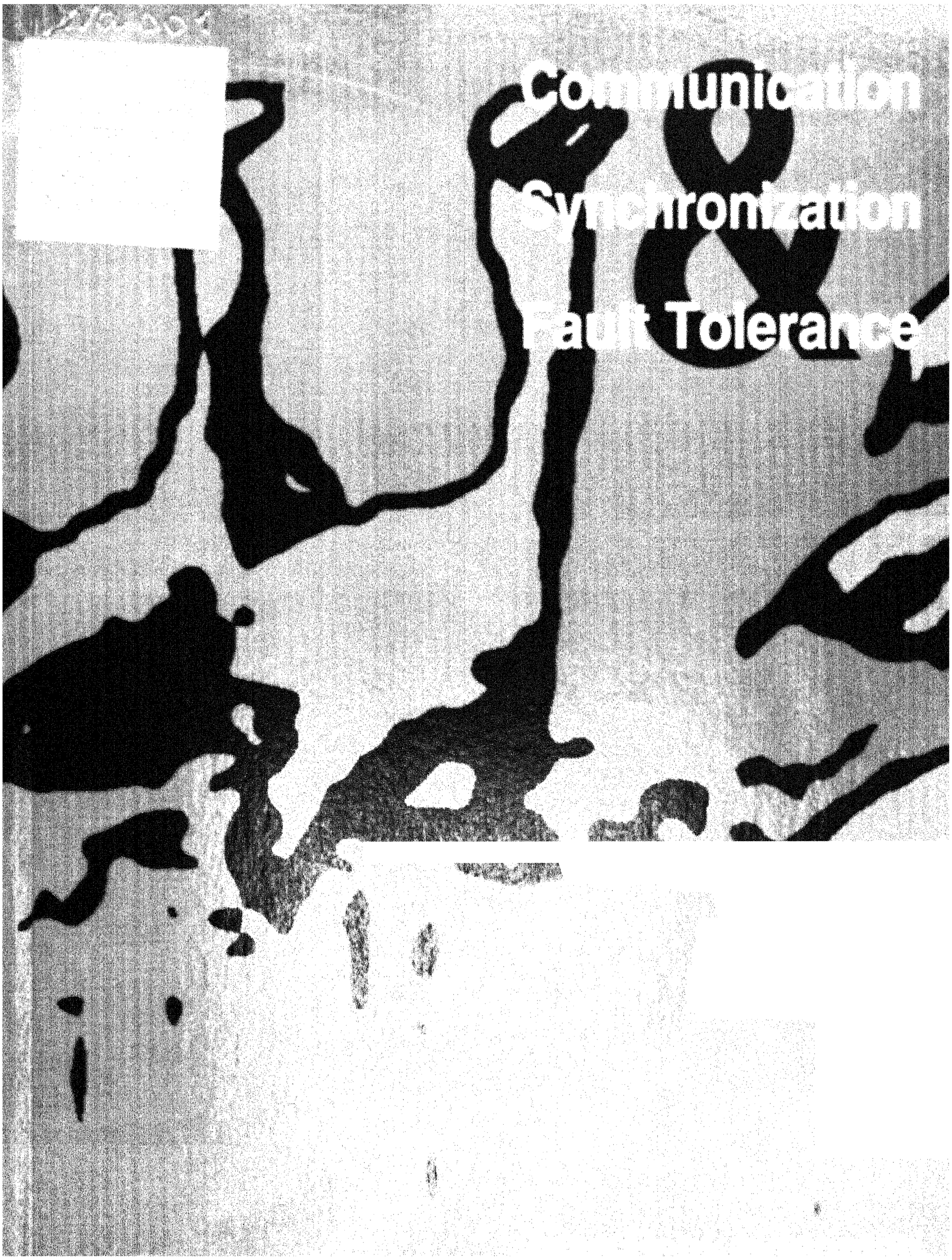
Communication
Synchronization
Fault Tolerance

Communication, Synchronization & Fault Tolerance

Communication, Synchronization & Fault Tolerance

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr. P. W. M. de Meijer
ten overstaan van een door het college
van dekanen ingestelde commissie
in het openbaar te verdedigen in de Aula der Universiteit
op vrijdag 28 juni 1996 te 11.30 uur

door

Jaap Henk Hoepman
geboren te Groningen

Promotor: Prof. dr. ir. P. M. B. Vitányi

Promotie-commissie:
Prof. dr. J. van Leeuwen
Prof. dr. S. J. Mullender
Prof. dr. W. H. Hesselink
Prof. dr. M. L. Kersten
Dr. P. van Emde Boas
Dr. L. Torenvliet
J. A. Garay, PhD

Faculteit Wiskunde en Informatica
Universiteit van Amsterdam

Research in this thesis was partially supported by the Dutch organization for scientific research NWO under NFI project Aladdin, project number NF 62-376.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

aan Heitie

Broeva, Haro

UDERZO & GOSCINNY
Asterix & Obelix - De Lauwerkrans van Cæsar

Voorwoord

Het is een bizar idee om nu, halverwege april, dit voorwoord te schrijven en zeker te weten dat eind juni, als ik dit proefschrift verdedig, de wereld er totaal anders uit zal zien. Er staat deze maanden van alles te gebeuren: leuke dingen, trieste dingen; ik weet alleen niet hoe, en wanneer. Het is echter niet de plaats noch het juiste moment om daar verder over uit te weiden. Zeker is slechts één ding: 1996 zal voor altijd in mijn geheugen gegrift staan.

Allereerst wil ik mijn promotor, Paul Vitányi, bedanken voor het aan mij toevertrouwen van een OiO plaats. Ik heb later begrepen dat uiteindelijk mijn bijvak psychologie de doorslag heeft gegeven: je weet inderdaad nooit hoe een koe een haas vangt. Paul, je hebt mij gestimuleerd om veel artikelen te schrijven en vaak naar conferenties te gaan, om zo al doende te leren waar het in de wetenschap om draait: kennis én contacten. Ik weet dat je het jammer vindt dat ik geen post-doc plaats in het buitenland (lees Amerika) ambiëer. Er is nu eenmaal iemand die ik nog belangrijker acht dan een grenzeloze zoektocht naar ware kennis.

Natuurlijk dank ik ook de overige leden van de promotie-commissie voor de tijd die ze gestoken hebben in het beoordelen van dit proefschrift. Vooral omdat het onderwerp nu eenmaal niet voor ieder van jullie even bekend terrein was.

Het vinden van huisvesting is het eeuwige probleem in een stad als Amsterdam. Forenzen vanuit Groningen is geen optie. Gelukkig kon ik de eerste maanden een kamer onderhuren, en toen dat ophield kon ik ook terecht bij een neef. Bedankt daarvoor, Barteld.

De eerste dagen van een nieuwe baan zijn altijd vreemd. Bij deze

dank ik mijn eerste kamergenoten John "Go" Tromp en Hyonyong "Bob" Shin die mij wegwijs hebben gemaakt op het CWI.

Groetjes ook aan de andere drie musketiers van het Aladdin project: Pascale van der Put, Dick Alstein en Bryan Olivier. We hebben de eerste jaren vrij intensief samengewerkt; het is jammer dat tegen het einde van het project we elkaar steeds minder zagen.

Also a big hi to all the visitors to the Aladdin project: Ted Herman, K. Vidyasankar, Amos Israeli, Shlomo Moran, Alessandro "pedantics" Panconesi, Jörg Keller, to name but a few. καλιμερα[1] to the "singers": Philippas and Marina. Thanks to Juan Garay for pointing out those religious things. And cheers to Mark. We may have had some difficulties writing that paper. But I sure will buy you a beer and play pool, anytime, any conference.

Op mijn kamer met hospita aan de Middenweg (jullie moesten eens weten hoe riant ik gewoond heb tijdens mij studententijd ...) mocht ik niet koken. Gelukkig heeft de UvA een prima mensa, en dankzij mijn kamergenoten Jeroen, Peter, en ook jij, Herman, at ik daar eigenlijk zelden alleen. Het was sowieso wel gezellig, die laatste jaren op M231 en M232. Also with you, André. En jij natuurlijk, Harry. We hadden allebei een geheimpje, en we hebben samen aan het idee kunnen wennen. Kom niet in M232: je krijgt er kinderen van.

Mijn vrienden: Frits, Miriam, Lucie en Robert. Je kunt tenslotte maar twee paranimfen hebben.

Heitie en Memmie. Het is rot. Er zijn gewoon geen woorden voor. En dat hoeft ook eigenlijk niet: we weten zo ook wel wat we aan elkaar hebben. Ik ben blij dat ik dit wel heb kunnen doen, laten we maar zeggen in plaats van jullie.

Lieve Lidewij. Je zult mij niet horen zeggen dat het schrijven van dit proefschrift een zware bevalling is geweest. Jouw bevalling van ons kind zal zeker vele malen zwaarder zijn. Daar staat dan tegenover dat je er later ook veel meer aan hebt ... zo'n kind. Schat, dat s-je wegpoetsen heeft mij 4 lange jaren in Amsterdam gehouden, terwijl jij het daar in Groningen voornamelijk alleen hebt moeten rooien. Bedankt voor je liefde, geduld en steun. Als er stelingen in dit proefschrift zouden staan, was de eerste: "It takes two to write a thesis". Nu kom ik terug, en zijn we eindelijk met z'n vijven. En voor jou, kleine — net op tijd: Papa is thuis!

Groningen, april 1996.

---

[1] "Zij zijn groot en ik is klein, en dat vind ik niet eerlijk! Oh nee!"

Contents

ARTHUR: *A duck.*
BEDEMIR: *Exactly! So, logically . . .,*
VILLAGER: *If . . . she . . . weighs the same as a duck, she's made of wood.*
BEDEMIR: *And therefore . . .?*
VILLAGER: *A witch!!*

MONTY PYTHON
Monty Python and the Holy Grail

1 Introduction

Computers have pervaded, indeed defined, modern society. Networks of computers are all around us, and they grow larger every day. Most offices, for instance, will have a local-area network these days; you don't need to be a computer freak anymore to know about the Internet and to actually have used electronic mail; and there is a cash point just around the corner, waiting to dispense money at your request. These are just three examples of, essentially, a distributed system.

Broadly speaking, a distributed system is any group of computers that depend on each other to get their work done. Obviously, these computers — which I will call processes further on — must communicate with each other. This is achieved by exchanging digital messages over a communication network. The processes may be cash points contacting the central bank to check your balance, they may be office computers waiting until the laser printer on this floor is ready to print a document, or they may be the computers involved in forwarding an Email message you have just sent to me. The network can consist of simple telephone wires, high-speed local-area networks, or trans-atlantic, wireless, satellite connections.

Distributed systems are heterogeneous: their processes may be overloaded central servers, simple terminals like cash points, or remote printers, interconnected by fast fibre-optic links as well as slow modem connections. The processes exhibit a vast difference in speed, and messages sent over certain links may take forever to arrive.

The key difficulty in designing a distributed system is that a single process cannot, in general, observe the state of all other processes in the system simultaneously. Thus the global state of the system, also called the

system configuration, lies out of immediate reach of the individual processes. Certainly, the processes can communicate with each other to collect some view of the system configuration. However, this will not give a globally consistent configuration that actually occurred at any real point in time. To appreciate this problem, consider the following example. Suppose you have two bank accounts, one holding £20 on bank *A*, and another holding £10 on bank *B*. Now, to balance the accounts you send bank *A* a request to transfer £5 to bank *B*.

Bank *A* and *B* have at one point agreed to transfer money in the following way. Bank *A* transfers money to *B* by withdrawing money from an account and sending a message to *B* specifying the amount and the account to deposit it on. On receipt of this message, bank *B* will deposit the money on this account, and the transfer is complete. This clearly takes some time (and may take considerable time, depending on the banks in question). Agreement on a method to transfer money is what constitutes a *protocol*. Without such an agreement, no transfer of money would be possible.

The next day you call both banks to check the state of affairs, just as they are engaged in transferring the money. Suppose you call bank *A* first, after it withdrew £5 from your account, and then call bank *B*, before it deposited the £5. You would find yourself owning only £25. Calling bank *B* first clearly does not help. Also, if the banks decide to change protocol and to deposit the money on the designated account first, before sending an acknowledgement back to the first bank allowing it to withdraw the deposited amount, you could find yourself owning £35. This would happen, for instance, if you called bank *B* after they deposited £5 on your account, while you called bank *A* before they received the acknowledgement to withdraw the £5 from your account.

The reader may be fooled into believing the above problem is an idiosyncrasy of the above protocol. This is not the case, however. In fact, the core problem in any (asynchronous) distributed system is that the observed order of events may differ among different observers. Indeed, the situation is quite similar to the observations made by Einstein [Ein16] regarding simultaneity of events. According to his theory of relativity, events that appear simultaneous to one observer *cannot* be observed simultaneously by an observer in a (suitably) different frame of reference. Hence, simultaneity is truly in the eye of the beholder.

In short, the processes in a distributed system have to maintain a global objective through local decisions based on local, partial, information. This makes the design of distributed systems an intricate matter.

The previous example also introduced two key features of any dis-

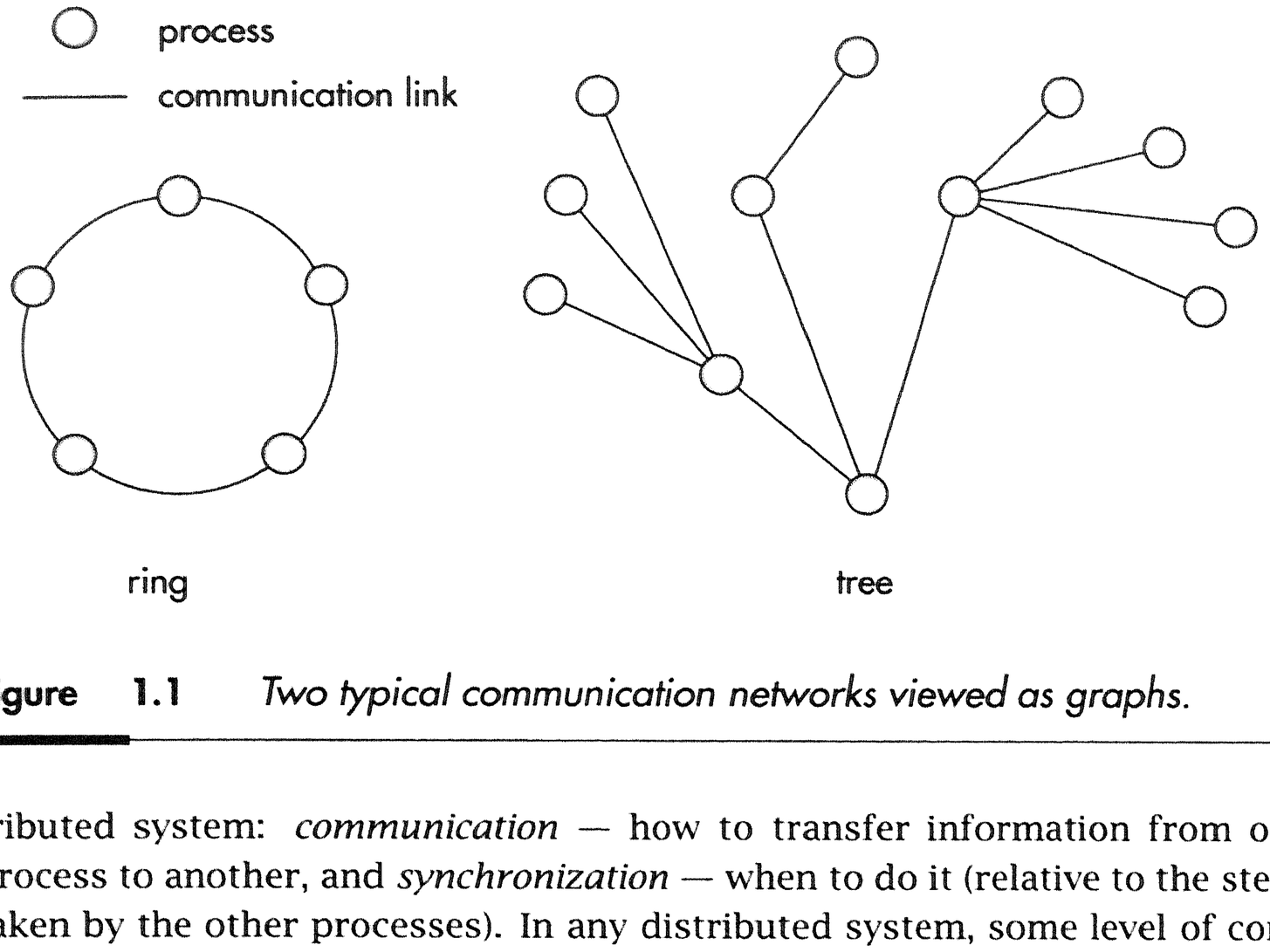

Figure 1.1 *Two typical communication networks viewed as graphs.*

tributed system: *communication* — how to transfer information from one process to another, and *synchronization* — when to do it (relative to the steps taken by the other processes). In any distributed system, some level of communication and synchronization will already be built-in by the hardware. Additional functionality will have to be implemented by software. The initial provisions for communication and synchronization supported by the hardware is called the *model* of the distributed system.

1.1 Communication & synchronization

Let us review some common provisions for communication and synchronization found in current distributed systems.

Unless the system is very small, processes will not be able to communicate directly with all other processes in the system. A process can only exchange information with neighbours to which it is directly connected: information destined for other processes will have to be passed on by the intermediate nodes (see the routing problem described below). The interconnection pattern can be described by a *graph*, where the nodes are processes and the edges are communication links (see Fig. 1.1). Processes that share a communication link are neighbours. Usually, this graph is *undirected*, and neighbours can directly communicate to each other (like two people talking with each other through a telephone). On the other hand, if the graph is *directed*, one

neighbour may be able to communicate information to the other neighbour, but not the other way around. This is similar to the way pagers work. Processes in a distributed system can exchange information either by sending (or broadcasting) *messages*, or by reading and writing *shared memory*.

If communicating processes wait until both parties have completed their part in the information exchange, or if processes have access to some global clock, the system is called *synchronous*. If not, the system is called *asynchronous*. Usually, we view a synchronous system to operate in lock-step mode, executing rounds in unison. At the start of every round processes receive all incoming messages, then perform some local computations, and finishing the round by sending out messages. These messages arrive before the start of the next round.

Asynchronous communication is very much like sending letters by post over a long distance, say to another continent. It may take weeks before you receive a reply. And if you want to conduct some kind of regular and frequent correspondence, you will have written another letter before the previous letter even arrived. In such cases it is not uncommon to receive a letter asking questions answered by the letter you already sent a few weeks before, simply because your letter had not arrived at its destination at the time. In fact, this type of asynchrony is the root of the two-banks problem sketched above. Asynchronous communication is suitable for systems that comprise both very slow and very fast processes, because the fast processes are not blocked when communicating with the slow processes.

Synchronous communication is similar to two people communicating by phone, in the sense that the information transfer is instantaneous, and that (usually) people do not both talk at the same time. So messages cannot 'cross' each other as in the asynchronous case, and whatever is said, is a response on everything said up till now.

Another means to exchange information is by storing it in shared memory, where it can be read by several other processes. This is essentially an asynchronous mode of communication, because the readers and the writers do not usually wait for one another. There is an important difference between exchanging messages and storing data in shared memory. In an asynchronous system, if a process sends a message to another process that is not ready to receive it, the message is usually stored in some message buffer at the receiving process for later delivery. If the sending process is fast, many new messages may get appended at the end of the buffer, but when the receiving process is ready for it, the first message sent will be the first message delivered. So when sending messages, all information in those messages will be received. When

using shared memory, however, a fast writing process can overwrite data it wrote earlier with new information, before the old data was even read by the reading process.

