadcast satisfies the causal order requirement. We leave the proof to the reader as an exercise (in the latter part we show how to achieve stronger reliable causal order service and provide the proof for that stronger case).

Theorem 13.26 Ordered-Broadcast *satisfies the total order requirement.*

Proof Integrity follows from the fact that each processor can enable event bc-recv$_i(m, j, cto)$ only if the triple (m, t, j) is pending (lines 41–45), which may happen after receiving a message m from processor j (lines 21–22). No-Duplicates property is guaranteed by the fact that there is at most one pending triple containing message m sent by processor j (lines 13 and 21–22).

Liveness follows from the fact that each pending triple satisfies conditions in lines 42–43 in some moment of the execution. The proof of this fact is by induction on the events in the execution — suppose to the contrary that (m, t, j) is the triple with smallest (t, j) which does not satisfy conditions in lines 42–43 at any moment of the execution. It follows that there is a moment from which triple (m, t, j) has smallest (t, j) coordinates among pending triples in processor p_i. Hence, starting from this moment, it must violate condition in line 43 for some k. Note that $k \neq i, j$, by updating rules in lines 23–25. It follows that processor p_i never receives a message from p_k with timestamp greater than $t-1$, which by updating rules in lines 24-26 means that processor p_k never receives a message $< m, t >$ from j, which contradicts the liveness property of ssf broadcast service.

To prove Total Order property it is sufficient to prove that for every processor p_i and messages m, m' sent by processors p_k, p_l with timestamps t, t' respectively, each of the triples (m, t, k), (m', t', l) are accepted according to the lexicographic order of $(t, k), (t', l)$. There are two cases.

Case 1. Both triples are pending in processor p_i at some moment of the execution. Then condition in line 42 guarantees acceptance in order of $(t, k), (t', l)$.

Case 2. Triple (m, t, k) (without loss of generality) is accepted by processor p_i before triple (m', t', l) is pending. If $(t, k) < (t', l)$ then still the acceptance is according to the order of $(t, k), (t', l)$. Otherwise $(t, k) > (t', l)$, and by condition in line 43 we get in particular that $t \leq T[l]$, and consequently $t' \leq T[l]$. This can not happen because of the ssf requirement and the assumption that processor p_i has not yet received message $< m', t' >$ from l via the ssf broadcast service. ∎

Now we address reliable versions of Causal Order and Total Order services. A Reliable Causal Order requirements can be implemented on the top of Reliable

Basic Broadcast service in asynchronous message-passing system with processor crashes using the following algorithm. It uses the same data structures as previous Ordered-Bbroadcast. The main difference between reliable Causally-Ordered-Broadcast and Ordered-Broadcast are as follows: instead of using integer timestamps processors use vector timestamps T, and they do not estimate timestamps of other processors, only compare in lexicographic order their own (vector) timestamps with received ones. The intuition behind vector timestamp of processor p_i is that it stores information how many messages have been sent by p_i and how many have been accepted by p_i from every p_k, where $k \neq i$.

In the course of the algorithm processor p_i increases corresponding position i in its vector timestamp T before sending a new message (line 12), and increases jth position of its vector timestamp after accepting new message from processor p_j (line 38). After receiving a new message from processor p_j together with its vector timestamp $\hat{T}$, processor p_i adds triple $(m, \hat{T}, j)$ to pending and accepts this triple if it is first not accepted message received from processor p_j (condition in line 33) and the number of accepted messages (from each processor $p_k \neq p_i$) by processor p_j was not bigger in the moment of sending m than it is now in processor p_i (condition in line 34). Detailed code of the algorithm follows.

Reliable-Causally-Ordered-Broadcast

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  initialisation
02      T[j] ← 0 for every 0 ≤ j ≤ n − 1
03      pending list is empty

11  if bc-send_i(m, rco) occurs
12     then T[i] ← T[i] + 1
13          enable bc-send_i(< m, T >, rbb)

21  if bc-recv_i(< m, T̂ >, j, rbb) occurs
22     then add triple (m, T̂, j) to pending