1.2 Fault tolerance

Nobody's perfect, not even a computer. Just like any other piece of equipment, a computer has a limited average life-time. Sooner or later it will stop functioning. When distributed systems grew larger, people soon realized that the chance of any single process failing increases with the total number of processes in the system. Careless design of a distributed system can make all processes vulnerable to failure of a single process. Indeed (L. Lamport, cf. [SL95], page 120):

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.[1]

If a single-computer system fails, there is nothing much you can do about it. However, if the system contains many independent processes that can take over each others tasks, it is not entirely unreasonable to expect that if only a small portion of the processes fail, the system as a whole remains functional. In fact, this form of redundancy is very desirable because a large number of people are likely to depend on a large distributed system. To put it another way: the more people depend on a distributed system, the more likely it becomes that one or two processes of that system fail.

Consider again the network of cash points contacting a central bank to check your balance before actually dispensing cash. Stupid implementation of this system using a single computer as a server storing all the balances, with all cash points contacting this single server, will result in a failure-prone system. If the single server fails, no cash point can check balances. Moreover, the single server gets all the load and is therefore more liable to fail. If the server fails, the cash points have no other access to the accounts, and therefore cannot and will not dispense money. This may inconvenience a lot of people. Fault tolerance becomes even more important in life-critical applications, like air-traffic control systems. Under no circumstances should failure of a single process result in a plane crash.

[1] Apparently, Lamport never made this statement in writing. This would explain why essentially the same quote appears in two, quite different, phrasings in [Mul89] and [Mul93].

We distinguish two types of failures: *process failures*, and *memory failures*. A process failure is a permanent failure, due to power-failures, circuit blows, etc. In case of power-failures, the process will stop whatever it was doing and will not perform any work until the mains have been reconnected or the fuse has been replaced. This type of failure is called a *crash failure*. System crashes due to misbehaving processes that reset an entire machine and killing all running processes can also be regarded as crash failures.

If only part of the process circuitry is corrupted, the process may still be running, but its output can no longer be trusted. In this case it is best to hope for the best but to prepare for the worst, and to assume that such a failed process actually tries to do its best to disrupt the task performed by the other processes. Such failures are called *Byzantine failures*. Crash failures are less severe than Byzantine failures. On strictly asynchronous systems a crashed process cannot be distinguished from a very slow process. Therefore, crash failures cannot be detected.

Memory failures cover all changes to volatile memory (RAM) as a result of transient system failures like short power glitches or noisy communication lines. Such failures change some or all of the data processed by a process, but leave the process itself in working order[2]. The program may, for instance, be stored in ROM, or the process may be implemented as a hardware circuit. Memory failures are not considered permanent. Therefore, next to masking memory failures by replication, they can also be repaired if one allows some time for this correction. This is achieved by so called *self-stabilizing* systems.

A self-stabilizing system will, when started in an arbitrary initial configuration, eventually reach a configuration satisfying its specification. Once such a *legitimate configuration* is reached, the system will remain in legitimate configurations forever (see Fig. 1.2). In other words, if after some transient error the states of some processes are corrupted, all processes will cooperate to repair the fault. Self-stabilizing systems are quite remarkable as no matter how bad the states are corrupted, after some finite number of steps the processes will reach a good, legitimate, configuration again. The point is that if transient errors are infrequent, and the stabilization time is short, the system will be in a legitimate configuration most of the time.

---

[2] I can personally vouch for very weird errors of this kind actually occurring.

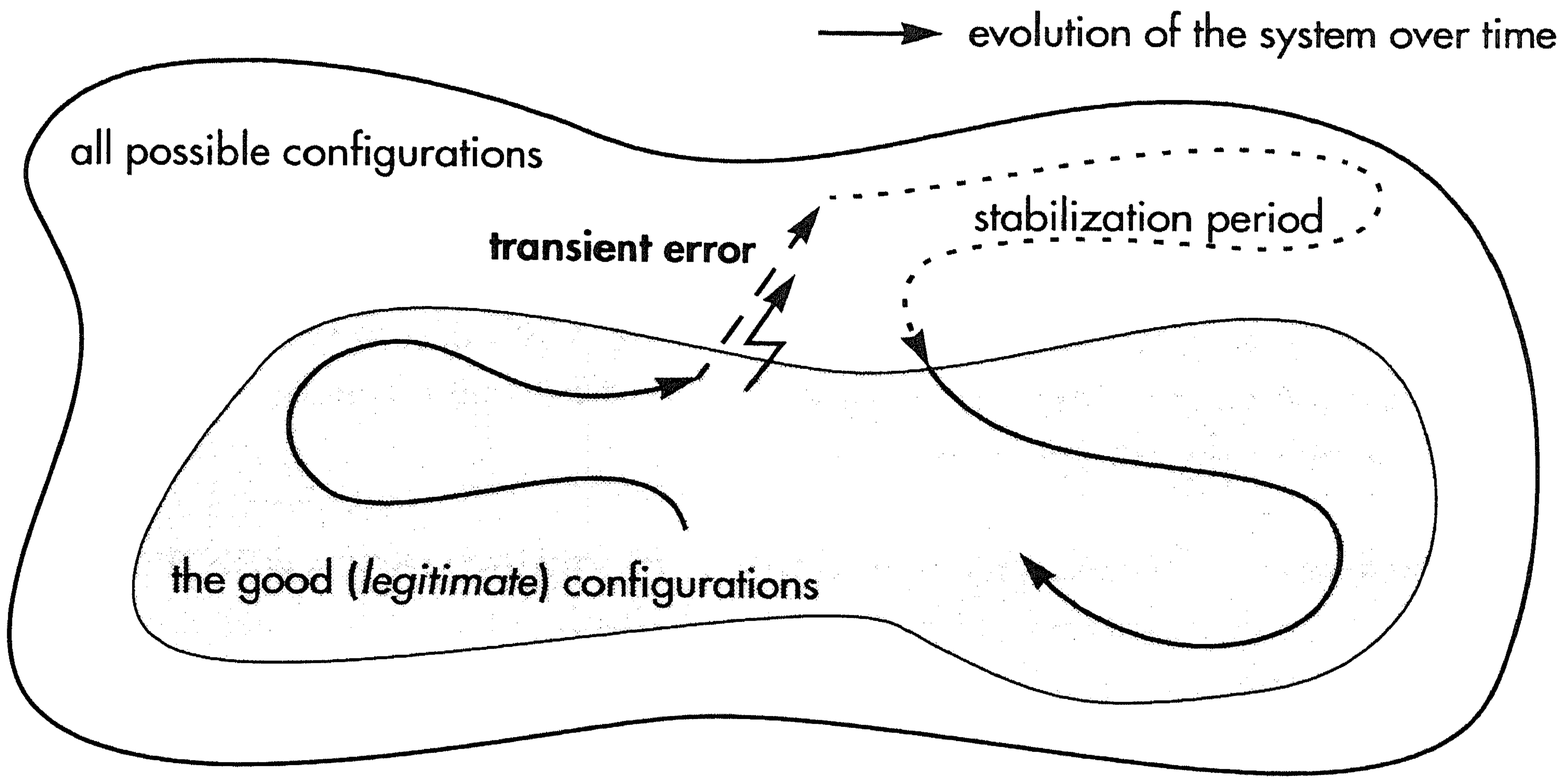

Figure 1.2 *A self-stabilizing distributed system.*

1.3 Typical problems

One can identify four typical problems studied in distributed computing that are specifically concerned with the fundamental issues of communication, synchronization and fault tolerance. First, the *routing problem* deals with providing full point-to-point communication in systems were not all processes are immediately connected. This is clearly a *communication* problem. Second, the *mutual exclusion* problem of how to implement exclusive a access to some critical shared resource, is a matter of proper *synchronization*. *Wait-free* constructions of shared memory objects are *fault-tolerant* in the face of *crash* failures. They are also closely related to the study of communication and synchronization. Finally, the *agreement* problem is usually studied when *Byzantine* failures are considered. I will describe these problems in more detail in the next few paragraphs.

Routing In most networks, not all processes can communicate directly with each other. If a process wishes to send a message to another process which is not an immediate neighbour, this message must be sent to a neighbour with the request to forward it to the ultimate destination. Every process checks whether each incoming message is destined to itself. If not, it forwards the

message to another neighbour on the route to its final destination.

Selection of the neighbour responsible for forwarding the message is determined by a *local routing function*. All local routing functions together constitute a *routing scheme*, which for any sender and any destination determines the route traversed by the message. A good routing scheme will route messages over the shortest path from sender to destination. Because sending a message over an edge always takes some time, such a *shortest-path* routing scheme minimizes the time that elapses before a message reaches its destination.

Mutual exclusion Research in distributed computing started with the study of mutual exclusion protocols in the sixties [Dij65].

A mutual exclusion protocol must ensure that no two processes can simultaneously access some critical resource. A typical example of such a critical resource is a laser printer shared in some local-area network. If two or more computers were allowed to simultaneously print some document, the contents of these documents would appear muddled together in print. A process that needs access to the resource must first request the privilege to do so using a mutual exclusion protocol. Clearly, the privilege must be passed fairly among the processes that request it. In other words, if someone requests to print a document, eventually this user should be allowed to do so. Unfair mutual exclusion might prevent one user from ever printing a document.

Wait-free constructions Suppose some processes in an asynchronous system communicate by reading and writing some shared variables. Intuitively, reads should return the value written by the most recent write. However, in asynchronous systems, operations on shared memory take some time to complete, and operations performed by different processes may overlap in time. The question is to properly define what is the most recent write of a read that overlaps with several writes to the same shared variable.

When processes communicate using shared memory, we usually assume that reads and writes of shared memory happen *atomically*. This means that, in the original Greek sense of the word, these actions happen indivisibly. They are assumed to take effect at one single point in time (called the *serialization time*); it is moreover assumed that no other action takes effect at that same time. If we consider a real asynchronous system communicating through shared memory, and record the invocation and response times of each individual action, all these action-intervals can be shrunk to a point within this interval at which time the action actually took place. This defini-

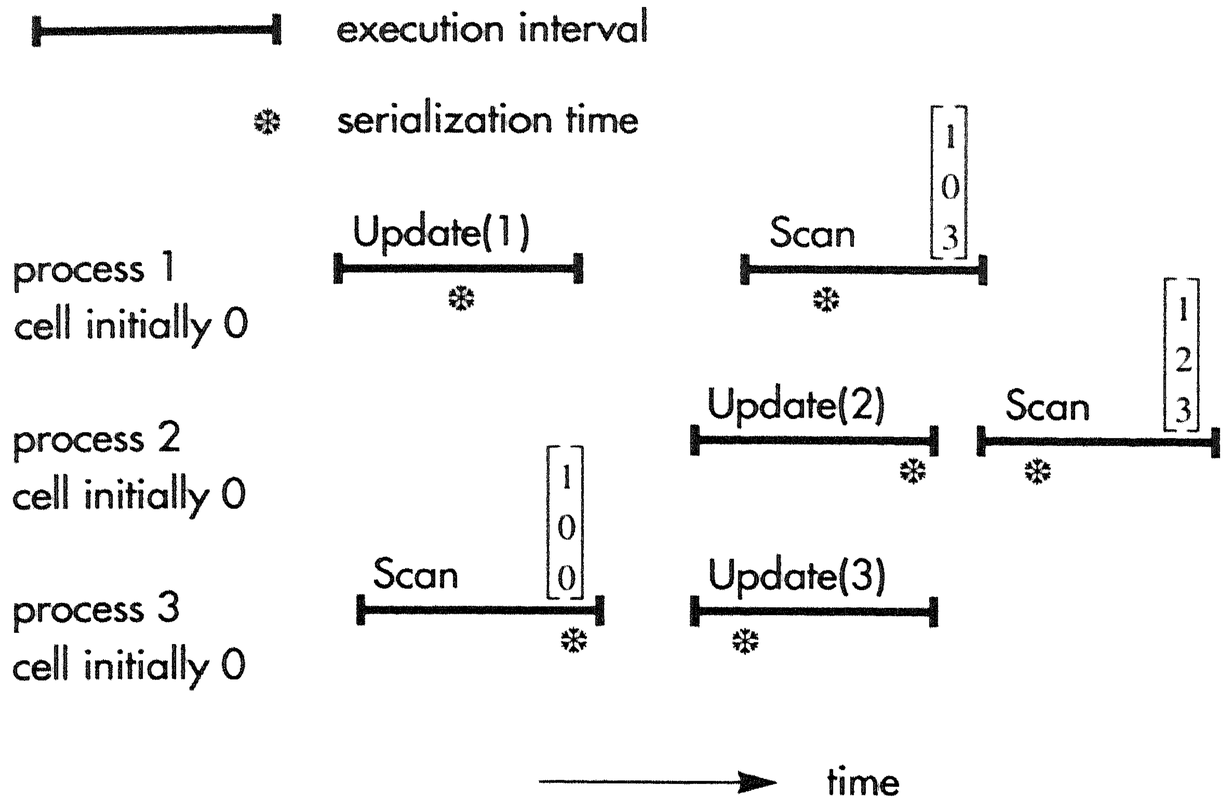

Figure 1.3 *Example execution over the snapshot object.*

tion matches our intuitive understanding of a shared variable for actions that do not overlap each other. It also guarantees, at all times, that the second of two successive reads of the same shared variable always returns a more recent value than the first.

Atomic registers are but one of many shared memory objects one might need when writing distributed protocols. Examples are clocks, counters, queues or stacks, to name but a few. One particularly interesting shared object is the *snapshot object*. A snapshot object distributed among n processes is an array of n cells stored in shared memory. Each processes can either atomically update the value of its own cell, or atomically scan the contents of all n cells in a single operation (see Fig. 1.3). To put it another way, one can view a snapshot object as a shared register storing a record that can either be read completely, or whose fields can be updated individually without destroying the contents of the other fields. If processes continually update their cell to correspond to their local state, each process in effect has access to the global configuration of the distributed system by scanning the snapshot object. Even when applied in a less drastic manner, snapshot objects constitute a very tight form of synchronization that clearly facilitates the implementation of distributed protocols.

When such, more complex, compound shared objects are not made

available by the hardware, one can implement them using the primitive shared objects that *are* available. Then, every operation on the compound object is implemented by some sequence of operations on the available primitive objects. These primitive objects store the data of the compound object being implemented, and are used for synchronization. Synchronization is needed to ensure that every execution of an operation on the high level object still appears to take effect instantaneously. Note that in reality these compound operations do take time, which is measured in the number of operations performed on the primitive objects, for each compound operation.

Because, by definition, atomic operations do not interfere with each other — in the sense that one comes before the other or the other way around — operations should be able to complete on systems were processes may crash, even if all other processes have crashed. Objects whose operations can always complete in a bounded number of steps even if some or all of the other processes have crashed are called *wait-free*.

Agreement In the *agreement* problem, initially each process holds some private input value. After exchanging information as specified by the agreement protocol (during which some processes may crash or even distribute false or inconsistent information) each process must irrevocably decide on some output value. All non-faulty processes are required to agree and to decide on the same value, and this decision value must have been the input to some process. Note that although an obvious decision value would be the majority among the values initially present as inputs, this is not at all required by the problem statement.

1.4 Overview of this thesis

Specialized though as it may seem to be, distributed computing is an active area of research that covers a wide range of problems. For many of these problems a comprehensive body of knowledge has only begun to emerge, and many interesting questions have yet remained unanswered. Maybe that explains why this thesis is less focussed on a single topic than I initially had in mind some four years ago. When the time came to wrap up the work and start writing this thesis, I decided that I would rather have it complete but less coherent, than leaving out some of the nice results I had obtained. Therefore, this thesis is best considered a proof of proficiency (*Dutch:* proeve van bekwaamheid) covering all problems discussed in the previous section.

Each chapter is based on a separate paper, each of which appeared

in the literature (see Appendix C for references and acknowledgements). I decided to keep the chapters self-contained, and to change their contents only in so far as to unify style and notation throughout the thesis. Again, each chapter is a separate entity that can be read independently of the others.

Chapter 2 The problem studied is that of giving a wait-free implementation of an atomic *snapshot object* using only single-writer single-reader shared registers. Each processes can either atomically update the value of its own cell, or atomically scan the contents of all n cells in a single operation. Even if all but one process crash, this one process must be able to either update or scan the snapshot object on its own. For the best known solutions to this problem both the update and the scan operation require $O(n \log n)$ steps. This still stands in stark contrast with the obvious lower bound of n steps surely needed to scan the values stored in each of the n cells.

We show that, given a *binary snapshot* object — whose cells can contain either 0 or 1 — one can implement snapshot objects storing arbitrary values in each cell with only linear time overhead. Hence the search for efficient implementations of snapshot objects can be restricted to the binary case. In fact, knowing that the snapshot object stores binary values makes the problem slightly easier because one can assume a process will only update its cell if the new value differs from the previous value. Hence, instead of an update operation, implementing an invert operation suffices.

Chapter 3 In most distributed systems each process has a unique and distinguishable *name*. Even for a small number of processes the range from which their names are taken may be large. For example, in a popular operating system like UNIX, process IDs may be 32 bits wide and thus range from 0 to 2^{32}. For protocols whose time or space complexity depends on the size of the name space (e.g., the implementation of a snapshot object if only the range of names is known as an upper bound on the number of processes), such a large name space may be prohibitively expensive if only a small number of processes actually participate. The *long-lived renaming* problem is concerned with reducing the size of the name space to allow such protocols to be used with less overhead.

Here up to k out of n processes repeatedly and concurrently acquire and release names from a much smaller destination name space. The problem is to give wait-free implementations for the operations that get or release a name, using only multi-writer registers. Of course the size of the destination name space should be as small as possible, but it is known that this size can-

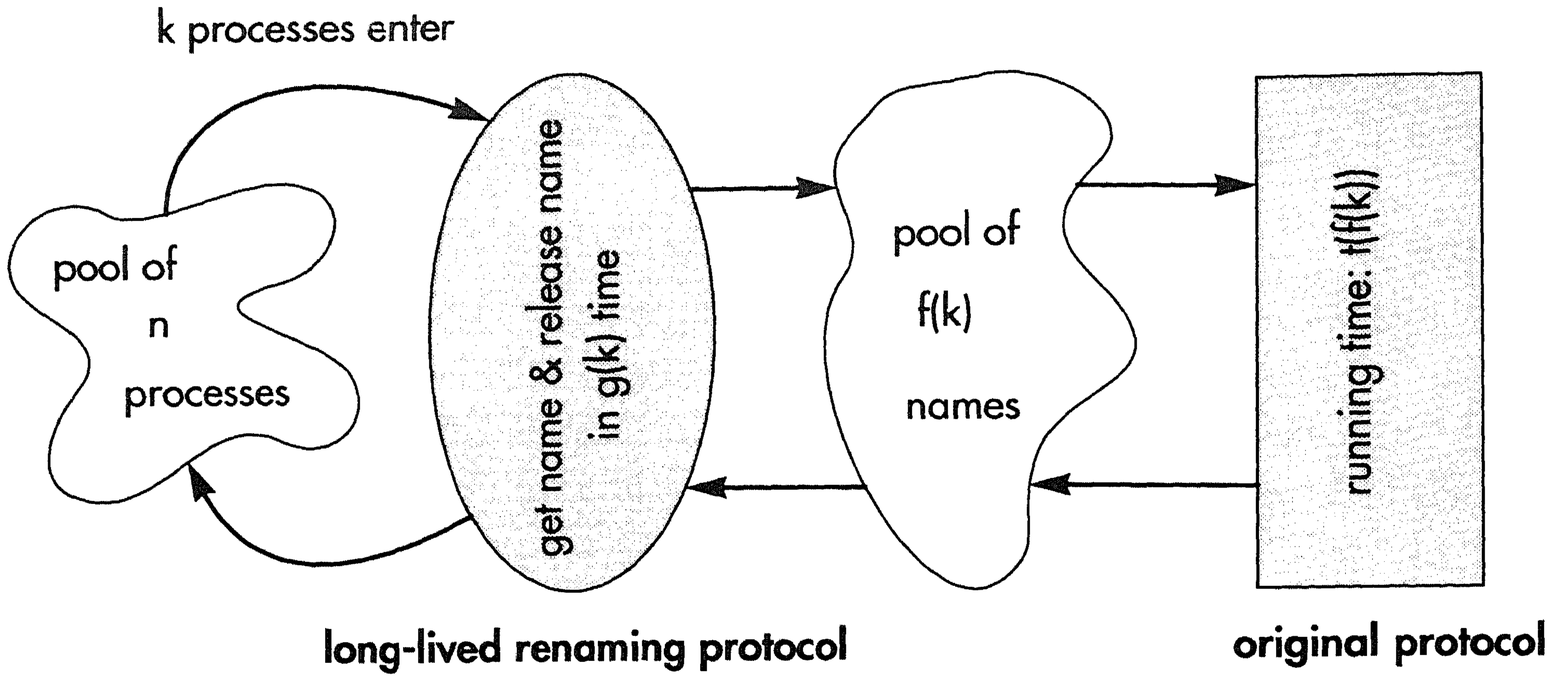

Figure 1.4 *Fast long-lived renaming reduces running time to g(k) + t(f(k)).*

not be smaller than $2k - 1$ [HS93]. Previous long-lived renaming protocols all have a time complexity depending on n or the size of the source name space. If for a long-lived renaming protocol the time complexity of both getting and releasing a name is a function of the number k of participating processes alone, we call such a protocol *fast*. In Chapter 3 I present the first fast and long-lived renaming protocol renaming to $O(k^2)$ names in $O(k^3)$ time.

The significance of this result is as follows. Consider a protocol whose time complexity depends on the size of the name space. Let us assume that only a small number of processes with names from a large range need to run this protocol concurrently. Without further arrangements, the amount of time spent would be large compared to the contention. Instead, assume each process first gets a new name running the fast long-lived renaming protocol (see Fig. 1.4). If this protocol renames the processes using a name space that depends on the small number of contending processes only, both protocols in sequence will be fast, i.e., have time complexity depending on the number of contenders only. If the number of contenders is small compared to the size of the original name space, this significantly improves the running time. Our fast long-lived renaming protocol shows, for the first time, that this approach is viable.

Chapter 4 In this chapter we focus on the *mutual exclusion* problem, i.e., the problem of passing a privilege fairly among all processes. Whether or not a process is privileged must be stored somewhere in its memory. Therefore, if some transient error disrupts the memory of certain processes, any number of processes may suddenly become privileged. A self-stabilizing mutual exclusion (*SSME*) protocol must correct this situation and eventually converge to a configuration in which at most one process is privileged. Such a protocol is useful in situations where a temporary violation of the mutual exclusion property costs less than preventing it, or detecting it and subsequently resetting the system. Moreover, detecting a violation, or resetting the system, may not be feasible at all.

SSME protocols have been extensively studied for undirected communication graphs where any two neighbours can read each others state. For directed communication graphs (where nodes can only read the state of their in-neighbours) only Dijkstra's protocol for directed rings [Dij74] and Tchuente's approach of covering a directed graph with directed rings were known [Tch81]. Tchuente's approach takes an exponential number of states per process, however. In Chapter 4 we give three protocols that each implement SSME on directed graphs using only a polynomial number of states per process. This is an exponential improvement. Reducing the space complexity of self-stabilizing protocols is especially important if they are to be implemented using simple hardware circuits. Constant space protocols are particularly sought after in these circumstances.

Chapter 5 *Ring-orientation* is the topic of the next chapter. Given an anonymous undirected ring as the communication graph, the unnamed and indistinguishable processes on the ring are required to orient the edges in between them. All edges must afterwards point in the same direction along the ring: either all clockwise, or all anti-clockwise. Moreover, the protocol is required to be self-stabilizing. The processes on the ring are required to find the orientation irrespective of the configuration in which the protocol starts.

It is known that rings of even length cannot be oriented by a deterministic protocol. A randomized orientation protocol for arbitrary rings, and a deterministic orientation protocol for odd-length rings do exist [IJ93]. However, this deterministic protocol uses $O(n)$ states per processor and hence depends on knowledge of the size of the ring. In this chapter we present self-stabilizing ring-orientation protocols for odd-length rings using only a constant number of states per processor. Hence these protocols can be applied, without modification, in odd-length rings of arbitrary size.

Chapter 6 So far we have studied process failures and transient memory errors separately. Processor failures were considered while constructing wait-free shared objects, and memory errors were countered by self-stabilizing protocols. Indeed, until recently, no protocols were studied that were resilient to both modes of error.

Here, we study an approach to merge the current models regarding process and memory failures into one single framework. We allow a fraction of the processes to fail by halting, and memory may get corrupted by transient errors, assuming that the ID and the protocol of each process is stored in non-volatile memory non-corruptible by errors. The problem studied is the construction of wait-free and self-stabilizing shared memory objects, and specifically the construction of wait-free stabilizing shared registers. We give a general definition of wait-free self-stabilizing shared objects that stabilize even if a fraction of the processes have crashed. Viability of this definition is shown by constructing these highly fault-tolerant registers from very basic stabilizing shared registers that resemble the safe bit introduced by Lamport [Lam86]. These basic registers are shown to have the minimal strength necessary to build more complex stabilizing shared objects.

Chapter 7 In the *mobile faults* model investigated in this chapter, a fixed number of computer viruses roam around in a computer network, without replicating themselves. A virus can move from one process to another with every message exchanged between the two. A virus that infects a process will take over control, after which the process will start misbehaving (i.e., behave according to the virus' instructions). Indeed, the viruses are assumed to be malicious, in the sense that they will try to disrupt the computation the processes are originally involved in. In other words, infected processes are assumed to suffer Byzantine failures.

We show that in a synchronous system where viruses can move with every round of message exchange, agreement can be reached and maintained in $O(n)$ rounds if at most one-third of the processes is faulty at any time and one process remains uncorrupted. This is optimal both in the number of rounds necessary, and in the fraction of processes allowed to fail. This shows that even if the viruses move around at full speed, agreement can still be reached.

Chapter 8 *Routing* is perhaps one of the most extensively used applications in everyday distributed systems, especially very large ones like the Internet. Hence its implementation should be efficient both in space and in

time. Because the time it takes for a message to arrive at its destination depends on the number of edges traversed, we are interested in shortest path routing algorithms.

The easiest way to implement the local routing function at a node is by means of a table indexed by all possible destinations. For each of these destinations the table lists the outgoing edge on the shortest path from this node to the destination. Every incoming message which is not addressed to the node itself is forwarded along the edge found for its destination in the table. In general, the amount of memory required to store this table is very large, namely $O(n \log n)$ bits per process in a system of n processes. The question is whether the local routing function can be stored more efficiently.

For special classes of communication networks, with a regular shape (like rings, meshes, trees, etc.) efficient representation of the local routing function is possible. Especially if one can change the names of the nodes used as addresses in the messages.

In this chapter we show several lower and upper bounds for the size of the routing tables, both for worst case networks, and on the average for all networks. We show that in most models, for almost all graphs $\Theta(n^2)$ bits are necessary and sufficient to store shortest path local routing functions at all nodes. There is a model were the average case lower bound rises to $\Omega(n^2 \log n)$, and there is another model where the average case upper bound drops to $O(n \log^2 n)$. These results are derived using incompressibility arguments based on Kolmogorov complexity.

Listen, ah kno it must sound absurd
But ah can hear the most melancholy sound
Ah ever heard!
Walk n cry! Kneel n cry!

NICK CAVE & THE BAD SEEDS
From Her to Eternity

2 Binary Snapshots

Abstract This chapter considers the wait-free *atomic snapshot* object in its simplest form where each cell contains a single bit. We demonstrate the 'universality' of this binary snapshot object by presenting an efficient linear-time implementation of the general multi bit atomic snapshot object using an atomic binary snapshot object as a primitive. Thus, the search for a fast wait-free atomic snapshot implementation may be restricted to the binary case.

2.1 Introduction

Consider a concurrent shared memory system. A snapshot memory object shared between n processes is a vector of n memory cells, one 'owned' by each process. All processes can independently and concurrently write to (*update*) the cell they own, and all processes can 'instantaneously' collect (*scan*) all values in the vector in a single operation.

The problem of implementing a wait-free atomic snapshot object was independently proposed and solved by Anderson [And90, And93, And94] and Afek *et al.* [AAD+90, AAD+93]. Anderson gives an exponential time[1] solution to this problem using single-writer multi-reader registers, and also considers the multi-writer case in which more than one process may update a partic-

[1] In the literature on this subject the time complexity is usually measured by the number of shared register accesses per operation as a function of the number of processes.

ular cell. In his solution for the multi-writer case he uses the single-writer snapshot object as a primitive, so his solution does not rely on multi-writer multi-reader registers. Afek *et al.* give a polynomial time implementation of a single-writer atomic snapshot object, also using single-writer multi-reader registers. They also consider the multi-writer case, but give a solution using multi-writer multi-reader registers instead.

The atomic snapshot memory object is a powerful tool to construct other atomic wait-free objects, like counters, logical clocks, or bounded concurrent time-stamping schemes. Aspnes and Herlihy [AH90] give a general method to convert a sequential specification of a shared memory object that satisfies certain constraints to a wait-free implementation of that object using an atomic snapshot memory object as a primitive. They also give a a polynomial-time implementation of a wait-free atomic snapshot object.

The main question remains whether it is possible to deterministically implement an atomic snapshot object with single-writer multi-reader registers such that the time complexity of both the update and the scan operations is linear. Much research has focused on affirming this, by imposing certain restrictions on the applicability of the solutions. In [KST91], Kirousis *et al.* present a linear-time solution for the case in which no two scans ever overlap. Dwork *et al.* [DHP+92] introduce the weaker time-lapse snapshot object, and give a linear time implementation of this object. Time-lapse snapshots satisfy the same properties atomic snapshots do, except that the former allow concurrent scans to contradict each other. Israeli *et al.* [ISS93, ISS95] present linear-time implementations for either the update or the scan operations, or for unbalanced systems in which the number of updaters is substantially smaller than the number of scanners, or vice versa. Attiya and Rachman [AR93] dramatically improve the time complexity of the general case to $O(n \log n)$ for both update and scan operations. Inoue *et al.* [ICM+94] further reduce the time complexity to $O(n)$, but have to resort to multi-writer registers to achieve this. Finally, Attiya *et al.* [AHR92] introduce the lattice agreement decision problem and show that a solution to this problem can be converted to a wait-free atomic snapshot implementation.

In this chapter we take a similar approach, and reduce the general atomic snapshot problem to a simpler one. We present a bounded space, linear time wait-free implementation of the general atomic snapshot object using an atomic wait-free *binary* snapshot object (where each cell can contain only two values) and a small number of safe and regular single-writer registers. Thus the search for a fast atomic snapshot implementation may be restricted to the binary case.

We will use a proof technique proposed by Awerbuch *et al.* [AKK+88], also used by Li *et al.* [LTV89] to prove the correctness of some atomic register constructions. The technique is a derivation of Lamport's system as described in [Lam86], where his two precedence relations *precedes* and *can affect* are replaced by a single interval order. We first present the model in Sect. 2.2, then we state the atomic wait-free snapshot problem in Sect. 2.3. The protocol is presented in Sect. 2.4, and is proven correct in Sect. 2.5.

2.2 The model

A concurrent shared memory system is a collection of sequential processes communicating asynchronously through shared memory data structures. At any time a process is executing at most one action. A process can at any time decide to invoke a new action when it is idle, or to respond to an ongoing action. The invocation time of an action a is denoted by $t_I(a)$ and the respond time by $t_R(a)$.

We model an execution of the shared memory system by a tuple $\langle \mathcal{A}, \rightarrow \rangle$. $\mathcal{A}$ is the set of all executed actions, that are assumed to be distinct. These actions are ordered by $\rightarrow$ such that a precedes b — denoted $a \rightarrow b$ — if $t_R(a) < t_I(b)$. We require for any execution $\langle \mathcal{A}, \rightarrow \rangle$ that for any $a \in \mathcal{A}$ there are only a finite number of actions $b \in \mathcal{A}$ with $\neg(a \rightarrow b)$. This way we require an execution to start at some point in time, rather than extending into the infinite past [Lam86, AKK+88]. With this definition, $\rightarrow$ is a special kind of partial order called an *interval order* (i.e., a transitive binary relation such that if $a \rightarrow b$ and $c \rightarrow d$ then $a \rightarrow d$ or $c \rightarrow b$). Now we have abstracted away from the actual time an action occurred, and we can specify the behaviour of actions involving access to the shared memory in terms of the interval order.

If one wishes to implement a certain *compound* shared memory object, one first assumes a set of *primitive* shared memory objects used in the implementation. Every operation on the compound object is implemented by a *protocol* which invokes actions on these primitive objects. Using the compound object will result in a *compound execution*. Since every operation on the compound object is implemented by a sequence of actions on the primitive objects, a compound execution $\langle \mathcal{A}, \rightarrow \rangle$ induces a *primitive execution* $\langle A, \rightarrow_A \rangle$ on the shared memory system. In a compound execution we model an operation as the set of actions it invokes. Then for operations $X, Y \in \mathcal{A}$, $X \rightarrow_{\mathcal{A}} Y$ iff all actions $x \in X$ precede all actions $y \in Y$ in $\langle A, \rightarrow_A \rangle$.

2.3 Atomic snapshot memories

A snapshot memory object on n processes is a vector of n memory cells. Number the processes from 1 to n. A process p can both write a new value to the p-th cell in the vector or instantaneously collect all values in the vector in a single operation. In the first case it performs an *update-operation*, in the latter case it performs a *scan-operation*.

We require our implementation to be *wait-free* to allow maximal concurrency, and failure-resiliency in the case of crash-failures. An implementation is wait-free if and only if all update and scan operations performed by any process will complete in an a priori bounded number of steps, regardless of the behaviour of the other processes.

Secondly, we require our implementation to be *atomic*. This means that all operations must appear to take effect at one instant of time during the actual time the operation executed. This allows us to 'shrink' the actual execution interval of an operation to a point, and we require a scan to return the values written by the most recent preceding updates. The next paragraph formalizes this.

Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary compound execution of a snapshot object. Then $\mathcal{A}$ is the set of all scan and update operations invoked in that execution. We assume for ease of presentation that $\mathcal{A}$ includes n initializing updates, one per processor, that precede all other operations in $\mathcal{A}$. The implementation of an atomic snapshot object is *correct* if for any of its executions $\langle \mathcal{A}, \rightarrow \rangle$ we can extend $\rightarrow$ to a total order $\Rightarrow$ such that for all scan operations $S \in \mathcal{A}$, S returns for any cell p the value written by the last update $U_p \in \mathcal{A}$ — executed by p — preceding S in $\Rightarrow$.

2.4 The solution

In the next two sections we give our implementation of the n process wait-free atomic snapshot object. The *architecture* describes all primitive shared memory objects used by the *protocols* — one for each type of operation on the shared memory object. The architecture also specifies for each primitive object the initial value, the operations each process is allowed to perform on it, and the type of values it holds.

The intuition behind our implementation is quite straightforward. Suppose update operations of process p write the new value alternately to two multi-reader registers $\mathsf{VAL}_p[0]$ and $\mathsf{VAL}_p[1]$ (a trick also used by Haldar and Vidyasankar [HV92] to avoid the need of large atomic registers in their atomic

snapshot constructions), after which they use an update on the binary snapshot to inform the scans of the position they wrote to. A scan first performs a scan on the binary snapshot, and tries to read the values from the registers VAL_p at the positions returned by the binary scan. As later updates may overwrite values before they are read by a concurrent scan, updates perform a scan operation as well, the result of which they write in the multi-reader register VIEW_p. A scan uses a *handshaking* mechanism to detect overwriting updates, in which case it copies the view written by an interfering update. The idea of letting updates perform a scan was introduced by Afek *et al.* in [AAD+90].

2.4.1 The architecture

Our implementation of an n process atomic snapshot memory — with cells of type T — will use one n process binary atomic snapshot object with, for every process p, operations $BinUpdate_p$ and $BinScan_p$. Each cell of this binary snapshot object is initially 0. In addition to this, our n-process atomic snapshot protocol will use the following shared registers. For each $p \in \{1, \ldots, n\}$:

- ▷ 2 safe registers of type T, $\mathsf{VAL}_p[0]$ and $\mathsf{VAL}_p[1]$, written by process p and read by all. Initially, $\mathsf{VAL}_p[1]$ may be arbitrary, but $\mathsf{VAL}_p[0]$ must be initialized to the desired initial value of cell p of the snapshot object.
- ▷ 1 regular register, VIEW_p (an n-value vector with elements of type T), written by process p and read by all, initially arbitrary.
- ▷ for each $q \in \{1, \ldots, n\}$: a safe bit C_{pq} (the 'complement'-bit), an atomic bit S_{pq} (the 'start'-bit) and a regular bit E_{pq} (the 'end'-bit). All written by process p, read by process q and initially 0.

Recall that, in the presence of concurrent operations, safe and regular registers satisfy weaker consistency guarantees than atomic ones. A register is *safe* if every read returns the most recently written value, unless the read is concurrent with a write in which case it may return an arbitrary value. A register is *regular* if every read returns the value written by a concurrent or an immediately preceding write.

2.4.2 The protocols

Every process p executes updates and scans according to Protocol 2.1. The Update-protocol uses local variables q (ranging over $\{1, \ldots, n\}$), and b, a static

```
operation Update_p(value)
        b := 1 - b ;
1       VAL_p[b] := value ;
2       BinUpdate_p(b) ;
        for q ∈ {1,...,n} do
3           S_pq := C_qp ;
4       VIEW_p := Scan_p() ;
        for q ∈ {1,...,n} do
5           E_pq := S_pq ;
```

```
operation Scan_p
        for q ∈ {1,...,n} do
6           C_pq := 1 - S_qp ;
7       b[1..n] := BinScan_p() ;
        for q ∈ {1,...,n} do
8           v[q] := VAL_q[b[q]] ;
9           if C_pq = S_qp
10            = E_qp
11          then return VIEW_q ;
        return v[1:n] ;
```

Protocol 2.1 *Scan and update protocols.*

bit variable initially 0, that retains its value in between successive invocations of the protocol. The Scan-protocol uses local variables b (an n-bit vector), q (ranging over $\{1,\ldots,n\}$), and v (an n-value vector with elements of type T).

A few words on the programming notation are in order. We use := to denote assignment, *name* to denote local variables, NAME to denote shared variables, and **name** for keywords. We choose not to mention the *Read*() or *Write*() actions on a shared variable explicitly. Instead, R := v is 'syntactic sugar' for *Write*(R, v). Similarly, if R occurs in an expression, *Read*(R) is implied. As usual, both actions may take some time to complete. Some assignments involve both a write and a read or a *Scan*(). These are to be executed sequentially, the read/scan first and then the write. E.g. $S_{pq} := C_{qp}$ is shorthand for t := *Read*(C_{qp}); *Write*(S_{pq}, t). This should not to be confused with read-modify-write operations that execute atomically.

We assume that the value of a shared register written by a process also belongs to that process's local state. This means that the value of for instance the shared variable C_{pq} in the Scan-protocol need not be explicitly read. The return statements in the Scan-protocol serve to return the indicated value to the caller, and to terminate the protocol immediately.