31  if (m, T̂, j) is the pending triple, and
32     T̂[j] = T[j] + 1, and
33     T̂[k] ≤ T[k] for every k ≠ i
34     then enable bc-recv_i(m, j, rco)
35          remove triple (m, T̂, j) from pending
36          T[j] ← T[j] + 1
```

We argue that the algorithm Reliable-Causally-Ordered-Broadcast provides Reliable Causal Order broadcast service on the top of the system equipped with the Reliable Basic Broadcast service. Integrity and No-Duplicate properties are guaranteed by *rbb* broadcast service and facts that each message is added to pending at most once and non-received message is never added to pending. Non-faulty and Faulty Liveness can be proved by one induction on the execution, using

facts that non-faulty processors have received all messages sent, which guarantees that conditions in lines 33-34 are eventually satisfied. Causal Order requirement holds since if message m happens before message m' then each processor p_i accepts messages m, m' according to the lexicographic order of $\hat{T}, \hat{T}'$, and these vector-arrays are comparable in this case. Details are left to the reader.

Note that Reliable Total Order broadcast service can not be implemented in the general asynchronous setting with processor crashes, since it would solve consensus in this model — first accepted message would determine the agreement value (against the fact that consensus is not solvable in the general model).

13.6.3. Multicast services

Multicast services are similar to the broadcast services, except each multicast message is destined for a specified subset of all processors.In the multicast service we provide two types of events, where *qos* denotes a quality of service required:

mc-send$_i(m, D,$ ***qos***$)$ **:** an event of processor p_i which sends a message m together with its id to all processors in a destination set $D \subseteq \{0, \ldots, n-1\}$.

mc-recv$_i(m, j,$ ***qos***$)$ **:** an event of processor p_i which receives a message m sent by processor p_j.

Note that the event mc-recv is similar to bc-recv.

As in case of a broadcast service, we would like to provide useful ordering and reliable properties of the multicast services. We can adapt ordering requirements from the broadcast services. Basic Multicast does not require any ordering properties. Single-Source FIFO requires that if one processor multicasts messages (possibly to different destination sets), then the messages received in each processors (if any) must be received in the same order as sent by the source. Definition of Causal Order remains the same. Instead of Total Order, which is difficult to achieve since destination sets may be different, we define another ordering property:

Sub-Total Order: orders of received messages in all processors may be extended to the total order of messages; more precisely, for any messages m, m' and processors p_i, p_j, if p_i and p_j receives both messages m, m' then they are received in the same order by p_i and p_j.

The reliability conditions for multicast are somewhat different from the conditions for reliable broadcast.

Integrity: each message m received in event mc-recv$_i$ was sent in some mc-send event with destination set containing processor p_i.

No Duplicates: each processor receives a message not more than once.

Nonfaulty Liveness: each message m sent by non-faulty processor p_i must be received in every non-faulty processor in the destination set.

Faulty Liveness: each message sent by a faulty processor is either received by all non-faulty processors in the destination set or by none of them.

One way of implementing ordered and reliable multicast services is to use the corresponding broadcast services (for Sub-Total Order the corresponding broad-

cast requirement is Total Order). More precisely, if event mc-send$_i(m, D, qos)$ occurs processor p_i enables event bc-send$_i(< m, D >, qos)$. When an event bc-recv$_j(< m, D >, i, qos)$ occurs, processor p_j enables event mc-recv$_j(m, i, qos)$ if $p_j \in D$, otherwise it ignores this event. The proof that such method provides required multicast quality of service is left as an exercise.

13.7. Rumor collection algorithms

Reliable multicast services can be used as building blocks in constructing algorithms for more advanced communication problems. In this section we illustrate this method for the problem of collecting rumors by synchronous processors prone to crashes. (Since we consider only fair executions, we assume that at least one processor remains operational to the end of the computation).

13.7.1. Rumor collection problem and requirements

The classic problem of ***collecting rumors,*** or ***gossip,*** is defined as follows:

> At the beginning, each processor has its distinct piece of information, called a *rumor,* the goal is to make every processor know all the rumors.

However in the model with processor crashes we need to re-define the gossip problem to respect crash failures of processors. Both Integrity and No-Duplicates properties are the same as in the reliable broadcast service, the only difference (which follows from the specification of the gossip problem) is in Liveness requirements:

Non-faulty Liveness: the rumor of every non-faulty processor must be known by each non-faulty processor.

Faulty Liveness: if processor p_i has crashed during execution then each non-faulty processor either knows the rumor of p_i or knows that p_i is crashed.

The efficiency of gossip algorithms is measured in terms of time and message complexity. Time complexity measures number of (synchronous) steps from the beginning to the termination. Message complexity measures the total number of point-to-point messages sent (more precisely, if a processor sends a message to three other processors in one synchronous step, it contributes three to the message complexity).

The following simple algorithm completes gossip in just one synchronous step: each processor broadcasts its rumor to all processors. The algorithm is correct, because each message received contains a rumor, and a message not received means the failure of its sender. A drawback of such a solution is that a quadratic number of messages could be sent, which is quite inefficient.

We would like to perform gossip not only quickly, but also with fewer point-to-point messages. There is a natural trade-off between time and communication. Note that in the system without processor crashes such a trade-off may be achieved, e.g., sending messages over the (almost) complete binary tree, and then time complexity is $O(\lg n)$, while the message complexity is $O(n \lg n)$. Hence by slightly increasing time complexity we may achieve almost linear improvement in message complexity. How-

ever, if the underlying communication network is prone to failures of components, then irregular failure patterns disturb a flow of information and make gossiping last longer. The question we address in this section is what is the best trade-off between time and message complexity in the model with processor crashes?

13.7.2. Efficient gossip algorithms

In this part we describe the family of gossip algorithms, among which we can find some efficient ones. They are all based on the same generic code, and their efficiency depends on the quality of two data structures put in the generic algorithm. Our goal is to prove that we may find some of those data structures that obtained algorithm is always correct, and efficient if the number of crashes in the execution is at most f, where $f \leq n-1$ is a parameter.

We start with description of these structures: communication graph and communication schedules.

Communication graph. A graph $G = (V, E)$ consists of a set V of *vertices* and a set E of *edges.* Graphs in this paper are always ***simple,*** which means that edges are pairs of vertices, with no direction associated with them. Graphs are used to describe communication patterns. The set V of vertices of a graph consists of the processors of the underlying distributed system. Edges in E determine the pairs of processors that communicate directly by exchanging messages, but this does not necessarily mean an existence of a physical link between them. We abstract form the communication mechanism: messages that are exchanged between two vertices connected by an edge in E may need to be routed and traverse a possibly long path in the underlying physical communication network. Graph topologies we use, for a given number n of processors, vary depending on an upper bound f on the number of crashes we would like to tolerate in an execution. A graph that matters, at a given point in an execution, is the one induced by the processors that have not crashed till this step of the execution.

To obtain an efficient gossip algorithm, communication graphs should satisfy some suitable properties, for example the following property $\mathcal{R}(n, f)$:

Definition 13.27 *Let $f < n$ be a pair of positive integers. Graph G is said to* satisfy property $\mathcal{R}(n, f)$, *if G has n vertices, and if, for each subgraph $R \subseteq G$ of size at least $n - f$, there is a subgraph $P(R)$ of G, such that the following hold:*

1 : $P(R) \subseteq R$
2 : $|P(R)| = |R|/7$
3 : *The diameter of $P(R)$ is at most* $2 + 30 \ln n$
4 : *If $R_1 \subseteq R_2$, then $P(R_1) \subseteq P(R_2)$*

In the above definition, clause (1.) requires the existence of subgraphs $P(R)$ whose vertices has the potential of (informally) inheriting the properties of the vertices of R, clause (2.) requires the subgraphs to be sufficiently large, linear in size, clause (3.) requires the existence of paths in the subgraphs that can be used for communication of at most logarithmic length, and clause (4.) imposes monotonicity

on the required subgraphs.

Observe that graph $P(R)$ is connected, even if R is not, since its diameter is finite. The following result shows that graphs satisfying property $\mathcal{R}(n,f)$ can be constructed, and that their degree is not too large.

Theorem 13.28 *For each $f < n$, there exists a graph $G(n,f)$ satisfying property $\mathcal{R}(n,f)$. The maximum degree Δ of graph $G(n,f)$ is $O\big(\frac{n}{n-f}\big)^{1.837}$.*

Communication schedules. A *local permutation* is a permutation of all the integers in the range $[0\,..\,n-1]$. We assume that prior the computation there is given set Π of n local permutations. Each processor p_i has such a permutation π_i from Π. For simplicity we assume that $\pi_i(0) = p_i$. Local permutation is used to collect rumor in systematic way according to the order given by this permutation, while communication graphs are rather used to exchange already collected rumors within large and compact non-faulty graph component.

Generic algorithm. We start with specifying a goal that gossiping algorithms need to achieve. We say that *processor p_i has heard about processor p_j* if either p_i knows the original input rumor of p_j or p knows that p_j has already failed. We may reformulate correctness of a gossiping algorithm in terms of hearing about other processors: algorithm is correct if Integrity and No-Duplicates properties are satisfied and if each processor has hard about any other processor by the termination of the algorithm.

The code of a gossiping algorithm includes objects that depend on the number n of processors in the system, and also on the bound $f < n$ on the number of failures which are "efficiently tolerated" (if the number of failures is at most f then message complexity of design algorithm is small). The additional parameter is a termination threshold τ which influences time complexity of the specific implementation of the generic gossip scheme. Our goal is to construct the generic gossip algorithm which is correct for any additional parameters f, τ and any communication graph and set of schedules, while efficient for some values f, τ and structures $G(n,f)$ and Π.

Each processor starts gossiping as a *collector*. Collectors seek actively information about rumors of the other processors, by sending direct inquiries to some of them. A collector becomes a ***disseminator*** after it has heard about all the processors. Processors with this status disseminate their knowledge by sending local views to selected other processors.

Local views. Each processor p_i starts with knowing only its ID and its input information $rumor_i$. To store incoming data, processor p_i maintains the following arrays:

$$\texttt{Rumors}_i,\ \texttt{Active}_i \text{ and } \texttt{Pending}_i,$$

each of size n. All these arrays are initialised to store the value `nil`. For an array $\texttt{X}_i$ of processor p_i, we denote its jth entry by $\texttt{X}_i[j]$ - intuitively this entry contains some information about processor p_j. The array `Rumor` is used to store all the rumors that a processor knows. At the start, processor p_i sets $\texttt{Rumors}_i[i]$ to its own input $rumor_i$. Each time processor p_i learns some $rumor_j$, it immediately sets $\texttt{Rumors}_i[j]$ to this

value. The array `Active` is used to store a set of all the processors that the owner of the array knows as crashed. Once processor p_i learns that some processor p_j has failed, it immediately sets $\texttt{Active}_i[j]$ to *failed*. Notice that processor p_i has heard about processor p_j, if one among the values $\texttt{Rumors}_i[j]$ and $\texttt{Active}_i[j]$ is not equal to NIL.

The purpose of using the array `Pending` is to facilitate dissemination. Each time processor p_i learns that some other processor p_j is fully informed, that is, it is either a disseminator itself or has been notified by a disseminator, then it marks this information in $\texttt{Pending}_i[j]$. Processor p_i uses the array $\texttt{Pending}_i$ to send dissemination messages in a systematic way, by scanning $\texttt{Pending}_i$ to find those processors that possibly still have not heard about some processor.

The following is a useful terminology about the current contents of the arrays `Active` and `Pending`. Processor p_j is said *to be active according to* p_i, if p_i has not yet received any information implying that p_j crashed, which is the same as having `nil` in $\texttt{Active}_i[j]$. Processor p_j is said *to need to be notified by* p_i if it is active according to p_i and $\texttt{Pending}_i[j]$ is equal to `nil`.

Phases. An execution of a gossiping algorithm starts with the processors initialising all the local objects. Processor p_i initialises its list $\texttt{Rumors}_i$ with `nil` at all the locations, except for the ith one, which is set equal to $rumor_i$. The remaining part of execution is structured as a loop, in which phases are iterated. Each phase consists of three parts: receiving messages, local computation, and multicasting messages. Phases are of two kinds: ***regular phase*** and ***ending phase.*** During regular phases processor: receives messages, updates local knowledge, checks its status, sends its knowledge to neighbours in communication graphs as well as inquiries about rumors and replies about its own rumor. During ending phases processor: receives messages, sends inquiries to all processors from which it has not heard yet, and replies about its own rumor. The regular phases are performed τ times; the number τ is a *termination threshold.* After this, the ending phase is performed four times. This defines a generic gossiping algorithm.

GENERIC-GOSSIP

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  initialisation
02      processor p_i becomes a collector
03      initialisation of arrays Rumors_i, Active_i and Pending_i

11  repeat τ times
12      perform regular phase

20  repeat 4 times
21      perform ending phase
```

Now we describe communication and kinds of messages used in regular and ending phases.

Graph and range messages used during regular phases. A processor p_i may send a message to its neighbour in the graph $G(n, f)$, provided that it is is still active

according to p_i. Such a message is called a ***graph*** one. Sending these messages only is not sufficient to complete gossiping, because the communication graph may become disconnected as a result of node crashes. Hence other messages are also sent, to cover all the processors in a systematic way. In this kind of communication processor p_i considers the processors as ordered by its local permutation π_i, that is, in the order $\pi_i(0), \pi_i(1), \ldots, \pi_i(n-1)$. Some of additional messages sent in this process are called ***range*** ones.

During regular phase processors send the following kind of range messages: inquiring, reply and notifying messages. A collector p_i sends an ***inquiring*** message to the first processor about which p_i has not heard yet. Each recipient of such a message sends back a range message that is called a ***reply*** one.

Disseminators send range messages also to subsets of processors. Such messages are called ***notifying*** ones. The target processor selected by disseminator p_i is the first one that still needs to be notified by p_i. Notifying messages need not to be replied to: a sender already knows the rumors of all the processors, that are active according to it, and the purpose of the message is to disseminate this knowledge.

REGULAR-PHASE

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  receive messages

11  perform local computation
12      update the local arrays
13      if p_i is a collector, that has already heard about all the processors
14          then p_i becomes a disseminator
15      compute set of destination processors: for each processor p_j
16          if p_j is active according to p_i and p_j is a neighbour of p_i in graph G(n, t)
17              then add p_j to destination set for a graph message
18          if p_i      is a collector and p_j is the first processor
                        about which p_i has not heard yet
19              then send an inquiring message to p_j
20          if p_i is a disseminator and p_j is the first processor
                    that needs to be notified by p_i
21              then send a notifying message to p_j
22          if p_j is a collector, from which an inquiring message was received
                    in the receiving step of this phase
23              then send a reply message to p_j