The for loops are indexed over a set to make clear that the n loop bodies may be interleaved arbitrarily. Since the registers accessed in the loop bodies are all disjoint, such a for statement can also be interpreted as a do-in-parallel construct. Thus the parallel time complexity [AGT+92] of snapshots equals the parallel time complexity of binary snapshots (up to a constant factor).

2.5 Proof of correctness

To prove correctness we assume the usual correctness conditions on the shared registers that we use in our implementation. We also assume the correctness of the atomic binary snapshot object used by our implementation. That is, in an execution $\langle A, \rightarrow_A \rangle$ we assume there exists a total order $\Rightarrow_A$ extending $\rightarrow_A$ such that every binary scan BS returns for bit p the value written by the last binary update BU_p executed by p preceding BS in $\Rightarrow_A$. In what follows, we write $\rightarrow$ for $\rightarrow_A$ and $\Rightarrow$ for $\Rightarrow_A$.

As a shorthand, we write U for *Update* and S for *Scan*. For operation $O \in \{U, S, BU, BS\}$, O_p^x denotes the x-th execution of O by process p, including scans S_p that are invoked by some update U_p^y. These scans are sometimes written as US_p^y. Note that BS_p^x is invoked by S_p^x, and BU_p^x is invoked by U_p^x. $\mathcal{A}$ contains all invocations S_p^x and U_p^x for $p \in \{1, \ldots, n\}$ and $x \geq 0$. Note that this also includes updates U_p^0 that wrote the initial values for the cells i, and scans US_p^x invoked by updates U_p^x.

If scan S_p^x sees $\mathsf{C}_{pq} = \mathsf{S}_{qp} = \mathsf{E}_{qp}$ at line 9–10, then process q (or some update U_q) is said to *interfere* with S_p^x. A scan is *direct* if no process interferes with it. S_p^x *contains* S_q^y iff $t_I(S_p^x) < t_I(S_q^y) < t_R(S_q^y) < t_R(S_p^x)$. The next lemma shows that direct scans will return correct values.

Lemma 2.1 *Assume q does not interfere with some scan S_p^x, and let S_p^x scan the value $b[q]$ updated by some U_q^y (i.e., $BU_q^y \Rightarrow BS_p^x \Rightarrow BU_q^{y+1}$). Then the value $\mathsf{VAL}_q[b[q]]$ read by S_p^x was written there by U_q^y.*

Proof Assume scan S_p^x does not see $\mathsf{C}_{pq} = \mathsf{S}_{qp} = \mathsf{E}_{qp}$ and that BS_p^x scans the value $b = b[q]$ for cell q updated by BU_q^y, i.e., $BU_q^y \Rightarrow BS_p^x \Rightarrow BU_q^{y+1}$.

The write of $\mathsf{VAL}_q[b]$ by U_q^y at line 1 precedes BU_q^y in $\rightarrow$, and the read of $\mathsf{VAL}_q[b]$ by S_p^x at line 8 follows BS_p^x in $\rightarrow$. Since $BU_q^y \Rightarrow BS_p^x$, we have $BS_p^x \not\leftarrow BU_q^y$, so the write of $\mathsf{VAL}_q[b]$ by U_q^y precedes the read of it by S_p^x. So if S_p^x does not read the value written by U_q^y it must be concurrent with or occur after a write to $\mathsf{VAL}_q[b]$ by a later update U_q^z at line 1. Note that this later update cannot be U_q^{y+1}, since this update writes to $\mathsf{VAL}_q[1-b]$.

Suppose the read of $\mathsf{VAL}_q[b]$ by S_p^x at line 8 is concurrent with or occurs after a write to it by an update U_q^z, $z > y+1$. Now $BS_p^x \Rightarrow BU_q^{y+1}$, so by a similar argument as before the read of C_{pq} by U_q^{y+1} at line 3 occurs after the write of C_{pq} by S_p^x at line 6 in $\rightarrow$. U_q^{y+1} writes the value of C_{pq} to S_{qp} and later to E_{qp} before S_p^x reads these, since the read of $\mathsf{VAL}_q[b]$ by S_p^x is

concurrent with or occurs after a write to it by update U_q^z. Now the values of C_{pq}, S_{qp} and E_{qp} are equal, and as long as S_p^x does not finish, C_{pq} does not change. This implies that any later writes to S_{qp} at line 3 and E_{qp} at line 5 by $U_q^{z'}$ do not change their value and thus, as they are atomic and regular, S_p^x sees $C_{pq} = S_{qp} = E_{qp}$, a contradiction. ◁

The next lemma shows that scans that cannot collect the values directly due to interfering updates can copy the result from such an interfering update. This interfering update will have stored the result, called a view, of a direct scan contained in the interfered scan.

Lemma 2.2 *If process q interferes with scan S_p^x, then the view S_p^x copied from VIEW_q is the result of a direct scan S_r^z, contained in S_p^x.*

Proof Since S_p^x sees $C_{pq} = S_{qp}$ at line 9 and S_{qp} is atomic and S_p^x sets $C_{pq} = 1 - S_{qp}$ at line 6, there is an update U_q^y that changes S_{qp} at line 3 after S_p^x reads S_{qp} at line 6. This implies that the scan US_q^y of U_q^y starts after S_p^x does. Note that after U_q^y changes S_{qp} at line 3, E_{qp} holds the old value of S_{qp} which is unequal to the current value of S_{qp}. Then if S_p^x also sees $C_{pq} = E_{qp}$, U_q^y writes E_{qp} at line 5 before or concurrent with the read of E_{qp} at line 10 by S_p^x. This implies that S_p^x reads VIEW_q at line 11 after U_q^y writes the result of US_q^y to it at line 4. This shows that S_p^x contains US_q^y. If US_q^y is a direct scan, we are done. If not, there is a sequence of at most $n - 3$ other scans each contained in its predecessor (and copying the scan returned by its successor) until finally a direct scan S_r^z occurs. Note that VIEW_q must be a regular register, since views written by later updates may interfere with the read of the view by S_p^x. ◁

We conclude by proving the correctness of our implementation of the atomic snapshot object. Both scans and updates are easily seen to make $O(n)$ accesses to shared registers, and to access the binary snapshot object at most 2 times. Hence, the implementation is wait-free.

Theorem 2.3 *For every execution $\langle \mathcal{A}, \rightarrow_{\mathcal{A}} \rangle$ there exists a total extension $\Rightarrow_{\mathcal{A}}$ of $\rightarrow_{\mathcal{A}}$ such that every scan S_p^x with $U_q^y \Rightarrow_{\mathcal{A}} S_p^x \Rightarrow_{\mathcal{A}} U_q^{y+1}$ returns for cell j the value written by U_q^y.*

Proof Define $\beta(\cdot)$ as follows. For direct scans S_p^x, let $\beta(S_p^x) = BS_p^x$. For indi-

rect scans S_p^x that copy the view collected by a direct scan S_q^y (see Lemma 2.2), let $\beta(S_p^x) = BS_q^y$. Finally, for updates, let $\beta(U_p^x) = BU_p^x$.

For any two actions $a, b \in \mathcal{A}$, define $a \Rightarrow_{\mathcal{A}} b$ if $\beta(a) \Rightarrow \beta(b)$. Note that neither $a \Rightarrow_{\mathcal{A}} b$ nor $b \Rightarrow_{\mathcal{A}} a$ iff $\beta(a) = \beta(b)$. By Lemma 2.2, $\beta(S)$ occurs inside S for any indirect scan S. So, if $a \rightarrow_{\mathcal{A}} b$ then also $\beta(a) \Rightarrow_A \beta(b)$ and thus $a \Rightarrow_{\mathcal{A}} b$. Then $\Rightarrow_{\mathcal{A}}$ extends $\rightarrow_{\mathcal{A}}$. Now extend $\Rightarrow_{\mathcal{A}}$ to a total order.

If for some scan S_p^x, we have $U_q^y \Rightarrow_{\mathcal{A}} S_p^x \Rightarrow_{\mathcal{A}} U_q^{y+1}$, then we also have $BU_q^y \Rightarrow_A \beta(S_p^x) \Rightarrow_A BU_q^{y+1}$ by the definition of β and $\Rightarrow$ (Note: if $\beta(a) = \beta(b)$, then both a and b are scans). If S_p^x is a direct scan, then $\beta(S_p^x) = BS_p^x$ and by Lemma 2.1 the theorem is proven. If S_p^x is not a direct scan, then by Lemma 2.2 it copies the result from a direct scan S_r^z, and thus $\beta(S_p^x) = BS_r^z$. But again by Lemma 2.1 the theorem is satisfied. ◁

2.6 Future research

Further research might be directed at finding an implementation of atomic binary snapshots with linear time complexity.

It is interesting to note that all atomic snapshot implementations we are aware of use at least $O(n)$ registers with $O(nv)$ size (where v is the maximal number of bits contained in any cell of the snapshot object). However, Dwork *et al.* [DHP+92] have shown that for time-lapse snapshots $O(n^2)$ registers with size $O(n + v)$ suffice. It is an interesting open question whether registers with size $O(nv)$ are necessary to implement atomic snapshot objects.

You never see me
Cause I'm always alone.

MINISTRY
N.W.O.

3 Long-Lived Renaming Made Fast

Abstract In the long-lived renaming problem — a generalization of the classical one-time renaming problem — n processors with unique names ranging over a *source name space* $\{0,\ldots,S-1\}$ repeatedly acquire and release unique names from a (smaller) *destination name space* $\{0,\ldots,D-1\}$. It is assumed that at most k out of n processors concurrently request or hold names. An efficient renaming protocol provides a useful front-end for protocols whose time complexity depends on the size of the name space containing the participating processes.

We consider long-lived renaming in the context of asynchronous, shared-memory multiprocessing systems that provide only read and write operations. A renaming protocol is *fast* iff the time complexity of acquiring and releasing a name is polynomial in k and independent of n and S. We present a wait-free, read/write protocol for long-lived renaming that achieves a destination name space of size $O(k^2)$ with time complexity $O(k^3)$. If S is polynomial in k, we further improve the time-complexity to $O(k \log k)$. This shows, for the first time, that fast, read/write protocols for long-lived renaming exist. Part of our wait-free solution uses mutual exclusion tournament trees, where we apply hashing based on polynomials over finite fields to avoid blocking. This technique may be of general interest.

3.1 Introduction

In the *one-time renaming problem* [BND89, ABND+90, BG93, PPT+94], n processes with unique identifiers in the range $\{0,\ldots,S-1\}$ obtain distinct names

from the smaller range $\{0,\ldots,D-1\}$ at most once. The *long-lived renaming problem* generalizes the one-time renaming problem by allowing processes to repeatedly acquire and release names. To model contention, it is furthermore assumed that at most k out of n processes acquire or hold names concurrently. Then a renaming protocol is *fast* if acquiring and releasing a name takes time $O(p(k))$ where p is a polynomial that is independent of both n, the total number of processes, and S, the size of their original name space.

A fast renaming protocol is a useful front-end for protocols whose time complexity depends on the size of the name space containing the participating processes. In particular, Anderson and Moir [AM94] have shown that the overhead associated with accessing a resilient shared object can be reduced by combining a long-lived renaming protocol with a shared object implementation for fewer processes. Fast renaming protocols can also be useful, for example, in Unix-based multiprocessing systems. In such systems, processes have unique identifiers from a large range, but the number of processes that run concurrently is usually much smaller. Suppose the time complexity of a task these processes are engaged in depends on the range of their identifiers. Then using a fast renaming protocol to reduce the size of that name space can dramatically improve performance.

We consider long-lived renaming protocols for asynchronous shared-memory multiprocessing systems. The long-lived renaming problem has been considered in this context by Moir and Anderson [MA94]. For systems supporting primitives such as *Test&Set*, they present renaming protocols that are both fast and long-lived. However, protocols that employ such strong operations are not as widely applicable or as portable as protocols that employ only reads and writes. We are therefore motivated to study fast, long-lived renaming protocols that use only read and write operations. Moir and Anderson's only read/write, long-lived renaming protocol, hereafter called MA, is not fast: it yields a name space of size $k(k+1)/2$ with time complexity $O(kS)$.

The long-lived renaming problem is related to the *slotted l-exclusion* (or *l-assignment*) problem [BP89b, ABND+90, BNDK+91]. In the latter problem, l distinguishable copies of a shared resource must be shared among n processes, k of which may fail. Processes unable to obtain a resource must wait until a slot is freed. For this problem it is known that if $l < n$, then necessarily $l > 2k$ (and if $l \geq n$, processes trivially claim the resource with their own name). Complementing these lower bounds, if $l = 2k+1$, there exist protocols for both asynchronous, shared-memory, multiprocessing systems [BP89b] and message passing systems [BNDK+91]. To be sure, these protocols are neither fast nor wait-free. However, [MG96] recently showed that these protocols can

be used as a back-end to a fast, long-lived renaming protocol to reduce the size of the destination name-space to $2k - 1$.

In this chapter, we present two long-lived renaming protocols: SPLIT and FILTER. These two protocols, combined with the non-fast protocol MA, yield the first fast *and* long-lived renaming protocol that uses only reads and writes to shared memory. Our protocol renames k processes to a name space of size $k(k+1)/2$, with time complexity $O(k^3)$. If the size S of the original name space is polynomial in k, protocol FILTER on its own renames k processes to a name space of size $O(k^2)$ with time-complexity $O(k \log k)$. We now present a brief description of the SPLIT and FILTER protocols, before presenting them in detail.

The SPLIT protocol uses a collection of building blocks called *splitters*. Each splitter, if accessed by at most m processes concurrently, dynamically partitions these processes into three output sets, each of which contains at most $m - 1$ processes. SPLIT employs a k-deep tree of splitters. Each leaf of the tree corresponds to a single name. A process p acquires a name by traversing the tree from the root to a leaf, accessing one splitter at each level. The splitter accessed at each level below the root is determined by the output set assigned by the splitter accessed at the previous level. After proceeding through $k - 1$ levels of the tree, a process is guaranteed to be in an output set that contains no other processes. Accessing a single splitter takes constant time. Thus, SPLIT renames to a name space of size 3^{k-1} with time complexity $O(k)$. A process releases a name by releasing each of the splitters it accessed. This also takes $O(k)$ time, so the SPLIT renaming protocol is fast.

FILTER is based on a set of mutual exclusion tournament trees — one for each destination name. In order to acquire a name, a process p competes for each of a set N_p of names by playing the mutual exclusion tournament associated with each name in N_p. We present a modified version of Peterson and Fischer's mutual exclusion tournament trees [PF77]. This modification enables processes to compete for all names efficiently and "in parallel". The set of names N_p is chosen in such a way that, while process p is attempting to acquire a name, there is always a name $x \in N_p$ for which no other process contends concurrently. This is achieved by the use of a special hashing technique that is based on unique polynomials over a finite field [EFF85]. Because there is always some tree T in which p is participating alone, the FIFO property of the mutual exclusion ensures that next time p participates in tree T, p makes progress towards the critical section of T. Therefore, p can eventually acquire a name.

Usually the size S of the source name space is bounded from above

by a function f of k. If $f \in O(k^c)$, then FILTER renames to a name space of size $O(c^2k^2)$ in time $O(k \log k)$. If $f = 3^{k-1}$, then FILTER renames to a name space of size $2k^4$ in time $O(k^3)$. If $f = 2k^4$, FILTER renames to a name space of size $72k^2$ in time $O(k \log k)$. Using protocol MA [MA94] we can further reduce the size of the name space to $k(k+1)/2$ in time $O(k^3)$. In practice we can assume $f \in O(3^k)$, so the first stage implemented by SPLIT is unnecessary.

The remainder of the chapter is organized as follows. Sect. 3.2 contains definitions used in the chapter. In Sect. 3.3 and 3.4, we present the SPLIT and FILTER protocols, respectively. Concluding remarks appear in Sect. 3.5.

3.2 Definitions

We consider an asynchronous, shared memory, multiprocessing environment in which n processes communicate through shared multi-writer variables that can be atomically read or written by any process. Each process p has a unique identifier from the set $\{0, \ldots, S-1\}$, where $S \geq n$.

A solution to the long-lived renaming problem consists of a wait-free implementation of operations $GetName_p$ — returning names in $\{0, \ldots, D-1\}$ — and $ReleaseName_p$ on a renaming object RN that is shared by n processes. For ease of discussion, define $name_p$ to equal:

?	if p is busy executing $GetName_p$ or $ReleaseName_p$, or
$\perp$	if the last operation executed by p was $ReleaseName_p$, and p is not busy executing $GetName_p$ or $ReleaseName_p$, or
$x \in \{0, \ldots, D-1\}$	if the last operation executed by p was $GetName_p$ returning x, and p is not busy executing $GetName_p$ or $ReleaseName_p$.

Initially, $name_p = \perp$ for all p.

It is required that every process p executes the operations $GetName_p$ and $ReleaseName_p$ alternately. $GetName_p$ can only be called if $name_p = \perp$, and $ReleaseName_p$ can only be called if $name_p \in \{0, \ldots, D-1\}$. Given that at most k processes concurrently access RN (i.e., $(\#p :: name_p \neq \perp) \leq k$), any solution to the long-lived renaming problem must guarantee that the following assertion is an invariant:

$$p \neq q \text{ and } name_p \in \{0, \ldots, D-1\} \text{ implies } name_p \neq name_q \ .$$

We measure the time complexity of our implementations by giving an upper bound on the number of shared memory accesses performed by any operation execution. We call a renaming protocol *fast* if both *GetName* and

ReleaseName have time-complexity polynomial in k and independent of S, the size of the original name space, and n, the total number of processes.

In our protocols, we use := to denote assignment, *name* to denote local variables, NAME to denote shared variables, and **keyw** for keywords. Each labelled statement in our protocols is assumed to be executed atomically. Note that each such statement contains at most one access of a shared variable.

For strings s of finite length over a finite alphabet, we use $l(s)$ to denote their length, $(s\,b)$ to denote the result of appending symbol b to string s, $s[1:k]$ to denote the string consisting of the first k symbols of s, and $s[i]$ to denote the ith symbol of s (for example, $s[1]$ is the first symbol of s). Also, we use A^h to denote the set of h-length strings over alphabet A and $A^{\leq h}$ to denote the set of such strings of length at most h. Finally, $(\#x : x \in X :: P(x))$ denotes the number of $x \in X$ such that $P(x)$ holds, and $|X|$ denotes the cardinality of X.

3.3 Renaming to 3^{k-1} names

In this section, we present the SPLIT protocol, which, for any S, renames to $3^k - 1$ names with time complexity $O(k)$. SPLIT uses a *splitter* similar to the one employed by Moir and Anderson [MA94]. Processes accessing our splitter are dynamically partitioned into three output sets, denoted -1, 0 and 1. The splitter ensures that, if at any time at most m processes access a splitter concurrently, then each output set contains at most $m - 1$ processes at any time. The splitter is much like a railway junction that splits an incoming track in three disconnected outgoing tracks. Suppose that there are never more than m trains that have crossed the junction and have not returned yet. Then the workings of the junction guarantees that no more than $m - 1$ trains are found on each of the three outgoing tracks -1, 0 and 1.

We start defining the splitter formally, and then show how it is used in the SPLIT protocol to implement long-lived renaming. Finally, in Sect. 3.3.1, we present an implementation of the splitter itself.