30  send graph/inquiring/notifying/reply messages to corresponding destination sets
```

Last-resort messages used during ending phases. Messages sent during the ending phases are called *last-resort* ones. These messages are categorised into inquiring, replying, and notifying, similarly as the corresponding range ones, which is because they serve a similar purpose. Collectors that have not heard about some processors yet send direct inquiries to *all* of these processors simultaneously. Such messages are called *inquiring* ones. They are replied to by the non-faulty recipients

in the next step, by way of sending *reply* messages. This phase converts *all* the collectors into disseminators. In the next phase, each disseminator sends a message to *all* the processors that need to be notified by it. Such messages are called *notifying* ones.

The number of graph messages, sent by a processor at a step of the regular phase, is at most as large as the maximum node degree in the communication graph. The number of range messages, sent by a processor in a step of the regular phase, is at most as large as the number of inquiries received plus a constant - hence the global number of point-to-point range messages sent by all processors during regular phases may be accounted as a constant times the number of inquiries sent (which is one per processor per phase). In contrast to that, there is no *a priori* upper bound on the number of messages sent during the ending phase. By choosing the termination threshold τ to be large enough, one may control how many rumors still needs to be collected during the ending phases.

Updating local view. A message sent by a processor carries its current local knowledge. More precisely, a message sent by processor p_i brings the following: the ID p_i, the arrays $\texttt{Rumors}_i$, $\texttt{Active}_i$, and $\texttt{Pending}_i$, and a label to notify the recipient about the character of the message. A label is selected from the following: *graph_message, inquiry_from_collector, notification_from_disseminator, this_is_a_reply*, their meaning is self-explanatory. A processor p_i scans a newly received message from some processor p_j to learn about rumors, failures, and the current status of other processors. It copies each rumor from the received copy of $\texttt{Rumors}_j$ into $\texttt{Rumors}_i$, unless it is already there. It sets $\texttt{Active}_i[k]$ to *failed,* if this value is at $\texttt{Active}_j[k]$. It sets $\texttt{Pending}_i[k]$ to *done,* if this value is at $\texttt{Pending}_j[k]$. It sets $\texttt{Pending}_i[j]$ to *done,* if p_j is a disseminator and the received message is a range one. If p_i is itself a disseminator, then it sets $\texttt{Pending}_i[j]$ to *done* immediately after sending a range message to p_j. If a processor p_i expects a message to come from processor p_j, for instance a graph one from a neighbour in the communication graph, or a reply one, and the message does not arrive, then p_i knows that processor p_j has failed, and it immediately sets $\texttt{Active}_i[j]$ to *failed.*

ENDING-PHASE

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  receive messages

11  perform local computation
12      update the local arrays
13      if p_i is a collector, that has already heard about all the processors
14          then p_i becomes a disseminator
15      compute set of destination processors: for each processor p_j
16          if p_i is a collector and it has not heard about p_j yet
17              then send an inquiring message to p_j
18          if p_i is a disseminator and p_j needs to be notified by p_i
19              then send a notifying message to p_j
```

```
20      if an inquiring message was received from p_j
          in the receiving step of this phase
21      then send a reply message to p_j

30  send inquiring/notifying/reply messages to corresponding destination sets
```

Correctness. Ending phases guarantee correctness, as is stated in the next fact.

Lemma 13.29 GENERIC-GOSSIP *is correct for every communication graph* $G(n, f)$ *and set of schedules* Π.

Proof Integrity and No-Duplicates properties follow directly from the code and the multicast service in synchronous message-passing system. It remains to prove that each processor has heard about all processors. Consider the step just before the first ending phases. If a processor p_i has not heard about some other processor p_j yet, then it sends a last-resort message to p_j in the first ending phase. It is replied to in the second ending phase, unless processor p_j has crashed already. In any case, in the third ending phase, processor p_i either learns the input rumor of p_j or it gets to know that p_j has failed. The fourth ending phase provides an opportunity to receive notifying messages, by all the processors that such messages were sent to by p_i. ■

The choice of communication graph $G(n, f)$, set of schedules Π and termination threshold τ influences however time and message complexities of the specific implementation of Generic Gossip Algorithm. First consider the case when $G(n, f)$ is a communication graph satisfying property $\mathcal{R}(n, f)$ from Definition 13.27, Π contains n random permutations, and $\tau = c \log^2 n$ for sufficiently large positive constant c. Using Theorem 13.28 we get the following result.

Theorem 13.30 *For every* n *and* $f \leq c \cdot n$, *for some constant* $0 \leq c < 1$, *there is a graph* $G(n, f)$ *such that the implementation of the generic gossip scheme with* $G(n, f)$ *as a communication graph and a set* Π *of random permutations completes gossip in expected time* $O(\log^2 n)$ *and with expected message complexity* $O(n \log^2 n)$, *if the number of crashes is at most* f.

Consider a small modification of Generic Gossip scheme: during regular phase every processor p_i sends an inquiring message to the first Δ (instead of one) processors according to permutation π_i, where Δ is a maximum degree of used communication graph $G(n, f)$. Note that it does not influence the asymptotic message complexity, since besides inquiring messages in every regular phase each processor p_i sends Δ graph messages.

Theorem 13.31 *For every* n *there are parameters* $f \leq n-1$ *and* $\tau = O(\log^2 n)$ *and there is a graph* $G(n, f)$ *such that the implementation of the modified Generic Gossip scheme with* $G(n, f)$ *as a communication graph and a set* Π *of random permutations completes gossip in expected time* $O(\log^2 n)$ *and with expected message complexity* $O(n^{1.838})$, *for any number of crashes.*

Since in the above theorem set Π is selected prior the computation, we obtain the following existential deterministic result.

Theorem 13.32 *For every n there are parameters $f \leq n-1$ and $\tau = O(\lg n)$ and there are graph $G(n,f)$ and set of schedules Π such that the implementation of the modified Generic Gossip scheme with $G(n,f)$ as a communication graph and schedules Π completes gossip in time $O(\lg n)$ and with message complexity $O(n^{1.838})$, for any number of crashes.*

Exercises

13.7-1 Design executions showing that there is no relation between Causal Order and Total Order and between Single-Source FIFO and Total Order broadcast services. For simplicity consider two processors and two messages sent.

13.7-2 Does broadcast service satisfying Single-Source FIFO and Causal Order requirements satisfy a Total Order property? Does broadcast service satisfying Single-Source FIFO and Total Order requirements satisfy a Causal Order property? If yes provide a proof, if not show a counterexample.

13.7-3 Show that using reliable Basic Broadcast instead of Basic Broadcast in the implementation of Single-Source FIFO service, then we obtain reliable Single-Source FIFO broadcast.

13.7-4 Prove that the Ordered Broadcast algorithm implements Causal Order service on a top of Single-Source FIFO one.

13.7-5 What is the total number of point-to-point messages sent in the algorithm Ordered-Broadcast in case of k broadcasts?

13.7-6 Estimate the total number of point-to-point messages sent during the execution of Reliable-Causally-Ordered-Broadcast, if it performs k broadcast and there are $f < n$ processor crashes during the execution.

13.7-7 Show an execution of the algorithm Reliable-Causally-Ordered-Broadcast which violates Total Order requirement.

13.7-8 Write a code of the implementation of reliable Sub-Total Order multicast service.

13.7-9 Show that the described method of implementing multicast services on the top of corresponding broadcast services is correct.

13.7-10 Show that the random graph $G(n,f)$ - in which each node selects independently at random $\frac{n}{n-f}\log n$ edges from itself to other processors - satisfies property $\mathcal{R}(n,f)$ from Definition 13.27 and has degree $O(\frac{n}{n-f}\lg n)$ with probability at least $1 - O(1/n)$.

13.7-11 Leader election problem is as follows: all non-faulty processors must elect one non-faulty processor in the same synchronous step. Show that leader election can not be solved faster than gossip problem in synchronous message-passing system with processors crashes.

13.8. Mutual exclusion in shared memory

We now describe the second main model used to describe distributed systems, the shared memory model. To illustrate algorithmic issues in this model we discuss solutions for the mutual exclusion problem.

13.8.1. Shared memory systems

The shared memory is modeled in terms of a collection of *shared variables*, commonly referred to as *registers*. We assume the system contains n processors, $p_0, \ldots, p_{n-1}$, and m registers $R_0, \ldots, R_{m-1}$. Each processor is modeled as a state machine. Each register has a *type*, which specifies:

1. the values it can hold,
2. the operations that can be performed on it,
3. the value (if any) to be returned by each operation, and
4. the new register value resulting from each operation.

Each register can have an initial value.

For example, an integer valued read/write register R can take on all integer values and has operations *read(R,v)* and *write(R,v)*. The read operation returns the value v of the last preceding write, leaving R unchanged. The *write(R,v)* operation has an integer parameter v, returns no value and changes R's value to v. A *configuration* is a vector $C = (q_0, \ldots, q_{n-1}, r_0, \ldots, r_{m-1})$, where q_i is a state of p_i and r_j is a value of register R_j. The *events* are computation steps at the processors where the following happens atomically (indivisibly):

1. p_i chooses a shared variable to access with a specific operation, based on p_i's current state,
2. the specified operation is performed on the shared variable,
3. p_i's state changes based on its transition function, based on its current state and the value returned by the shared memory operation performed.

A finite sequence of configurations and events that begins with an initial configuration is called an *execution*. In the asynchronous shared memory system, an infinite execution is admissible if it has an infinite number of computation steps.

13.8.2. The mutual exclusion problem

In this problem a group of processors need to access a shared resource that cannot be used simultaneously by more than a single processor. The solution needs to have the following two properties. (1) *Mutual exclusion*: Each processor needs to execute a code segment called a *critical section* so that at any given time at most one processor is executing it (i.e., is in the critical section). (2) *Deadlock freedom*: If one or more processors attempt to enter the critical section, then one of them eventually succeeds as long as no processor stays in the critical section forever. These two properties do not provide any individual guarantees to any processor. A stronger property

is (3) *No lockout*: A processor that wishes to enter the critical section eventually succeeds as long as no processor stays in the critical section forever. Original solutions to this problem relied on special synchronisation support such as semaphores and monitors. We will present some of the *distributed solutions* using only ordinary shared variables.

We assume the program of a processor is partitioned into the following sections:

- **Entry / Try**: the code executed in preparation for entering the critical section.
- **Critical**: the code to be protected from concurrent execution.
- **Exit**: the code executed when leaving the critical section.
- **Remainder**: the rest of the code.

A processor cycles through these sections in the order: remainder, entry, critical and exit. A processor that wants to enter the critical section first executes the entry section. After that, if successful, it enters the critical section. The processor releases the critical section by executing the exit section and returning to the remainder section. We assume that a processor may transition any number of times from the remainder to the entry section. Moreover, variables, both shared and local, accessed in the entry and exit section are not accessed in the critical and remainder section. Finally, no processor stays in the critical section forever. An algorithm for a shared memory system solves the mutual exclusion problem with no deadlock (or no lockout) if the following hold:

- **Mutual Exclusion:** In every configuration of every execution at most one processor is in the critical section.
- **No deadlock:** In every admissible execution, if some processor is in the entry section in a configuration, then there is a later configuration in which *some* processor is in the critical section.
- **No lockout:** In every admissible execution, if some processor is in the entry section in a configuration, then there is a later configuration in which *that same* processor is in the critical section.

In the context of mutual exclusion, an execution is *admissible* if for every processor p_i, p_i either takes an infinite number of steps or p_i ends in the remainder section. Moreover, no processor is ever stuck in the exit section (unobstructed exit condition).

13.8.3. Mutual exclusion using powerful primitives

A single bit suffices to guarantee mutual exclusion with no deadlock if a powerful test&set register is used. A test&set variable V is a binary variable which supports two atomic operations, test&set and reset, defined as follows:

test&set(V: memory address) returns binary value:
    $temp \leftarrow V$
    $V \leftarrow 1$
    return ($temp$)
reset(V: memory address):

$V \leftarrow 0$

The test&set operation atomically reads and updates the variable. The reset operation is merely a write. There is a simple mutual exclusion algorithm with no deadlock, which uses one test&set register.

MUTUAL EXCLUSION USING ONE TEST&SET REGISTER

Initially V equals 0

⟨Entry⟩:
1 wait until test&set(V) = 0
⟨Critical Section⟩
⟨Exit⟩:
2 reset(V)
⟨Remainder⟩

Assume that the initial value of V is 0. In the entry section, processor p_i repeatedly tests V until it returns 0. The last such test will assign 1 to V, causing any following test by other processors to return 1, prohibiting any other processor from entering the critical section. In the exit section p_i resets V to 0; another processor waiting in the entry section can now enter the critical section.

Theorem 13.33 *The algorithm using one test &set register provides mutual exclusion without deadlock.*

13.8.4. Mutual exclusion using read/write registers

If a powerful primitive such as test&set is not available, then mutual exclusion must be implemented using only read/write operations.

The bakery algorithm Lamport's bakery algorithm for mutual exclusion is an early, classical example of such an algorithm that uses only shared read/write registers. The algorithm guarantees mutual exclusion and no lockout for n processors using $O(n)$ registers (but the registers may need to store integer values that cannot be bounded ahead of time).