Splitter S consists of wait-free implementations of the two operations $Enter_p(S)$ — returning -1, 0, or 1 — and $Release_p(S)$. For ease of discussion, define $out_p(S)$ to equal :

?	if p is busy executing $Enter_p(S)$ or $Release_p(S)$, or
$\perp$	if the last operation executed by p was $Release_p(S)$, and p is not busy executing $Enter_p(S)$ or $Release_p(S)$, or

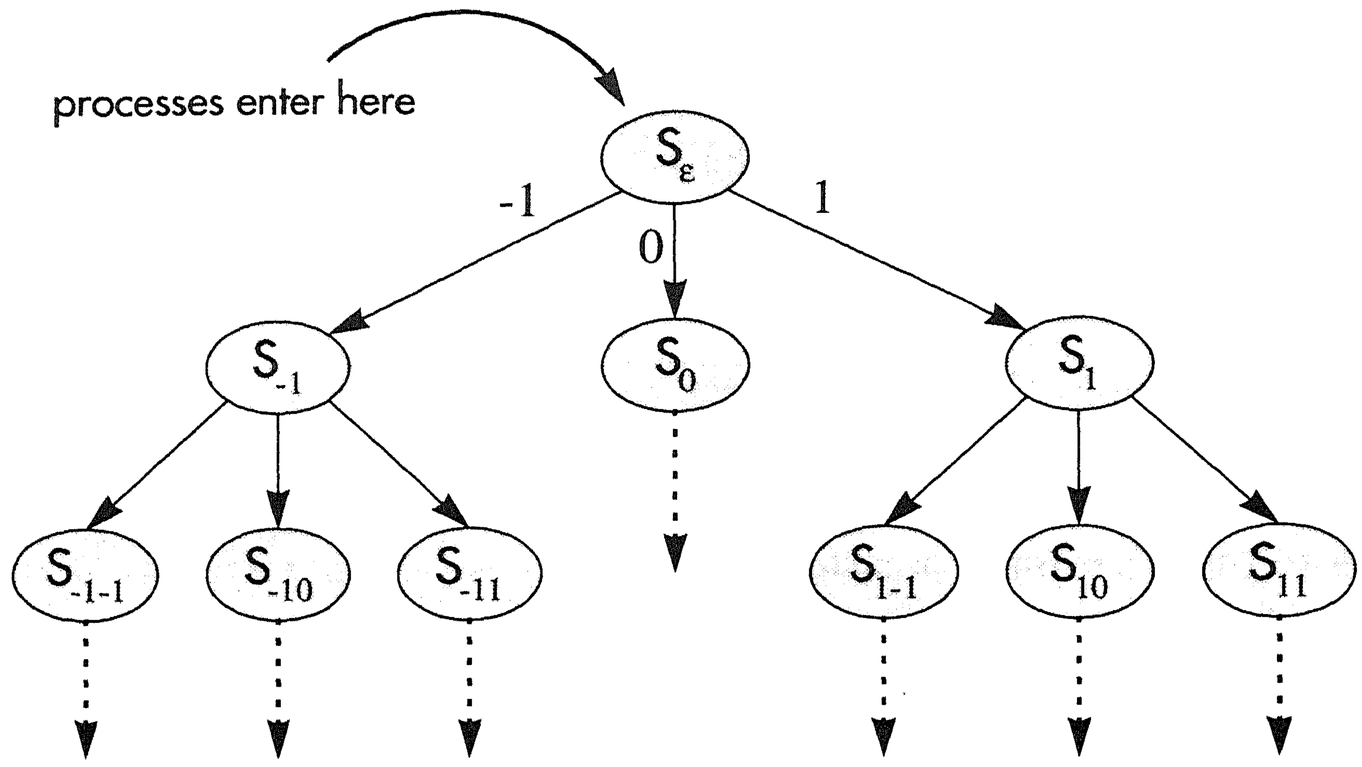

Figure 3.1 *The tree of splitters used in protocol* SPLIT.

$x \in \{-1, 0, 1\}$ if the last operation executed by p was $Enter_p(\mathsf{S})$ returning x, and p is not busy executing $Enter_p(\mathsf{S})$ or $Release_p(\mathsf{S})$.

Initially, $out_p(\mathsf{S}) = \perp$ for all p.

Again it is required that p executes $Enter_p(\mathsf{S})$ and $Release_p(\mathsf{S})$ on S alternately, starting with $Enter_p(\mathsf{S})$. Given that at most m processes concurrently access S (i.e., $(\#p :: out_p(\mathsf{S}) \neq \perp) \leq m$), the implementation of splitter S must guarantee that the following assertion is an invariant:

$$(\forall\, d \in \{-1, 0, 1\} :: (\#p :: out_p(\mathsf{S}) = d) < m) \ .$$

Protocol SPLIT, shown in Prot. 3.1, conceptually uses a tree of depth $k-1$ of splitters as shown in Figure 3.1, to rename k processes. Each interior node has three children — connected to each of the three outputs -1, 0, and 1. Each node in the tree is labelled by a string from $\{-1, 0, 1\}^{\leq k-1}$. The splitter at the root is labelled as S_ε. Every other splitter S is labelled as $(s\ d)$, where s is the label of S's parent, and d is -1, 0, or 1, corresponding to the output of S's parent to which it is connected. The leaves of the tree are nodes connected to the outputs of the splitters at level $k-1$. They are labelled by strings of length $k-1$ obtained by appending the corresponding output of the last splitter to

Local variables (static):
$s \in \{-1,0,1\}^{\leq k-1}$;
$e \in \{-1,0,1\}$;
$\ell \in \{1,\ldots,k\}$;

operation $GetName_p$
$s := \varepsilon$;
for $\ell := k-1$ **downto** 1
do $e := Enter_p(S_s)$;
$s := (s\ e)$; (* Append e to s *)
return $\overline{s}$; (* Compute name from path *)

operation $ReleaseName_p$
for $\ell := 1$ **to** $k-1$
do $Release_p(S_s)$;
$s := s[1:l(s)-1]$; (* Discard last label in s *)

Protocol 3.1 *Renaming protocol* SPLIT *for process p.*

the label of that splitter.

In order to acquire a name, process p starts at the root of the tree, and then traverses the tree until it reaches a leaf. At each node at level ℓ, p enters the splitter corresponding to that node. The output value returned by that splitter selects the next node at level $\ell+1$ to visit. The label s of the leaf reached is used to compute the name $\overline{s}$ to be returned as follows:

$$\overline{s} = \sum_{i=1}^{k-1} (1+s[i])3^{i-1} .$$

As the depth of the tree is $k-1$, the properties of the splitter guarantee that no other process is currently holding the same name. To release a name, a process simply releases all splitters it accessed in acquiring that name, in reverse order.

We now prove the correctness of protocol SPLIT.

Lemma 3.1 *For all* $s \in \{-1,0,1\}^{\leq k-2}$, *the following assertion is an invariant.*

$$(\#p :: out_p(S_s) \neq \perp) \leq k - l(s) .$$

Proof By induction on $l(s)$. For $l(s) = 0$ (i.e., $s = \varepsilon$), the lemma holds by assumption that at most k processes concurrently hold or attempt to acquire names. We inductively assume that the lemma holds for all strings over $\{-1, 0, 1\}$ of length at most m. Let s be any string over $\{-1, 0, 1\}$ of length $m + 1$, and set $t = s[1 : m]$. By the inductive hypothesis, the following assertion is an invariant.

$$(\#p :: out_p(\mathsf{S}_t) \neq \perp) \leq k - l(t) \ .$$

We assume that the splitter S_t is correct. Therefore, the correctness condition for the splitter implies that the following property is an invariant.

$$(\#p :: out_p(\mathsf{S}_t) = s[m+1]) \leq k - l(t) - 1 \ .$$

It is easy to show that $out_p(\mathsf{S}_s) \neq \perp$ implies $out_p(\mathsf{S}_t) = s[m + 1]$. Therefore

$$(\#p :: out_p(\mathsf{S}_s) \neq \perp) \leq k - l(s)$$

is an invariant. ◁

In the next section, we present a wait-free implementation of splitter S. This implementation performs $O(1)$ shared accesses per operation, which allows us to prove the following theorem.

Theorem 3.2 *Protocol* SPLIT *implements wait-free, long-lived renaming to* 3^{k-1} *names in time* $O(k)$*, and hence is fast.*

Proof Given that each splitter S can be implemented in a wait-free manner, it is easy to see that SPLIT is wait-free. Moreover, the splitter implementation presented in Sect. 3.3.1 performs at most 9 shared variable accesses per operation. As each operation of SPLIT accesses $k - 1$ splitters, the time complexity of SPLIT is $O(k)$.

As

$$\overline{s} = \sum_{i=1}^{k-1} (1 + s[i])3^{i-1} < 3^{k-1} \ ,$$

SPLIT yields a name space of size 3^{k-1}. Also, it is easy to show that $s \neq t$ implies $\overline{s} \neq \overline{t}$. Let p's name be $\overline{s}$, and let $s = (t\, d)$. Then p accessed splitter S_t at level $k - 1$ returning d. By Lemma 3.1, $(\#p :: out_p(\mathsf{S}_t) \neq \perp) \leq 2$ is an invariant. Now the correctness condition for splitter S_t implies $(\#p :: out_p(\mathsf{S}_t) = d) \leq 1$. Hence, distinct processes do not concurrently hold the same name. ◁

3.3.1 *Implementing the splitter*

In this section we present a wait-free implementation of the splitter S defined and used in protocol SPLIT in the previous section. It partitions processes into three sets, -1, 0, and 1. In order to join a set, process p calls $Enter_p(S)$ and joins the output set associated with the value returned. Later, p leaves that set by calling $Release_p(S)$[1]. As stated earlier, the implementation is required to ensure that, provided at most m processes concurrently access the splitter, no output set contains more than $m - 1$ processes at any time.

The splitter implementation appears in Prot. 3.2. While accessing the splitter, processes attempt to pass "advice" to each other about which output set can be safely joined. For example, if a process p chooses set 1, then p can safely advise another process to join set -1 because not all processes are in set -1. Similarly, if p leaves set 1, then p can advise another process to join set 1. Because of the difficulty of correctly passing advice between asynchronous processes using only read and write operations, the splitter employs an "interference detection" mechanism that allows the advice to be incorrect in all but one special case. In this special case, described below, the processes execute *Enter* "sequentially" (the steps of one process are not interleaved with those of another). This lack of interleaving allows correct advice to be passed between the processes in the special case, thereby ensuring that each output set contains at most $m - 1$ processes at any time. Below, we describe the splitter implementation in more detail, before giving a correctness proof.

LAST stores the identity of the last process to enter the splitter. Process p writes its identity to LAST upon entry, and reads it again at line 7 before deciding on a return value. If p reads LAST $\neq p$ (i.e., p detects "interference" from another process), then p returns 0, otherwise p returns *dir*. It is easy to show that at most $m - 1$ processes are in output set 0 at any time. To see why this is so, observe that if m processes are inside the splitter, then the last process q to assign LAST reads LAST $= q$ (does not detect any interference). In this case, q returns *dir*, which is easily shown to be non-zero. Thus, if m processes are inside the splitter, then at least one of the processes is not in output set 0.

To see that at most $m - 1$ processes are in output set 1 at any time is more complicated (the case for set -1 is symmetric). First, note that if m processes are in output set 1, then each process p read LAST $= p$ at line 7.

[1] In the remainder of this section we omit the suffix (S), because we are interested in one splitter from now on.

Shared multi-writer variables:

LAST $\in \{0, \ldots, S-1\}$;
ADVICE[1] $\in \{\perp, -1, 1\}$; **initially** 1
ADVICE[2] $\in \{-1, 1\}$; **initially** 1

Local variables (static):

dir $\in \{-1, 1\}$;
adv2 $\in \{true, false\}$;

operation $Enter_p$

```
1    LAST := p ;
2    dir := ADVICE[1] ;
     if dir = ⊥
3       then dir := ADVICE[2] ;
4    ADVICE[1] := −dir ;
5    adv2 := (LAST = p) ;
     if adv2
6       then ADVICE[2] := −dir ;
7    if LAST = p
        then return dir ;
        else return 0 ;
8       (* Working Section *)
```

operation $Release_p$

```
9    if LAST = p
10      then ADVICE[1] := dir ;
     if ¬adv2
11      then ADVICE[1] := ⊥ ;
12      (* Remainder Section *)
```

Protocol 3.2 *Process p's code for splitter S.*

This is only possible if m processes execute *Enter* "sequentially" — that is, each process executes step 7 before the following process executes step 1. Consider the second-to-last process q to execute *Enter*. Because q detects no interference, and because q joins group 1, it is easy to show that q assigns -1 to ADVICE[1] and ADVICE[2] at steps 4 and 6 respectively. Informally, this represents advice to the last process p to join set -1. If this advice remains until p executes *Enter*, then it is easy to see that p joins output set -1, contradicting the assumption that all m processes are in set 1 concurrently. A key property in the correctness proof presented below, is that when p executes *Enter* in this scenario, q's advice is intact. This property is based on the observation that, apart from the $m-1$ processes already in an output set after q executes step 7, at most one process accesses the splitter concurrently

before p executes 1. Because none of the remaining $m-1$ processes take any steps in this interval, it is straightforward to show that ADVICE[1] $= -1$ or ADVICE[1] $= \perp \wedge$ ADVICE[2] $= -1$ holds when p begins executing *Enter*. In either case, p joins set -1, and therefore does not violate the correctness condition for the splitter. We now present a formal correctness proof for the splitter.

3.3.2 Correctness of the splitter

In this section, we assume that $(\#p :: out_p \neq \perp) \leq m$ holds throughout a fixed, but arbitrary execution (modelled by $\Rightarrow$) of the splitter implementation shown in Prot. 3.2. To show that the implementation is correct, we prove that, for each $d \in \{-1, 0, 1\}$, the following assertion is also invariant.

$$(\#p :: out_p = d) < m\ .$$

In the following, we use p to denote both process p and an invocation of $Enter_p$ and the corresponding $Release_p$. In the rare instances where we consider different invocations by the same process, we distinguish the invocations (e.g., using p and p').

We use out_p to denote the value for *Enter* returned by invocation p; $p.1$ denotes the atomic action of executing line 1 by invocation p; $p@1$ means that the next line to be executed by process p is 1; $p_i@8\text{–}10$ means that the next line to be executed by process p lies in the range $8, \ldots, 10$. Finally, dir_p denotes the value of local variable *dir* held by processor p during this particular invocation.

The following claim is used to prove the above invariant for $d = 0$.

Claim 3.3 *For an arbitrary invocation p, $out_p = 0$ iff there exists an invocation q such that $p.1 \Rightarrow q.1 \Rightarrow p.7$.*

Proof If there exists a q such that $p.1 \Rightarrow q.1 \Rightarrow p.7$, then at line 7, p reads LAST $\neq p$ and so $out_p = 0$, else p reads LAST $= p$ and then $out_p \neq 0$. ◁

For the case where $d = -1$ or $d = 1$ we first prove the following lemma.

Lemma 3.4 *Let $d \neq 0$. Suppose that at some point there are $m-1$ invocations $p_1, \ldots, p_{m-1}$ with $p_i@8\text{–}10$ and $out_{p_i} = d$, and there is an invocation q with $q@4$ and $p_i.7 \Rightarrow q.1$ for $1 \leq i \leq m-1$. Then $dir_q = -d$.*

Proof Without loss of generality, assume $p_i.1 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq m-2$. By Claim 3.3 and by assumption $out_{p_i} \neq 0$ for $1 \leq i \leq m-1$, we also have $p_i.7 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq m-2$. In other words, $p_1, \ldots, p_{m-1}$ must have entered sequentially. Towards a contradiction, suppose that at time t for the first time the conditions of the lemma hold, but $dir_q = d$.

There are four cases for the last write to ADVICE[1] before q reads it. Either there is some invocation r writing ADVICE[1] after p_{m-1} did (case 1, 2, and 3 below), or not (case 4 below). If some invocation r writes ADVICE[1] after p_{m-1} but before q reads it, it writes ADVICE[1] in line 10 (case 1), in line 11 (case 2), or in line 4 (case 3). In the first three cases r must finish before q starts, because $(\#p :: out_p \neq \perp) \leq m$. Hence if r executes line 10 or 11 then $r.10, r.11 \Rightarrow q.1$. Also in these three cases, as $out_{p_{m-1}} \neq 0$, by Claim 3.3 we have either $r.1 \Rightarrow p_{m-1}.1$ (case *a*) or $p_{m-1}.7 \Rightarrow r.1$ (case *b*).

1. For some r, $p_{m-1}.4 \Rightarrow r.10 \Rightarrow q.2$. Then q reads ADVICE[1] $= dir_r = d$. If at line 9, LAST $= r$ then also at line 7 and 5. Hence $out_r = d$. But then at a time $t' < t$ we have an earlier bad case contrary to assumption.
   For if $p_{m-1}.7 \Rightarrow r.1$ (case *b*) then at time t' where r@4 we have $dir_r = d$ and still p_i@8-10 and $out_{p_i} = d$ for $i = 1, \ldots, m-1$.
   And if $r.1 \Rightarrow p_{m-1}.1$ (case *a*, the only other possibility, see above) as LAST $= r$ at line 9 we must have $r.9 \Rightarrow p_{m-1}.1$. But then at time t' where p_{m-1}@4 we have $dir_{p_{m-1}} = d$ by assumption that $out_{p_{m-1}} = d$ and still p_i@8-10 and $out_{p_i} = d$ for $i = 1, \ldots, m-2$ and (because $p_{m-1}.4 \Rightarrow r.10$ by assumption) also r@8-10 and $out_r = d$.
2. For some r, $p_{m-1}.4 \Rightarrow r.11 \Rightarrow q.2$. Then q reads ADVICE[1] $= \perp$ and hence reads ADVICE[2] $= d$ at line 3. Then this value is written there by p_{m-1}—because p_{m-1} sees LAST $= p_{m-1}$ at 7 by assumption that $out_{p_{m-1}} \neq 0$, surely p_{m-1} sees LAST $= p_{m-1}$ at 5 and hence executes $p_{m-1}.6$. But then $out_{p_{m-1}} = -d$, a contradiction.
   To see why p_{m-1} is the last to write ADVICE[2], let us suppose to the contrary that there is a r' with $p_{m-1}.6 \Rightarrow r'.6 \Rightarrow q.3$. Because r executes line 11, it does not execute line 6, so $r' \neq r$. As by assumption r is the last to write ADVICE[1] at line 11, $r'.2 \Rightarrow r.11$. By $(\#p :: out_p \neq \perp) \leq m$ then also $r'.6 \Rightarrow r.1$, and using $p_{m-1}.6 \Rightarrow r'.6$ we get $p_{m-1}.1 \Rightarrow r.1$. Again by $(\#p :: out_p \neq \perp) \leq m$ then r reads LAST $= r$ at line 5 thus setting $adv2_r = true$, contrary to the assumption that r executes step 11.

3. For some $r \neq p_{m-1}$, $p_{m-1}.4 \Rightarrow r.4 \Rightarrow q.2$. As $(\#p :: out_p \neq \perp) \leq m$, if $p_{m-1}.7 \Rightarrow r.1$ (case b) then r reads LAST $= r$ at line 9 and executes line 10. But then $r.10 \Rightarrow q.2$; this is case 1 above.
   Now consider $r.1 \Rightarrow p_{m-1}.1$ (case a, the only other possibility, see above). By $p_{m-1}.4 \Rightarrow r.4$ we get $r.1 \Rightarrow p_{m-1}.1 \Rightarrow r.5$, hence r reads LAST $\neq r$ at line 5. Then $adv2_r = false$ and r executes line 11. But then $r.11 \Rightarrow q.2$; this is case 2 above.
4. q reads from ADVICE[1] what p_{m-1} writes there (i.e., none of the above cases) at $p_{m-1}.4$ (because p_{m-1}@8–10, p_{m-1} did not execute line 10 yet). As $out_{p_{m-1}} = d$ this must be $-d$, a contradiction.

This completes the proof of Lemma 3.4. ◁

Theorem 3.5 *For each $d \in \{-1, 0, 1\}$, the following assertion is invariant.*

$$(\#p :: out_p = d) \leq m - 1 \ .$$

Proof Suppose for $d = 0$ the theorem does not hold. Then at some point $(\#p :: out_p = 0) = m$. Of all these invocations p, let q be the last to write LAST in step 1 (i.e., $p \neq q$ implies $p.1 \Rightarrow q.1$). Then by $(\#p :: out_p \neq \perp) \leq m$ there is no invocation r such that $q.1 \Rightarrow r.1 \Rightarrow q.7$. Hence by Claim 3.3, $out_q \neq 0$, a contradiction.