Processors wishing to enter the critical section behave like customers in a bakery. They all get a number and the one with the smallest number in hand is the next one to be "served". Any processor not standing in line has number 0, which is not counted as the smallest number.

The algorithm uses the following shared data structures: *Number* is an array of n integers, holding in its i-th entry the current number of processor p_i. *Choosing* is an array of n boolean values such that *Choosing*[i] is true while p_i is in the process of obtaining its number. Any processor p_i that wants to enter the critical section attempts to choose a number greater than any number of any other processor and writes it into *Number*[i]. To do so, processors read the array *Number* and pick the greatest number read +1 as their own number. Since however several processors might be

reading the array at the same time, symmetry is broken by choosing $(Number[i], i)$ as i's ticket. An ordering on tickets is defined using the lexicographical ordering on pairs. After choosing its ticket, p_i waits until its ticket is minimal: For all other p_j, p_i waits until p_j is not in the process of choosing a number and then compares their tickets. If p_j's ticket is smaller, p_i waits until p_j executes the critical section and leaves it.

BAKERY

Code for processor p_i, $0 \leq i \leq n-1$.
Initially $Number[i] = 0$ and
$Choosing[i] =$ FALSE, for $0 \leq i \leq n-1$

⟨Entry⟩:
1 $Choosing[i] \leftarrow$ TRUE
2 $Number[i] \leftarrow \max(Number[0], \ldots, Number[n-1]) + 1$
3 $Choosing[i] \leftarrow$ FALSE
4 **for** $j \leftarrow 1$ **to** n $(\neq i)$ **do**
5 **wait until** $Choosing[j] =$ FALSE
6 **wait until** $Number[j] = 0$ or $(Number[j], j) > (Number[i], i)$ ⟨Critical Section⟩
⟨Exit⟩:
7 $Number[i] \leftarrow 0$
⟨Remainder⟩

We leave the proofs of the following theorems as Exercises 13.8-2 and 13.8-3.

Theorem 13.34 BAKERY *guarantees mutual exclusion.*

Theorem 13.35 BAKERY *guarantees no lockout.*

A bounded mutual exclusion algorithm for n processors Lamports BAKERY algorithm requires the use of unbounded values. We next present an algorithm that removes this requirement. In this algorithm, first presented by Peterson and Fischer, processors compete pairwise using a two-processor algorithm in a *tournament tree* arrangement. All pairwise competitions are arranged in a complete binary tree. Each processor is assigned to a specific leaf of the tree. At each level, the winner in a given node is allowed to proceed to the next higher level, where it will compete with the winner moving up from the other child of this node (if such a winner exists). The processor that finally wins the competition at the root node is allowed to enter the critical section.

Let $k = \lceil \log n \rceil - 1$. Consider a complete binary tree with 2^k leaves and a total of $2^{k+1} - 1$ nodes. The nodes of the tree are numbered inductively in the following manner: The root is numbered 1; the left child of node numbered m is numbered $2m$ and the right child is numbered $2m + 1$. Hence the leaves of the tree are numbered $2^k, 2^k + 1, \ldots, 2^{k+1} - 1$.

With each node m, three binary shared variables are associated: $Want^m[0]$,

$Want^m[1]$, and $Priority^m$. All variables have an initial value of 0. The algorithm is recursive. The code of the algorithm consists of a procedure NODE$(m, side)$ which is executed when a processor accesses node m, while assuming the role of processor *side*. Each node has a critical section. It includes the entry section at all the nodes on the path from the nodes parent to the root, the original critical section and the exit code on all nodes from the root to the nodes parent. To begin, processor p_i executes the code of node $(2^k + \lfloor i/2 \rfloor, i \mod 2)$.

TOURNAMENT-TREE

```
     procedure NODE(m: integer; side: 0..1)
1    Want^m[side] ← 0
2    wait until (Want^m[1 − side] = 0 or Priority^m = side)
3    Want^m[side] ← 1
4    if Priority^m = 1−side
5       then if Want^m[1 − side] = 1)
6               then goto line 1
7               else wait until Want^m[1 − side] = 0
8    if v = 1
9       then ⟨Critical Section⟩
10      else NODE(⌊m/2⌋, m mod 2)
11      Priority^m = 1− side
12      Want^m[side] ← 0
     end procedure
```

This algorithm uses bounded values and as the next theorem shows, satisfies the mutual exclusion, no lockout properties:

Theorem 13.36 *The tournament tree algorithm guarantees mutual exclusion.*

Proof Consider any execution. We begin at the nodes closest to the leaves of the tree. A processor enters the critical section of this node if it reaches line 9 (it moves up to the next node). Assume we are at a node m that connects to the leaves where p_i and p_j start. Assume that two processors are in the critical section at some point. It follows from the code that then $Want^m[0] = Want^m[1] = 1$ at this point. Assume, without loss of generality that p_i's last write to $Want^m[0]$ before entering the critical section follows p_j's last write to $Want^m[1]$ before entering the critical section. Note that p_i can enter the critical section (of m) either through line 5 or line 6. In both cases p_i reads $Want^m[1] = 0$. However p_i's read of $Want^m[1]$, follows p_j's write to $Want^m[0]$, which by assumption follows p_j's write to $Want^m[1]$. Hence p_i's read of $Want^m[1]$ should return 1, a contradiction.

The claim follows by induction on the levels of the tree. ∎

Theorem 13.37 *The tournament tree algorithm guarantees no lockout.*

Proof Consider any admissible execution. Assume that some processor p_i is starved. Hence from some point on p_i is forever in the entry section. We now show that p_i

cannot be stuck forever in the entry section of a node m. The claim then follows by induction.

Case 1: Suppose p_j executes line 10 setting $Priority^m$ to 0. Then $Priority^m$ equals 0 forever after. Thus p_i passes the test in line 2 and skips line 5. Hence p_i must be waiting in line 6, waiting for $Want^m[1]$ to be 0, which never occurs. Thus p_j is always executing between lines 3 and 11. But since p_j does not stay in the critical section forever, this would mean that p_j is stuck in the entry section forever which is impossible since p_j will execute line 5 and reset $Want^m[1]$ to 0.