Towards a contradiction in the case where $d = -1$ or $d = 1$, suppose $(\#p :: out_p = d) = m$ at some point. Then similar to the proof of Lemma 3.4, by Claim 3.3 $p_1, \ldots, p_m$ must have entered sequentially, i.e., $p_i.7 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq m-1$. Then at p_m@4 we have p_i@8–10 and $out_{p_i} = d$ for $1 \leq i \leq m-1$ and hence by Lemma 3.4 $dir_{p_m} = -d$. This contradicts $out_{p_m} = d$. ◁

3.4 Reducing the name space

Protocol SPLIT on its own implements fast, long-lived renaming. However, the size of the resulting name space is 3^{k-1}, which is unacceptably high. We are therefore motivated to find long-lived renaming protocols that can reduce the size of the name space. Protocol FILTER, to be presented in this section, achieves exactly that.

In summary, if $S \leq 3^k - 1$ (the result after applying SPLIT), FILTER renames to a name space of size $2k^4$ with time complexity $O(k^3)$. Also, if $S = 2k^4$, FILTER renames to a name space of size $72k^2$ with time complexity $O(k \log k)$. Applying these three protocols one after another yields a fast,

long-lived renaming protocol to $O(k^2)$ names. More applications of our long-lived renaming protocols are presented in Sect. 3.4.4. We now give a brief overview of the FILTER protocol, before presenting it in detail.

FILTER uses a collection of mutual exclusion tournament trees — one tree T_m for each name m in $\{0, \ldots, D-1\}$ (recall that D is the size of the destination name space). To acquire a name m, a process competes in the mutual exclusion tree T_m associated with that name. This immediately guarantees that no two processes ever hold the same name simultaneously.

The use of mutual exclusion in a wait-free protocol might seem counterintuitive. However, as described in detail below, each process p competes "in parallel" for each of a set N_p of names. This is allowed for by using a modified version of Peterson and Fischer [PF77] mutual exclusion tournament trees. The modification allows a process to detect that it is blocked in one tree, and to attempt to acquire another name from N_p by continuing to compete in another tree associated with that name.

The collection of sets N_p is constructed using a special hashing technique involving polynomials over a finite field, such that no set is covered by the union of $k-1$ others. Collections with that property were studied by Erdös *et al.* [EFF85], and in [BLS93]. In our application this property ensures that, at any time, there is some name $m \in N_p$ for which p is competing alone.

Section 3.4.1 below explains this hashing technique in detail. Then, in Sect. 3.4.2, we present the modified mutual exclusion tree and the FILTER protocol. Correctness of the protocol is proven in Sect. 3.4.3. Finally, in Sect. 3.4.4, we discuss the performance of FILTER under varying assumptions on the relationship between S and k.

3.4.1 Hashing names to sets of names

The hashing technique used to assign a set of names to each process uses two parameters d and z, which are chosen based on particular values of k and S. As is discussed later, the choices of d and z for given values of k and S influence the time and space complexity of the resulting instance of the FILTER protocol, as well as the size of the destination name space. We now show how the set of names N_p, for which process p competes, is defined, and state the constraints on the parameters d and z as we proceed.

First, let $\mathrm{GF}(z)$ be a finite field, with z a prime (i.e., the elements of the field range over the set $\{0, \ldots, z-1\}$). Each process p must be assigned a polynomial $Q_p(x) = a_d x^d + \cdots a_1 x + a_0$ of degree d over field $\mathrm{GF}(z)$ (where $0 \leq a_i < z$, and multiplication and addition are performed modulo z), such

that the polynomials assigned to distinct processes differ in at least one coefficient. If

$$S \leq z^{d+1} , \tag{3.1}$$

then we can assign to each process p the polynomial with for each i, $0 \leq i \leq d$, $a_i = (p \text{ div } z^i) \bmod z$. In other words, the i-th coefficient equals the i-th digit in the z-ary denotation of p's name.

Define $n_p(x) \triangleq zx + Q_p(x)$ (not doing arithmetic modulo z here) and $N_p \triangleq \{n_p(0), \ldots, n_p(2d(k-1)-1)\}$. Note that for arbitrary $p \in \{0, \ldots, S-1\}$ and $x \in \mathsf{GF}(z)$, $Q_p(x) < z$. It follows that $n_p(x) = n_q(y)$ if and only if $x = y$ and $Q_p(x) = Q_q(y)$. Thus, there are $2d(k-1)$ distinct names in N_p. Also, for distinct polynomials Q_p and Q_q of degree d over a finite field $\mathsf{GF}(z)$ with z prime, there are at most d values of x such that $Q_p(x) = Q_q(x)$ [Coh74]. Recall that for $p \neq q$, Q_p and Q_q are distinct. Then

$$z \geq 2d(k-1) \tag{3.2}$$

implies $|N_p \cap N_q| \leq d$. This guarantees that there are at most d names competed for by both of any pair of two processes.

Furthermore, suppose P is an arbitrary set of $k-1$ processes such that $p \notin P$. Then there are at most $d(k-1)$ names $n_p(x) \in N_p$ which are also a member of some N_q with $q \in P$. As $|N_p| = 2d(k-1)$, this implies that there exist at least $d(k-1)$ names $n_p(i) \in N_p$ which are not a member of any N_q with $q \in P$.

In the FILTER protocol presented in the next section, process p competes only for names in N_p. Because at any time while p is attempting to acquire a name, at most $k-1$ other processes acquire or hold names, the property above implies that at any time, there are $d(k-1)$ names in N_p for which no other process is competing. This property is crucial in proving that process p can always acquire a name. Note that if we had required $z > d(k-1)$ a similar argument would have shown that there is at least one name in N_p for which no other process is competing. However, taking $z \geq 2d(k-1)$ allows us to arrive at a better bound on the time complexity in Theorem 3.10, at the expense of a small increase in size of the resulting name space.

The largest name competed for by any p is the maximum over all p and all x ranging over $0 \leq x \leq 2d(k-1)-1$ of $n_p(x) = zx + Q_p(x)$. Clearly $Q_p(x)$ is bounded by z, so this shows that no process competes for a name larger than $z2d(k-1)$. Thus, an instance of the FILTER protocol specified by d and z achieves a destination name space of size

$$D = z2d(k-1) . \tag{3.3}$$

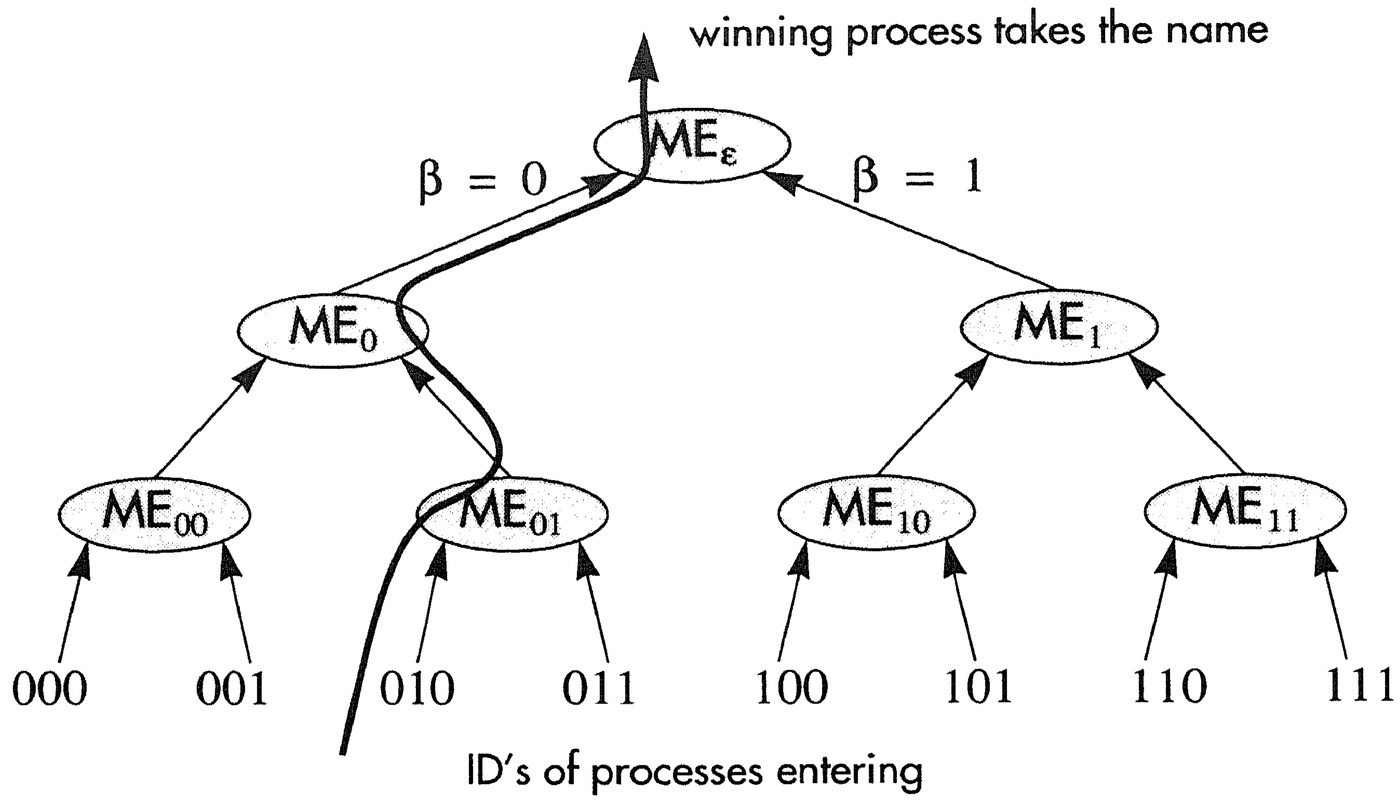

Figure 3.2 *A tournament tree for 8 processes.*

To achieve the smallest destination name space possible, d and z should be chosen to minimize $z2d(k-1)$, while satisfying (3.1) and (3.2) and the requirement that z be prime. In Sect. 3.4.4, we discuss various settings of d and z and the resulting name-space size for several combinations of k and S. We now present the FILTER protocol.

3.4.2 *Protocol* FILTER *in detail*

For each name $m \in \{0,\ldots,D-1\}$, FILTER uses a binary mutual exclusion tournament tree T_m of $\lceil \log S \rceil$ levels. The leaves are at level 1 and the root is at level $\lceil \log S \rceil$. Each node in tree T_m is a distinct two-process mutual exclusion block (ME) with two "inputs" labelled 0 for left and 1 for right. A process playing in a tournament tree starts by entering a mutual exclusion at level 1 over one of its two inputs. If p reaches the critical section of a mutual exclusion at level ℓ, it advances to the parent at level $\ell + 1$. We write $\mathsf{ME}^m_{p,\ell}$ for the ME block in tree T_m played by process p at level ℓ. No two different processes must ever enter the tournament tree at the same ME block at level 1 over the same input. Only then, the construction of the tournament tree, and the correctness of the two-process mutual exclusion guarantee that no two processes can, at the same time, be in the critical section of the ME block at

Shared multi-writer variables:
R[0], R[1] $\in \{\perp, 0, 1\}$;

operation *Enter*(ME, β)
if R[$1-\beta$] = $\perp$ **then** R[β] := 1
else R[β] := $\beta \oplus$ R[$1-\beta$] ;
if R[$1-\beta$] $\neq \perp$ **then** R[β] := $\beta \oplus$ R[$1-\beta$] ;

operation *Check*(ME, β)
return (R[$1-\beta$] = $\perp$) $\vee$
$\beta \oplus$ (R[β] $\neq$ R[$1-\beta$])) ;

operation *Release*(ME, β)
R[β] := $\perp$;

Protocol 3.3 *Implementation of the 2-process mutual exclusion block ME.*

the root of the tree. Then, a process reaching the critical section at the root of T_m can safely acquire name m.

Let us define the structure of the tournament trees, as well as the mutual exclusions played by a certain processor, more rigorously. To this end, label every ME block at level ℓ by a binary string of length $\lceil \log S \rceil - \ell$, as follows. The root ME block is labelled ε. If ME at level $\ell - 1$ is connected to input i of ME′ at level ℓ, and if ME′ has label s, then ME has label $(s\, i)$. Note that there are $2^{\lceil \log S \rceil} \geq S$ "inputs" to the leaves of the tree. Consider p's name to be a binary string of length $\lceil \log S \rceil$ (padded in front by 0's). Let p enter tree T_m at the ME block at level 1 with label $p[2 : \lceil \log S \rceil]$ over input $p[1]$. Then no other process will enter this ME block over the same input, as required. Given the above labelling of the ME blocks, it follows that $\mathsf{ME}^m_{p,\ell}$ has label $p[\ell + 1 : \lceil \log S \rceil]$. Process p enters $\mathsf{ME}^m_{p,\ell}$ over input $p[\ell]$. Figure 3.2 shows an example.

To allow processes to compete in several tournament trees "in parallel", the two-process ME block used in these trees is a modified version of the two-process mutual exclusion algorithm of Peterson and Fischer [PF77]. Peterson and Fischer's algorithm is split into three procedures, *Enter*, *Check*, and *Release*, and uses multi-writer variables to avoid the costly search for an opponent. Except for these modifications, both algorithms are essentially

```
repeat (* this starts a new round *)
    forall m ∈ N_p do
        if p did not enter tree m yet
          then Enter tree T_m at the appropriate leaf.
               Let ℓ be the last entered level of tree T_m.
               while ℓ ≠ ⌈log S⌉ and checking ME at level ℓ in
                     tree T_m returns true
               do Let ℓ := ℓ + 1 and enter ME at level ℓ.
               if ℓ = ⌈log S⌉ and checking ME at level ℓ in
                  tree T_m returns true
                  then return name m
until a name was found
```

Protocol 3.4 *How p gets a name in the* FILTER *protocol.*

the same. The implementation of the two-process ME block is presented in Prot. 3.3. In this protocol, we use $\oplus$ to denote *exclusive or*, and $\vee$ to denote disjunction. We set *true* = 1 and *false* = 0.

Processes enter a mutual exclusion block ME either from the left (0) or from the right (1) subtree, as indicated by the parameter β. In order to compete in a particular ME block from direction β, process p calls *Enter*(ME, β) and then repeatedly calls *Check*(ME, β) until *Check* returns *true*. At this point p is in the critical section of ME. Later, p calls *Release*(ME, β) in order to release ME. Each mutual exclusion block ME uses a separate multi-writer register R.

Because processes are mapped one-one to the inputs of the ME blocks at level 1, and because a process must reach the critical section of one ME block before accessing that block's parent, no two processes concurrently compete in any ME block from the same direction. Thus, each two-process ME block is accessed by at most two processes concurrently, one with $\beta = 0$ and the other with $\beta = 1$. This is the essence of the correctness proof, presented in Lemma 3.6 below, for each mutual exclusion tree.

Having described the tournament trees and the mutual exclusion blocks they use, we can now explain protocol FILTER. An informal description how process p obtains a name appears in Prot. 3.4. The details appear in Prot. 3.5. Process p competes "in parallel" in all the mutual exclusion trees associated with names in N_p (as defined in Sect. 3.4.1 above). This is achieved

```
operation GetName_p
    for m ∈ N_p do level_p[m] := 0 ;          (* initially, no tree is entered yet *)
    repeat                                      (* this starts a new round *)
        for m ∈ N_p do                          (* try all names in any order *)
            if TryTree_p(m) then return m;
    until false ;

operation TryTree_p(m)
    if level_p[m] = 0                           (* if p hasn't entered T_m yet *)
      then Enter_p(ME^m_{p,1}, p[1]) ; ℓ := 1   (* enter T_m at level 1 *)
      else ℓ := level_p[m] ;    (* resume where you left in previous round *)
    while (ℓ ≠ ⌈log S⌉) ∧ Check_p(ME^m_{p,ℓ}, p[ℓ])   (* increase level until *)
      do ℓ := ℓ + 1 ; Enter_p(ME^m_{p,ℓ}, p[ℓ]) ;    (* ME is blocked *)
    level_p[m] := ℓ ;                           (* save reached level *)
    return (ℓ = ⌈log S⌉ ∧ Check_p(ME^m_{p,ℓ}, p[ℓ]) ;

operation ReleaseName_p
    for m ∈ N_p
    do for ℓ = level_p[m] to 1                  (* release all claimed ME blocks *)
        do Release_p(ME^m_{p,ℓ}, p[ℓ]) ;        (* (order is relevant) *)
```

Protocol 3.5 *Precise implementation of protocol* FILTER.

by proceeding through a tree as far as possible (until a call to *Check* returns *false*) and then switching to the tree for the next name. This is repeated until a name has been acquired (by reaching the critical section of the root of the tree for that name), or all names have been tried. In the latter case, process p returns to the tree for p's first name and tries all the trees again. This is repeated until p acquires a name.

To release a name (only shown in Prot. 3.5), process p releases all mutual exclusion blocks it entered when acquiring the name, including blocks in which p did not reach the critical section. Mutual exclusion blocks must be released in reverse of the order in which they were entered, i.e, the last block entered must be released first.

To see that p eventually acquires a name using protocol FILTER, recall that the set of trees competed for by p is chosen in such a way that, at any

time, one of the trees in p's set is not being accessed by any other process (provided at most k processes request or hold names concurrently) . Suppose that at some point t, p is competing for name m, and no other process is competing for name m. If p fails to acquire a name, then p eventually tries to proceed in tree T_m. Because there was no other process competing for T_m at time t, the FIFO property of the 2-process mutual exclusion block ensures that p is able to proceed at least one level in T_m. Repeating this argument, p eventually reaches the critical section of some tree, and therefore acquires a name. This argument is the essence of FILTER's correctness proof, which is presented in the next section.

3.4.3 *Correctness proof for* FILTER

We say that a process p is *in tree* T_m *at level* ℓ if it has started entering $\mathsf{ME}^m_{p,\ell}$ and has not finished releasing $\mathsf{ME}^m_{p,\ell}$. Similarly, a process p is *in tree* T_m *at level* $\lceil \log S \rceil + 1$ if it has acquired and not yet released name m (i.e., if $name_p = m$). We say that a process p is *in tree* T_m if it is in T_m at some level. We assume that there are at most k processes in all the trees at any time. A process p starts a new *round* whenever p attempts to advance in the tree for the first name in N_p. We say a process p is *stuck at round* r *in tree* T_m *at level* ℓ if in two successive rounds $r-1$ and r, checking $\mathsf{ME}^m_{p,\ell}$ at level ℓ in tree T_m returns *false.*

The following lemma states that T_m is indeed a tournament tree.

Lemma 3.6 *For all* ℓ *with* $1 \le \ell \le \lceil \log S \rceil$ *and all* $p, q \in \{0, \ldots, S-1\}$, *if* $\mathsf{ME}^m_{p,\ell} = \mathsf{ME}^m_{q,\ell}$, *then* p *and* q *cannot be in* T_m *at level* $\ell + 1$ *simultaneously.*

Proof Suppose the lemma does not hold. Towards a contradiction pick the minimal ℓ for which there are p, q with $\mathsf{ME}^m_{p,\ell} = \mathsf{ME}^m_{q,\ell}$ that *are* in T_m at level $\ell + 1$ simultaneously.

Then for all ℓ', such that $1 \le \ell' < \ell$, no $p, q \in \{0, \ldots, S-1\}$ with $\mathsf{ME}^m_{p,\ell'} = \mathsf{ME}^m_{q,\ell'}$ are in T_m at level $\ell'+1$ simultaneously. Also, by assumption, at most one process enters a ME at level 1 from any direction. Hence at most two processors p, q, with $\mathsf{ME}^m_{p,\ell} = \mathsf{ME}^m_{q,\ell} = \mathsf{ME}$ concurrently access ME: one with $\beta = 0$, the other with $\beta = 1$. Now the original proof of mutual exclusion can be applied to show that at most one process can win [PF77]. This contradicts the assumption that both are in T_m at level $\ell + 1$. ◁

Lemma 3.7 *If p is stuck in round r in tree T_m at level ℓ, then there is another process $q \neq p$ in tree T_m at the start of the r-th round of p.*

Proof Let ME = $\mathsf{ME}^m_{p,\ell}$, the mutual exclusion at level ℓ in tree T_m played by p. Let us write R_p for the variable written by p in ME, and R'_p for the variable read by p and written by its opponent. Let $\beta = p[\ell]$, be the direction of p. As p is stuck in round r in tree T_m at level ℓ, its check of ME in round r returns *false*. At that time, then, $\mathsf{R}'_p \neq \perp$. Hence, there is a process q that writes R'_p while entering ME before p reads R'_p, and q does not release ME until after p reads R'_p. Such q must, using Lemma 3.6, have direction $1 - \beta$.

As p is stuck in round r, p checks ME in round $r - 1$, which implies that during round r, p (and by Lemma 3.6 no other process either) does not write and thus change R_p, and that $\mathsf{R}_p \neq \perp$. Let r_p be the value of R_p during round r. Suppose q enters T_m after p starts round r. Then upon entering ME it reads $r_p \neq \perp$ from R_p and thus sets R_q to $(1 - \beta) \oplus r_p$. Then if p checks ME in round r, p reads $(1 - \beta) \oplus r_p$ from R_q (which corresponds to R'_p) and evaluates $\beta \oplus (r_p \neq ((1-\beta) \oplus r_p))$ yielding *true* for arbitrary β and r_p, contrary to assumption that p is stuck in round r. Hence, q is in T_m before p starts round r. ◁

The following proposition states that for different processes p and q, there are at most d trees that are tried by both of them. It is an easy consequence of the discussion in Sect. 3.4.1.

Proposition 3.8 *If $p \neq q$, then $|N_p \cap N_q| \leq d$.*

Lemma 3.9 *In each round, every process p can advance to a higher level in at least $d(k-1)$ trees it is trying.*

Proof Let r be an arbitrary round. By assumption at most $k - 1$ processes other than p can be in any tree at the start of round r. Proposition 3.8 then implies that at the start of round r there is a process $q \neq p$ in at most $d(k-1)$ trees p is trying. By Lemma 3.7, p is stuck in round r in at most $d(k-1)$ trees it tries. As p plays $2d(k-1)$ trees per round, the lemma follows. ◁

Theorem 3.10 *Protocol* FILTER *implements wait-free and long-lived renaming to $z2d(k-1)$ names in time $O(dk \log S)$ using $O(zdkS)$ variables.*

Proof By Lemma 3.6, a name obtained by p is not obtained by any other processes for as long as p holds that name. Hence the protocol is a renaming protocol. As there are no restrictions on the execution of the protocol—except for the fact that at most k processes are contending for, or in fact have, a name—processes may repeatedly get and release a name. Hence the protocol is long-lived.

There are D trees, containing roughly $2S$ mutual exclusion blocks ME that each use 2 variables. By Sect. 3.4.1, Eq. (3.3) we have $D = z2d(k-1)$, so FILTER uses $O(zdkS)$ variables[2].

The maximum number of shared variable accesses performed by any process before it gets a name is computed as follows. Because a process p plays in at most $2d(k-1)$ trees, it can advance at most $2d(k-1)\lceil \log S \rceil$ times. Suppose a process p executes a *Check* $4d(k-1)$ times. If it loses $3d(k-1)$ or more of them, then there is a round in which p is stuck in at least $d(k-1)$ trees it is trying, contradicting Lemma 3.9. (If it loses $3d(k-1)$ out of $4d(k-1)$, then $2d(k-1)$ of those were lost immediately, and $d(k-1)$ of those must belong to a single round. Because they were lost immediately, p is stuck at those $d(k-1)$ ME blocks.) We conclude that for every $4d(k-1)$ *Check*'s performed, a process wins at least $d(k-1)$ of them. Then after at most $8d(k-1)\lceil \log S \rceil$ calls of *Check*, each taking 1 shared access, process p obtains a name. A process will *Enter* the $2d(k-1)\lceil \log S \rceil$ mutual-exclusions at most once, at the cost of 4 shared accesses. Clearly, releasing all played mutual exclusion blocks takes no more time than entering them. This shows that the protocol is wait-free, with time complexity $O(dk \log S)$. ◁

3.4.4 *Using* FILTER

In this section, we discuss how the FILTER protocol can be used. First, the choice of d and z given k and S influences its time complexity, its space complexity, and the size of the destination name space. To demonstrate this, we consider various examples of k and S, and show how d and z can be chosen to satisfy the requirements outlined in Sect. 3.4.1. At the end of this section, we discuss how instances of filter (parameterized by d and z) can be combined with each other, and with other long-lived renaming protocols, to successively reduce the size of the destination name space.

---

[2] Actually, because processes contend only for a fraction of the names, certain mutual exclusion blocks may never be accessed. These blocks may safely be removed from the trees, but this will not significantly decrease the space complexity.

In the following paragraphs, we consider various relationships between k and S. For each case, we give nearly optimal choices for d and z that satisfy the requirements set out in Sect. 3.4.1 and give the space and time complexity of the resulting protocol, along with the size of the destination name space.

$S \leq c^k$: Let $d = k$ and z prime such that $2k(k-1)+c \leq z \leq 4k(k-1)+2c$. This satisfies Eq. (3.1) and Eq. (3.2). Substituting d and z in Eq. (3.3) yields $D \leq [4k(k-1)+2c]2k(k-1) \leq 8k^4 + 4ck^2$. The time complexity of the resulting protocol is $O(k^3)$ and the space complexity is $O(k^4c^k)$.

$S \leq 3^{k-1}$: If we set $d = (k-2)/2$ we get, by Eq. (3.2), $z \geq k^2 - 3k + 2$. As for any a there is a prime between a and $2a$ [Che52], we select $k^2 \leq z \leq 2k^2$. Then $z^{d+1} \geq (k^2)^{((k-2)/2)+1} = k^k$. As $k^k \geq 3^{k-1}$ if $k \geq 1$, this satisfies Eq. (3.1), and, by Eq. (3.3), $D \leq 2k^2(k-2)(k-1) \leq 2k^4$. The time complexity of the resulting protocol is $O(k^3)$ and the space complexity is $O(k^4 3^k)$.

$S \leq k^{\log^c k}$: Pick $d = \log^c k$ and z a prime such that $2k \log^c k \leq z \leq 4k \log^c k$. This again satisfies Eq. (3.1) and Eq. (3.2). Substituting d and z in Eq. (3.3) yields $D \leq 8k(k-1)\log^c k \log^c k$. The time complexity of the resulting protocol is $O(k \log^{2c+1} k)$ and the space complexity is $O(k^{2+\log^c k} \log^{2c} k)$.

$S \leq k^c$: Select $d = c$ and let z be a prime such that $2c(k-1) \leq z \leq 4c(k-1)$. This also satisfies Eq. (3.1) and Eq. (3.2). Substituting d and z in Eq. (3.3) yields $D \leq 4c(k-1)2c(k-1)$. The time complexity of the resulting protocol is $O(k \log k)$ and the space complexity is $O(k^{c+2})$.

$S \leq 2k^4$: Let us set $d = 3$. Then $z \geq 6(k-1)$ by Eq. (3.2), so let us pick $6k \leq z \leq 12k$ and z a prime. Then $z^{d+1} \geq (6k)^4 \geq 2k^4$ satisfying Eq. (3.1). In this case, for the size of the destination name-space we find $D \leq 12k6(k-1) \leq 72k^2$. The time complexity of the resulting protocol is $O(k \log k)$ and the space complexity is $O(k^4)$.

This analysis shows that if the size of the source name space is polynomial in k, then FILTER renames to a name space of size $O(c^2k^2)$ in time $O(k \log k)$. Only if the size of the source name space becomes exponential in k will FILTER require $O(k^3)$ time. In all cases, the space complexity is never much bigger than S.

Renaming protocols can be nested in order to repeatedly reduce the size of the name space. To see this, observe that after acquiring a name from a particular long-lived renaming protocol, a process can then use the acquired

name to access another long-lived renaming protocol. The latter protocol can have a source name space the same size as the destination name space of the former. In fact, several long-lived renaming protocols can be chained together in this fashion. The result is a long-lived renaming protocol with the same size of source name space as the first in the chain and the same size of destination name space as the last in the chain. The time and space complexity of the resulting protocol is the sum of the time and space complexities, respectively, of all the protocols in the chain.

To see how this combining approach can be helpful, observe that for $S = k^{\log^c k}$ and $S = c^k$, applying FILTER twice yields $D \in O(k^2)$ with no asymptotic increase in the time or space complexity listed above. Unfortunately, Prop. 3.4 of Erdös *et al.* [EFF85] shows that multiple applications of protocol FILTER will not lead to a name space smaller than $k(k+1)/2$. To actually reach a name space of $k(k+1)/2$ one might apply protocol MA on the last name space obtained, at the expense of increasing the time complexity to $O(k^3)$.

For arbitrary values of S and k a destination name space of $k(k+1)/2$ can be achieved by combining SPLIT, the second and last instances of FILTER given above, and finally Moir and Anderson's protocol MA. This approach yields the following result.

Theorem 3.11 *Long-lived renaming from a source name space of size S to a destination name space of size $k(k+1)/2$ can be achieved with time complexity $O(k^3)$ and space complexity $O(k^4 \min(3^k, S))$.*

3.5 Concluding remarks

As discussed previously, renaming to a smaller name space results in lower overhead. Thus, we are motivated to seek renaming protocols whose destination name spaces are as small as possible. Herlihy and Shavit [HS93] have shown that one-time renaming (and hence long-lived renaming) cannot be solved in a wait-free manner using atomic reads and writes unless $D \geq 2k-1$. For one-time renaming this bound is tight [BG93, MA94]; for long-lived renaming Moir and Garay [MG96] recently showed that, as a final back-end, the Burns and Peterson l-assignment protocol [BP89b] can be used to solve fast and long-lived renaming to $2k-1$ names.

Everybody said 'How-do-you-do' to Eeyore,
and Eeyore said that he didn't,
not to notice, and then they sat down;

A. A. MILNE
The House at Pooh Corner

4 Self-Stabilizing Mutual Exclusion on Directed Graphs

Abstract This chapter investigates the complexity of self-stabilizing mutual exclusion protocols for distributed systems, where processors communicate through shared memory according to a strongly connected directed communication graph of n nodes. Tchuente's approach of covering a network with one directed cycle is taken as point of departure. This protocol requires $O(n^{2n})$ states per processor together with some preprocessing. By coalescing states a protocol requiring only $O(n^3)$ states per processor — still requiring preprocessing — is derived. Finally two protocols based on spanning trees are considered. Combining these protocols with a self-stabilizing spanning tree protocol yields a $O(nD^2)$ and a $O(n^2D)$ states per processor protocol (where D = diam G, the diameter of the graph) that both require knowledge of processor identities.

4.1 Introduction

In a distributed system many processors cooperate to perform certain tasks. A prerequisite for achieving cooperation are protocols that implement distributed control, i.e., protocols that reach or maintain a global objective despite the fact that processors can only access partial, local information. Because distributed control protocols are extensively used to implement distributed systems, many researchers have tried to implement them efficiently. Mutual exclusion protocols constitute a well-know example. These protocols pass a single privilege fairly among the processors in the system. A processor requiring exclusive access to a critical resource must request the privilege and

wait for it to arrive. After a processor is done with the privilege, it forwards the privilege to some other processor requesting it. Many solutions to the mutual exclusion problem exist, but most of them assume that the system operates flawlessly at all times.

One can add failure resilience to distributed control protocols by making them self-stabilizing. In abstract terms a self-stabilizing protocol will, when started on a system in an arbitrary initial configuration, reach a certain desirable — legitimate — configuration after a finite number of steps. The legitimate configurations characterize the global objective that has to be maintained in the system. Once such a legitimate configuration is reached, the protocol will keep the system in a legitimate configuration forever. For a mutual exclusion protocol for instance, the legitimate configurations would be those in which at most one processor holds the privilege. In practice a self-stabilizing protocol is resilient to transient errors that change the state of some processors, but will leave the processors themselves in working order. Since, as far as a self-stabilizing protocol is concerned, the erroneous state just after an error might also have been its initial state, self-stabilizing protocols will recover from such errors.

Most research on self-stabilizing protocols has focussed on systems in which part of the state of a processor can be read by other, neighbouring, processors as if they communicated through shared memory. An important complexity measure in this model is the number of states per processor used by such a protocol. The distributed system is — as usual — modelled by a communication graph containing the processors as vertices and a directed edge between two processors if the state of the first can be read by the second. This graph is considered undirected if for every edge there exists an edge between the same processors in the reverse direction as well. It is assumed that this communication graph is strongly connected.

Self-stabilization was introduced by Dijkstra [Dij74, Dij82] in the context of the mutual exclusion problem. Dijkstra gives an $O(n)$-state[1] self-stabilizing mutual exclusion (SSME) protocol for directed rings of n nodes, a four-state protocol for an undirected chain, and a three-state protocol for undirected rings. All his protocols are non-uniform: the protocol of at least one processor differs from the protocol of the other processors to break the symmetry of the system. He also proved that for rings of non-prime size no uniform SSME protocol can exist. Burns and Pachl [BP89a] complemented this

[1] Unless stated otherwise, an x-state protocol should be read as x-state-per-processor protocol.

result with a uniform SSME protocol for oriented rings of prime size. Using the results of Chapter 5 of this thesis, the same protocol can also be used for un-oriented rings.

Elaborating on Dijkstra's results, Tchuente [Tch81] gives some lower bounds on the number of states per processor for any SSME protocol on a chain or a ring. He also extends Dijkstra's results to SSME protocols for arbitrary graphs in two ways: by covering a graph with a ring[2], and by covering a graph with chains and rings that all share at least one common node. The resulting protocols require knowledge of the coverings, and therefore require some preprocessing.

For undirected graphs, many SSME protocols have been developed. There the focus has been on optimizing the time needed to reach a legitimate configuration and the time needed to pass the privilege from one processor to the other. Of special interest to us is a SSME protocol for arbitrary undirected graphs based on spanning trees, by Dolev *et al.* [DIM93]. They introduce the notion of a fair protocol combination to combine a self-stabilizing spanning tree protocol with a SSME algorithm for tree-shaped graphs to obtain a SSME protocol for arbitrary graphs. The resulting protocol only requires one a priori special node and can even handle topological changes, as long as the diameter and the maximal degree of the graph do not exceed the limits assumed by the protocol.

To summarize: space efficient self-stabilizing mutual exclusion protocols for undirected graphs exist, but for arbitrary directed graphs of n nodes only Tchuente's covering with a directed ring can be used. This requires $\Omega(n^{2n})$ states per processor in the worst case. In the following, let D be the diameter of the graph. Our contribution is twofold. First we reduce the number of states needed to cover a graph with a virtual ring to $O(n^3)$ by coalescing states on the processors. Recall that this cover still needs to be computed beforehand. Secondly, we present two new protocols, the first using $O(nD)$ and the second using $O(n^2)$ states per processor, that do not require preprocessing, but do require that processors know their identities. Instead of a virtual ring, these protocols use a spanning tree which can be maintained by a separate self-stabilizing protocol requiring an additional $O(D)$ states per processor. Merging both protocols using fair protocol combination yields a $O(nD^2)$ and a $O(n^2D)$ states per processor protocol for self-stabilizing mutual exclusion on arbitrary, strongly connected, directed graphs.

[2] Here a ring is a cyclic path in the graph that may visit nodes more than once, and may traverse edges more than once.

The structure of this chapter is as follows. We begin by describing the model, give a precise statement of the problem and introduce some notation in Sect. 4.2. Section 4.3 discusses Tchuente's results from [Tch81], and is followed by Sect. 4.4, in which we improve his solution of covering a graph with a ring. Our protocols based on a spanning tree are presented in Sect. 4.5. Finally, Sect. 4.6 discusses our results, and suggests ways of further research. A full proof of a slightly different version of the coalesced states protocol in Lamport's Temporal Logic of Actions can be found in Alstein *et al.* [AHO+94a].

4.2 Model and notation

We consider arbitrary strongly connected distributed systems modelled by a directed communication graph $G = (V, E)$, with *nodes* $v \in V = \{0, \ldots, n-1\}$ and directed *edges* $uv \in E \subseteq V \times V$. Let $D = \operatorname{diam} G$ be the diameter of graph G. If $uv \in E$, then u is an *in-neighbour* of v and v an *out-neighbour* of u. Internally, nodes can distinguish between neighbours. We define the set of in-neighbours $\mathsf{In}(v) = \{u \mid uv \in E\}$. Nodes in the system communicate with each other by reading each others state. Node v can read the state of node u if and only if $uv \in E$.

A *configuration* C of the system is the Cartesian product $\prod_{v \in V} s_v$ of *states* s_v of all nodes $v \in V$. We write $C[v]$ for the state of node v in configuration C. A *program* for a node v describes the steps it can perform, based on the state of v and the state of its in-neighbours $\mathsf{In}(v)$. We model a program for a node v as a deterministic state-transition function δ_v (mapping the state of v and the states of all its in-neighbours to the new state of v). A *step* of v in configuration C changes the state of v to $\delta_v(C) = \delta_v(C[v], (C[u])_{u \in \mathsf{In}(v)})$, yielding configuration C' with $C'[v] = \delta_v(C)$ and $C'[u] = C[u]$ for all $u \neq v$. We write $C \to_v C'$. A *protocol* consists of a program for each node $v \in V$.

All protocols assume the existence of a *central daemon* [Dij74]. In this model, nodes take a step one at a time according to a fair *schedule* $(v_0 v_1 \ldots)$, such that each $v \in V$ occurs infinitely often in the schedule. An *initial configuration* C_0 and a schedule $(v_i)_{i \geq 0}$ induce an *execution* $(C_0 C_1 \ldots)$ such that $C_i \to_{v_i} C_{i+1}$ for all $i \geq 0$. This execution is fair iff the schedule is fair. We consider normalized executions from which all void steps $C_i \to_{v_i} C_{i+1}$ with $C_i = C_{i+1}$ have been removed.

Let $\mathcal{L}$ be the set of *legitimate configurations*, i.e., the set of configurations the distributed systems should be in. Then a protocol is *self-stabilizing* to $\mathcal{L}$ if for all executions $C_0 C_1 \ldots$ there exists an i such that $C_i \in \mathcal{L}$, and moreover, for all $C' \in \mathcal{L}$ and C'' with $C' \to C''$, we have $C'' \in \mathcal{L}$ as well. In other

words, a protocol is self-stabilizing to $\mathcal{L}$ if in every execution a configuration in $\mathcal{L}$ is reached, and if $\mathcal{L}$ is closed under transitions of the protocol.

The performance of a self-stabilizing protocol is usually measured by the number of states per processor used by the protocol, and the rate of convergence, i.e., the worst case number of non-void steps needed to reach a legitimate configuration. The state-per-processor measure does not count any overhead incurred by communicating the state to the neighbour. This cost is considered part of the topology. Of course, any critical assessment of a self-stabilizing protocol should also consider the complexity measures related to the original problem to be solved by the protocol. For self-stabilizing mutual exclusion protocols this would include a bound on the time a processor may have to wait before it receives the privilege.