Case 2: Suppose p_j never executes line 10 at some later point. Hence p_j must be waiting in line 6 or be in the remainder section. If it is in the entry section, p_j passes the test in line 2 ($Priority^m$ is 1). Hence p_i does not reach line 6. Therefore p_i waits in line 2 with $Want^m[0] = 0$. Hence p_j passes the test in line 6. So p_j cannot be forever in the entry section. If p_j is forever in the remainder section $Want^m[1]$ equals 0 henceforth. So p_i cannot be stuck at line 2, 5 or 6, a contradiction.

The claim follows by induction on the levels of the tree. ■

Lower bound on the number of read/write registers So far, all deadlock-free mutual exclusion algorithms presented require the use of at least n shared variables, where n is the number of processors. Since it was possible to develop an algorithm that uses only bounded values, the question arises whether there is a way of reducing the number of shared variables used. Burns and Lynch first showed that any deadlock-free mutual exclusion algorithm using only shared read/write registers must use at least n shared variables, regardless of their size. The proof of this theorem allows the variables to be multi-writer variables. This means that each processor is allowed to write to each variable. Note that if the variables are single writer, that the theorem is obvious since each processor needs to write something to a (separate) variable before entering the critical section. Otherwise a processor could enter the critical section without any other processor knowing, allowing another processor to enter the critical section concurrently, a contradiction to the mutual exclusion property.

The proof by Burns and Lynch introduces a new proof technique, a ***covering argument:*** Given any no deadlock mutual exclusion algorithm A, it shows that there is some reachable configuration of A in which each of the n processors is about to write to a ***distinct*** shared variable. This is called a ***covering*** of the shared variables. The existence of such a configuration can be shown using induction and it exploits the fact that any processor before entering the critical section, must write to at least one shared variable. The proof constructs a covering of all shared variables. A processor then enters the critical section. Immediately thereafter the covering writes are released so that no processor can detect the processor in the critical section. Another processor now concurrently enters the critical section, a contradiction.

Theorem 13.38 *Any no deadlock mutual exclusion algorithm using only read/write registers must use at least n shared variables.*

13.8.5. Lamport's fast mutual exclusion algorithm

In all mutual exclusion algorithms presented so far, the number of steps taken by processors before entering the critical section depends on n, the number of processors even in the absence of contention (where multiple processors attempt to concurrently enter the critical section), when a single processor is the only processor in the entry section. In most real systems however, the expected contention is usually much smaller than n.

A mutual exclusion algorithm is said to be ***fast*** if a processor enters the critical section within a constant number of steps when it is the only processor trying to enter the critical section. Note that a fast algorithm requires the use of multi-writer, multi-reader shared variables. If only single writer variables are used, a processor would have to read at least n variables.

Such a fast mutual exclusion algorithm is presented by Lamport.

FAST-MUTUAL-EXCLUSION

Code for processor p_i, $0 \leq i \leq n-1$. Initially *Fast-Lock* and *Slow-Lock* are 0, and *Want*[i] is false for all i, $0 \leq i \leq n-1$

```
    ⟨ Entry ⟩:
 1  Want[i] ← TRUE
 2  Fast-Lock ← i
 3  if Slow-Lock ≠ 0
 4     then Want[i] ← FALSE
 5     WAIT UNTIL Slow-Lock =0
 6     goto 1
 7  Slow-Lock ← i
 8  if Fast-Lock ≠ i
 9     then Want[i] ← FALSE
10          for all j, wait until Want[j] = FALSE
11          if Slow-Lock ≠ i
12             then wait until Slow-Lock = 0
13          goto 1
    ⟨Critical Section⟩
    ⟨Exit⟩:
14  Slow-Lock ← 0
15  Want[i] ← false
    ⟨Remainder⟩
```

Lamport's algorithm is based on the correct combination of two mechanisms, one for allowing fast entry when no contention is detected, and the other for providing deadlock freedom in the case of contention. Two variables, *Fast-Lock* and *Slow-Lock* are used for controlling access when there is no contention. In addition, each processor p_i has a boolean variable *Want*[i] whose value is true if p_i is interested in

entering the critical section and false otherwise. A processor can enter the critical section by either finding *Fast-Lock* $= i$ - in this case it enters the critical section on the *fast path* - or by finding *Slow-Lock* $= i$ in which case it enters the critical section along the *slow path*.

Consider the case where no processor is in the critical section or in the entry section. In this case, *Slow-Lock* is 0 and all *Want* entries are 0. Once p_i now enters the entry section, it sets *Want*$[i]$ to 1 and *Fast-Lock* to i. Then it checks *Slow-Lock* which is 0. then it checks *Fast-Lock* again and since no other processor is in the entry section it reads i and enters the critical section along the fast path with three writes and two reads.

If *Fast-Lock* $\neq i$ then p_i waits until all *Want* flags are reset. After some processor executes the for loop in line 10, the value of *Slow-Lock* remains unchanged until some processor leaving the critical section resets it. Hence at most one processor p_j may find *Slow-Lock*$= j$ and this processor enters the critical section along the slow path. Note that the Lamport's Fast Mutual Exclusion algorithm does not guarantee lockout freedom.

Theorem 13.39 *Algorithm* FAST-MUTUAL-EXCLUSION *guarantees mutual exclusion without deadlock.*

Exercises

13.8-1 An algorithm solves the *2-mutual exclusion* problem if at any time at most two processors are in the critical section. Present an algorithm for solving the 2-mutual exclusion problem using test & set registers.

13.8-2 Prove that bakery algorithm satisfies the mutual exclusion property.

13.8-3 Prove that bakery algorithm provides no lockout.

13.8-4 Isolate a bounded mutual exclusion algorithm with no lockout for two processors from the tournament tree algorithm. Show that your algorithm has the mutual exclusion property. Show that it has the no lockout property.

13.8-5 Prove that algorithm FAST-MUTUAL-EXCLUSION has the mutual exclusion property.

13.8-6 Prove that algorithm FAST-MUTUAL-EXCLUSION has the no deadlock property.

13.8-7 Show that algorithm FAST-MUTUAL-EXCLUSION does not satisfy the no lockout property, i.e. construct an execution in which a processor is locked out of the critical section.

13.8-8 Construct an execution of algorithm FAST-MUTUAL-EXCLUSION in which two processors are in the entry section and both read at least $\Omega(n)$ variables before entering the critical section.

Problems

***13-1 Number of messages of the algorithm* Flood**

Prove that the algorithm FLOOD sends $O(e)$ messages in any execution, given a

graph G with n vertices and e edges. What is the exact number of messages as a function of the number of vertices and edges in the graph?

13-2 Leader election in a ring

Assume that messages can only be sent in CW direction, and design an asynchronous algorithm for leader election on a ring that has $O(n \lg n)$ message complexity. *Hint.* Let processors work in phases. Each processor begins in the ***active mode*** with a ***value*** equal to the identifier of the processor, and under certain conditions can enter the ***relay mode,*** where it just relays messages. An active processor waits for messages from two active processors, and then inspects the values sent by the processors, and decides whether to become the leader, remain active and adopt one of the values, or start relaying. Determine how the decisions should be made so as to ensure that if there are three or more active processors, then at least one will remain active; and no matter what values active processors have in a phase, at most half of them will still be active in the next phase.

13-3 Validity condition in asynchronous systems

Show that the validity condition is equivalent to requiring that every nonfaulty processor decision be the input of some processor.

13-4 Single source consensus

An alternative version of the consensus problem requires that the input value of one distinguished processor (the *general*) be distributed to all the other processors (the *lieutenants*). This problem is also called ***single source consensus problem.*** The conditions that need to be satisfied are:

- **Termination**: Every nonfaulty lieutenant must eventually decide,
- **Agreement**: All the nonfaulty lieutenants must have the same decision,
- **Validity**: If the general is nonfaulty, then the common decision value is the general's input.

So if the general is faulty, then the nonfaulty processors need not decide on the general's input, but they must still agree with each other. Consider the synchronous message passing system with Byzantine faults. Show how to transform a solution to the consensus problem (in Subsection 13.4.5) into a solution to the general's problem and vice versa. What are the message and round overheads of your transformation?

13-5 Bank transactions

Imagine that there are n banks that are interconnected. Each bank i starts with an amount of money m_i. Banks do not remember the initial amount of money. Banks keep on transferring money among themselves by sending messages of type <10> that represent the value of a transfer. At some point of time a bank decides to find the total amount of money in the system. Design an algorithm for calculating $m_1 + \cdots + m_n$ that does not stop monetary transactions.

Chapter Notes

The definition of the distributed systems presented in the chapter are derived from the book by Attiya and Welch [3]. The model of distributed computation, for message passing systems without failures, was proposed by Attiya, Dwork, Lynch and

Stockmeyer [2].

Modeling the processors in the distributed systems in terms of automata follows the paper of Lynch and Fisher [16].

The concept of the execution sequences is based on the papers of Fischer, Gries, Lamport and Owicki [16, 17, 18].

The definition of the asynchronous systems reflects the presentation in the papers of Awerbuch [4], and Peterson and Fischer [**?**].

The algorithm SPANNING-TREE-BROADCAST is presented after the paper due to Segall [21].

The leader election algorithm BULLY was proposed by Hector Garcia-Molina in 1982 [9]. The asymptotic optimality of this algorithm was proved by Burns [**?**].

The *two generals problem* is presented as in the book of Gray [**?**].

The *consensus problem* was first studied by Lamport, Pease, and Shostak [13, 19]. They proved that the Byzantine consensus problem is unsolvable if $n \leq 3f$ [19].

One of the basic results in the theory of asynchronous systems is that the consensus problem is not solvable even if we have reliable communication systems, and one single faulty processor which fails by crashing. This result was first shown in a breakthrough paper by Fischer, Lynch and Paterson [8].

The algorithm CONSENSUS-WITH-CRASH-FAILURES is based on the paper of Dolev and Strong [7].

Berman and Garay [5] proposed an algorithm for the solution of the Byzantine consensus problem for the case $n > 4f$. Their algorithm needs $2(f + 1)$ rounds.

The bakery algorithm [11] for mutual exclusion using only shared read/write registers to solve mutual exclusion is due to Lamport [11]. This algorithm requires arbitrary large values. This requirement is removed by Peterson and Fischer [**?**]. After this Burns and Lynch proved that any deadlock-free mutual exclusion algorithm using only shared read/write registers must use at least n shared variables, regardless of their size [6].

The algorithm FAST-MUTUAL-EXCLUSION is presented by Lamport [12].

The source of the problems 13-3, 13-4, 13-5 is the book of Attiya and Welch [3].

Important textbooks on distributed algorithms include the monumental volume by Nancy Lynch [15] published in 1997, the book published by Gerard Tel [22] in 2000, and the book by Attiya and Welch [3]. Also of interest is the monograph by Claudia Leopold [14] published in 2001, and the book by Nicola Santoro [20], which appeared in 2006.

A recent book on the distributed systems is due to A. D. Kshemkalyani and M. [10].

Finally, several important open problems in distributed computing can be found in a recent paper of Aspnes et al. [1].

Bibliography

[1] J. Aspnes, C. Busch, S. Dolev, F. Panagiota, C. Georgiou, A. Shvartsman, P. Spirakis, R. Wattenhofer. Eight open problems in distributed computing. *Bulletin of European Association of Theoretical Computer Science of EATCS*, 90:109–126, 2006. 637

[2] H. Attiya, C. Dwork, N. A. Lynch, L. J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41:122–142, 1994. 637

[3] H. Attiya, J. Welch. *Distributed Computing, Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998. 636, 637

[4] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. 637

[5] P. Berman, J. Garay. Cloture votes: $n/4$-resilient distributed consensus in $t+1$ rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993. 637

[6] J. E. Burns, N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993. 637

[7] D. Dolev, R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. 637

[8] M. J. Fischer, N. A. Lynch, M. S. Paterson. Impossibility of distributed consensus with one faulty proces. *Journal of the ACM*, 32(2):374–382, 1985. 637

[9] H. Garcia-Molina, J. Seiferas. Elections in a distributed computing systems. *IEEE Transactions on Computers*, C-31(1):47–59, 1982. 637

[10] A. D. Kshemkalyani, M. Singhal. *Distributed Computing.* Cambridge University Press, 2008. 637

[11] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 18(8):453–455, 1974. 637

[12] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computers*, 5(1):1–11, 1987. 637

[13] L. Lamport, R. Shostak M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. 637

[14] C. Leopold. *Parallel and Distributed Computing.* Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Copyrights 2001. 637

[15] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufman Publisher, 2001 (5th edition). 637

[16] N. A. Lynch, M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1):17–43, 1981. 637

[17] S. Owicki, D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976. 637

[18] S. Owicki, L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. 637

[19] M. Pease, R. Shostak L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. 637

[20] N. Santoro. *Design and Analysis of Distributed Algorithms*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2006. 637

[21] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, 1983. 637

[22] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000 (2nd edition). 637

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *exe* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.