The problem to be solved can now be stated as follows: design a protocol, self-stabilizing to a set $\mathcal{L}$ of legitimate configurations in which at most one node is privileged. Each node should be able to determine whether it is privileged or not based on its own state and that of its in-neighbours. A further requirement is that during an execution of the protocol, each node gets privileged infinitely often.

We use the following notational conventions. The state of a processor is split into several named fields. A field *name* of the state of v is written $name_v$. To distinguish the value of this field in different configurations C^a and C^b we write $name_v^a$ for its value in C^a and $name_v^b$ for its value in C^b; its value in configuration C is simply written as $name_v$. The program δ_v for a node v is denoted as a sequence of statements to transform the state of v to its new state.

4.3 Tchuente's approach

Tchuente [Tch81] extends Dijkstra's results to SSME protocols for arbitrary undirected graphs in two ways: by covering a graph with a ring, and by covering a graph with chains and rings that all share at least one common node. The resulting protocols require knowledge of the coverings, and therefore require some preprocessing. Some of Tchuente's ideas are applicable to directed graphs as well.

The first approach is based on the observation that every strongly connected directed graph can be covered by a 'virtual' ring. This virtual ring G_R is a directed path of edges in the original graph G visiting each node of this graph at least once. Hence neighbours in G_R are neighbours in the original graph G as well. Then a SSME protocol for the general graph G is obtained by

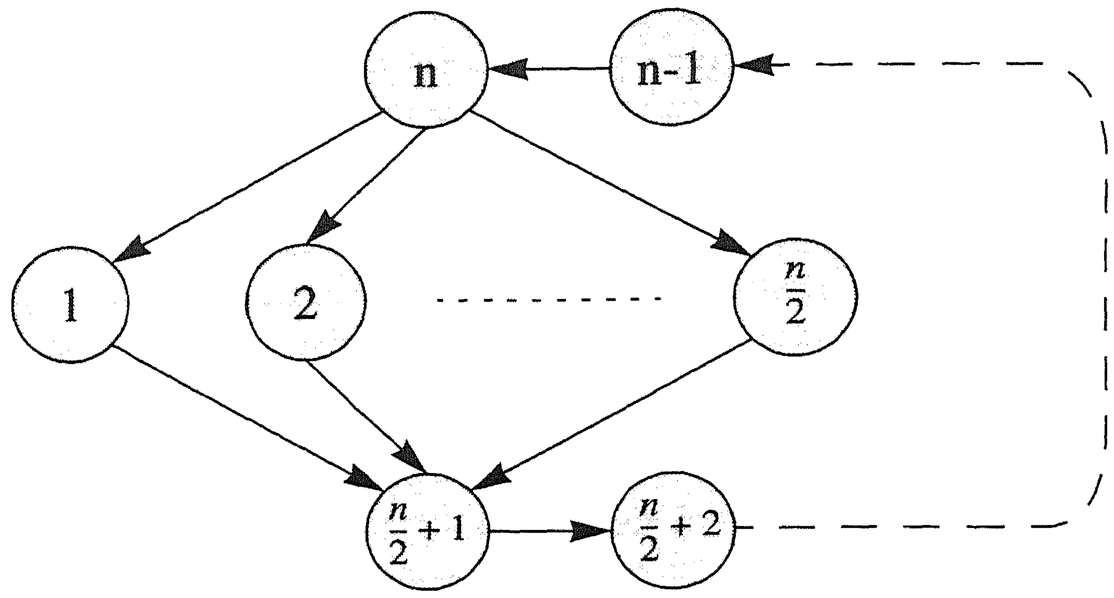

Figure 4.1 *Worst case graph with $\Omega(n^2)$ virtual ring.*

simulating Dijkstra's SSME protocol for the ring G_R on G as follows. Each node $v \in G$ simulates (i.e., executes the program of) each of its occurrences on the ring G_R in separate *simulation instances*, each using a separate field of the state of v. In this simulation, if a node wishes to read the state of a neighbour in G_R, the whole state of the simulating neighbour in G will be read, after which the appropriate field will be selected. A node $v \in G$ is privileged whenever one of the nodes it simulates is privileged.

The main drawback of this approach is that there are graphs over n nodes which cannot be covered by rings of length less than $\Omega(n^2)$ nodes. A worst case example is presented in Fig. 4.1. There, the virtual ring in G is $(1,\ldots,n,2,\ldots,n,\ldots,n/2,\ldots,n,1)$, containing $n/2(n/2+1)$ nodes. Because the diameter of a strongly connected graph over n nodes is at most $n-2$ (counting the number of intermediate nodes), it is clear that there always exists a virtual ring of at most $n(n-2)+n$ nodes.

This is particularly bad if we consider the number of states used by this protocol in the worst case. The state of a node v in graph G should represent the states of all its simulating instances. Because the length of the ring is $\Omega(n^2)$, each of this simulating instances must — according to Dijkstra's protocol — have $\Omega(n^2)$ states. Moreover, on average, each node must simulate $\Omega(n)$ nodes. Hence, on average, each node uses at least $\Omega(n^{2n})$ states.

Tchuente's second approach of covering a graph with both rings and chains is similar in spirit to the method just described. There, the graph is covered by rings and chains that all visit one particular central node. Let us focus on the case where the graph is covered by rings only. Then this central

node simulates the root node for each and every covering subring. Nodes again simulate the protocol for each occurrence on these subgraphs. Because these subgraphs run a SSME protocol, eventually a token will circulate in every subgraph. To ensure that only one node is privileged in the original graph, the central node will only forward a token on one particular subgraph if it holds all tokens for all subgraphs. So it will, for every root it simulates, run the original protocol, except that it will only allow one of these roots to take a step if all roots it simulates are privileged according to the simulated protocol.

This construction reduces the number of states used, but still leaves it exponential in n in the worst case. Again consider the graph in Fig. 4.1. Instead of a single covering ring of length $\Omega(n^2)$ we now need n rings of length $n/2$, with node n being the central node. Each of these rings need $\Omega(n)$ states per processor, and again half the nodes simulate $\Omega(n)$ rings. Hence, on average, each node uses $\Omega(n^n)$ states.

4.4 Reducing the number of states

Tchuente's approach of covering a graph with a ring and running Dijkstra's SSME protocol on that covering was shown to use $\Omega(n^{2n})$ states per processor in the worst case. In this section we present a $O(n^3)$ states per processor protocol by coalescing the states of Dijkstra's protocol for each group of nodes on the ring that are simulated by a single node in the graph. Instead of using separate state-spaces for all simulating instances of a node, these simulating instances will now share the state among them.

In Dijkstra's original SSME protocol (presented in Prot. 4.1) for directed rings of n processes, each processor v uses a single variable cur_v storing numbers in $\{0, \ldots, n-1\}$. Nodes v copy the number from their anti-clockwise (left) neighbour $v-1$, except for the root 0. The root increments its number (modulo n) whenever its number equals that of its anti-clockwise neighbour $n-1$. In the legitimate configurations there is one node such that all nodes from the root up to and including this node hold the same number as the root, and all nodes beyond this node (in clockwise direction) hold the number of the root minus 1 (modulo n). The clockwise (right) neighbour of this particular node then is the only privileged node. Self-stabilization of this protocol can be shown as follows. If the root cannot take a step, there is at least one node with a number unequal to its left neighbour which therefore can take a step. If the root takes a step, its number equals that of its left neighbour, and because there are at least n different values stored as numbers, one of these values, say a, does not occur anywhere in the ring. Observe from

Node $v = 0$

if $cur_{n-1} = cur_0$

then $cur_0 := (cur_0 + 1) \bmod n$

Node $v \neq 0$

if $cur_{v-1} \neq cur_v$

then $cur_v := cur_{v-1}$

Nodes are privileged whenever their guard is true.

Protocol 4.1 *Dijkstra's SSME protocol for a ring of n processes.*

the protocol that the root is the only node introducing new values in the ring. Then if, after a while, the root assumes the value a (or a similar 'new' value), it will not advance until after this value has propagated along the ring. When that happens, all nodes hold the value a. Then the configuration is legitimate, and is easily seen to remain legitimate.

Our reduction of the number of states is based on the observation that in the legitimate states of this protocol, there are only two different numbers held by each node. Then a node simulating k nodes of the virtual ring only needs to store the current value of the root, and an index ranging from $\{1,\ldots,k\}$ indicating the last simulated node holding the current value of the root. This implicitly encodes that all other nodes hold the previous value of the root.

The implementation Let $G = (V, E)$ be an arbitrary strongly connected directed graph, and let $G_R = (V_R, E_R)$ be the virtual ring of n_R nodes covering G as described in the previous section. Label the nodes on G_R with labels taken from the set $\{0,\ldots,n_R - 1\}$ such that the left neighbour of 0 is $n_R - 1$, and the left neighbour of any other node v is $v - 1$. Let $f : V_R \mapsto V$ be the function that for each node v of the virtual ring G_R returns the node of the original graph G simulating v (i.e., the node covered by v). G_R must cover G, so the range of f equals V. Define $\mathsf{Sim}(v) \triangleq \{v_R \in V_R \mid f(v_R) = v\}$, the nodes on the virtual ring simulated by v, and define $\mathsf{Prev}(v)$ to equal the biggest node $u < v$ such that $f(u) = f(v)$. Let $\mathsf{Prev}(v) = \perp$ if v itself is the smallest element of $\mathsf{Sim}(f(v))$. It is assumed that each node in the graph knows the covering. In other words, $\mathsf{Sim}(v)$ (and thus also f and Prev) is stored in stable

Node $v = 0$

if $num'_{n_R-1} = num'_0 \wedge valid'_{n_R-1} \geq n_R - 1$

then $num'_0 := (num'_0 + 1) \bmod n_R$

$valid'_0 := 0$

$cur_0 := num'_0$

Node v with $\mathsf{Prev}(v) = \perp$

if $num'_{v-1} \neq num'_v \wedge valid'_{v-1} \geq v - 1$

then $num'_v := num'_{v-1}$

$valid'_v := v$

$cur_v := num'_v$

Node v with $\mathsf{Prev}(v) \neq \perp$

if $num'_{v-1} = num'_v \wedge valid'_{v-1} \geq v - 1 \wedge valid'_v = \mathsf{Prev}(v)$

then (* $num'_v = num'_{v-1}$ *)

$valid'_v := v$

$cur_v := num'_v$

Nodes $v = 0$ or nodes v with $\mathsf{Prev}(v) = \perp$ are privileged whenever their guard is true.

Protocol 4.2 *The coalesced states protocol.*

storage and is accessible to v.

The protocol appears in Prot. 4.2. Each node $v \in V$ stores a number $num_v \in \{0, \ldots, n_R - 1\}$ and a pointer $valid_v \in \mathsf{Sim}(v)$. Both variables are shared among all nodes simulated by v. We define

$$num'_v \triangleq num_{f(v)}, \qquad valid'_v \triangleq valid_{f(v)}.$$

The protocol describes the possible steps for a *simulated* node $v \in V_R$. cur_v is a *ghost variable* kept for each simulated node $v \in V_R$. Because it is only a ghost variable it does not occupy any real space[3]. Whenever a node $v \in V$ is scheduled to take a step it fairly picks one of the nodes it simulates and takes the appropriate step as described in Prot. 4.2.

Protocol 4.2 is slightly different from the one that appeared else-

[3] Just like 'real' ghosts.

where [AHO+94a, AHO+94b], which uses only $O(n^2)$ states per processor. To simplify the proof of correctness here, we have stretched the range of values for *num* from $\{0,\ldots,n\}$ to $\{0,\ldots,n_R\}$ and have added the extra condition $valid'_v = \mathsf{Prev}(v)$ to the third rule (for v with $\mathsf{Prev}(v) \neq \perp$). With these modifications we get an $O(n^3)$ states per processor with a slightly stronger synchronization, whose behaviours correspond one-to-one to the behaviours of Dijkstra's original protocol (Prot. 4.1). Together with a proof that the protocol cannot deadlock, this easily establishes that Prot. 4.2 is a SSME protocol.

Proof of correctness Variable cur_v is a ghost variable that corresponds to the number stored by node v in Dijkstra's original SSME protocol (Prot. 4.1, with n replaced with n_R). Because *cur* does not influence the flow of control itself, if we show the protocol correct for one particular initial setting of cur_v, we have shown it correct for every possible initial setting of cur_v. Therefore we can assume that, initially, $cur_v = num'_v$.

Lemma 4.1 *Protocol 4.2 does not deadlock.*

Proof We will first show by induction that if the protocol deadlocks, then the following equation holds for $v = n_R - 1$.

$$(\forall u : u \leq v :: valid'_u \geq u \ \wedge \ num'_u = num'_0) \ . \tag{4.1}$$

Clearly, Eq. (4.1) holds for $v = 0$. Now suppose Eq. (4.1) holds for $v = w$. We will show it also holds for $v = w + 1$. There are two cases.

Case 1, $\mathsf{Prev}(w+1) = \perp$: Then $w+1$ is the smallest value in $\mathsf{Sim}(f(w+1))$. As $valid'_{w+1} \in \mathsf{Sim}(f(w+1))$, $valid'_{w+1} \geq w+1$.

Also by the induction hypothesis (4.1) $valid'_w \geq w$. By Prot. 4.2, if w cannot take a step, then $num'_w = num'_{w+1}$. Using Eq. (4.1), $num'_{w+1} = num'_0$.

Case 2, $\mathsf{Prev}(w+1) \neq \perp$: Then by Eq. (4.1), $num'_{\mathsf{Prev}(w+1)} = num'_0$. By definition of num', $num'_{\mathsf{Prev}(w+1)} = num'_{w+1}$ and so $num'_{w+1} = num'_0$.

Also by Eq. (4.1), $valid'_w \geq w$. Using the previous paragraph and Eq. (4.1), also $num'_{w+1} = num'_w$. By Prot. 4.2, if w cannot take a step, then $valid'_{w+1} \neq \mathsf{Prev}(w+1)$. By Eq. (4.1), $valid'_{\mathsf{Prev}(w+1)} \geq \mathsf{Prev}(w+1)$. By definition of $valid'$, $valid'_{w+1} = valid'_{\mathsf{Prev}(w+1)}$. Hence $valid'_{w+1} \geq w+1$.

But now, using Eq. (4.1), $num'_{n_R-1} = num'_0 \ \wedge \ valid'_{n_R-1} \geq n_R - 1$ if the protocol deadlocks. Hence node 0 *can* take a step. ◁

Lemma 4.2 *In Prot. 4.2, the following assertion is invariant:*

$$cur_v = num'_v \quad \text{if} \quad valid'_v \geq v \ . \tag{4.2}$$

Proof By assumption the invariant is initially true. If $w = 0$ or w with $\mathsf{Prev}(w) = \perp$ takes a step, the invariant remains true because in both cases we set $valid'_v \geq v$ only for $v = w$ (as for u with $f(u) = f(w)$, we have $u > w$), and we set $cur_w = num'_w$ as required. If w with $\mathsf{Prev}(w) \neq \perp$ takes a step, then $valid'_w$ changes from $\mathsf{Prev}(w)$ to w possibly (only) invalidating the invariant for $v = w$. But for $v = w$ we set $cur_v = num'_v$ as required. ◁

The next lemma shows that each execution of Prot. 4.2 corresponds to an execution of Dijkstra's original SSME protocol on the ring V_R using cur_v as the number held by node v.

Lemma 4.3 *All executions of Prot. 4.2 correspond to an execution of Prot. 4.1, with* $n = n_R$.

Proof For each step of Prot. 4.2 we verify that there is a corresponding step in Prot. 4.1:

Case 1, $v = 0$ *takes a step*: By Lemma 4.2 then $cur_{n_R-1} = cur_0$ and after the step $cur_0 = (cur_{n_R-1} + 1) \bmod n_R$.

Case 2, v *with* $\mathsf{Prev}(v) = \perp$ *takes a step*: By Lemma 4.2 then $cur_{v-1} \neq cur_v$ and after the step $cur_{v-1} = cur_v$.

Case 3, v *with* $\mathsf{Prev}(v) \neq \perp$ *takes a step*: If $cur_v \neq num'_v$, by Lemma 4.2 and Prot. 4.2 we have $cur_{v-1} \neq cur_v$ and after the step $cur_{v-1} = cur_v$. If $cur_v = num'_v$, the step is void.

In the third case a void step is only applicable once after the node w with $f(w) = f(v)$ and $\mathsf{Prev}(w) = \perp$ takes a non void step. This shows that every infinite execution of Prot. 4.2 corresponds to an infinite execution of Dijkstra's SSME protocol. ◁

Theorem 4.4 *Protocol 4.2 implements self-stabilizing mutual exclusion on arbitrary graphs.*

Proof By Lemma 4.1 the protocol does not deadlock. By lemma 4.3 all executions correspond to an execution of Dijkstra's original SSME protocol on the ring V_R using cur_v as the number held by node v.

Clearly for each node $v \in V$ there is a $w \in V_R$ with $f(w) = v$ and either $\mathsf{Prev}(w) = \perp$ or $w = 0$. By definition, v is privileged iff w is. If $w = 0$ is privileged, then $num'_{n_R-1} = num'_0 \wedge valid'_{n_R-1} \geq n_R - 1$. By Eq. (4.2) then $cur_{n_R-1} = cur_0$, and hence $w = 0$ is privileged in Dijkstra's protocol too. If w with $\mathsf{Prev}(w) = \perp$ is privileged, then $num'_{v-1} \neq num'_v \wedge valid'_{v-1} \geq v - 1$. By Eq. (4.2) then $cur_{v-1} \neq cur_v$ and again w is privileged in Dijkstra's protocol.

This shows that in every legitimate configuration of Dijkstra's original protocol, in the corresponding configuration in Prot. 4.2 only one node $v \in V$ is privileged. ◁

4.5 Two protocols based on spanning trees

Although the protocol of the previous section is very space-efficient, it has one major drawback in that it requires preprocessing to compute the covering of the graph. In this section we lift this restriction and present two SSME protocols, the first using $O(nD^2)$ states and the second using $O(n^2D)$ states per processor, both for arbitrary, strongly connected, directed graphs G with n nodes, and diameter D.

In the following two sections we will describe two SSME protocols, assuming that we are given a stable spanning tree for the graph. Several self-stabilizing spanning-tree protocols have been published, for instance by Afek *et al.* [AKY90] for undirected graphs, and Dolev *et al.* [DIM93] for arbitrary communication graphs. Combining this second protocol with one of the SSME protocols described below using fair protocol combination [DIM93] will yield a SSME protocol for arbitrary strongly connected directed graphs. This composition increases the state complexity by a factor of $O(D)$. For details on both the spanning-tree protocol and fair protocol combination we refer to Sect. 4.5.3 and Dolev *et al.* [DIM93]. Throughout this section we assume that each node has access to its identity $u \in \{0, \ldots, n-1\}$ which is stored in some storage resilient to transient errors.

4.5.1 The first protocol

The central idea in the first SSME protocol is to use the root of the spanning tree as coordinator. The root will continually generate numbers in the range $\{0, \ldots, n-1\}$ in cyclic order. This number indicates the next node to become

privileged. All non-root nodes cooperate to pass this number down the spanning tree. Whenever a non-root node discovers that its identity is equal to the number held by its parent, it becomes privileged until it stores this number in its own state.

To ensure that only one node is privileged at a time, the root is only allowed to generate a new number if it can be sure that all nodes hold the same number. Note that in that case the node with identity equal to the number held by the root has already used its privilege. Inspection of the numbers held by all incoming nodes of the root alone will not guarantee this, because the root may not be able to read all leaves of the spanning tree directly. To allow the root to make the correct decision, all nodes are required to express their trust in the number they hold. For a node holding a certain number, its trust roughly corresponds to the length of the shortest path to this node starting from an arbitrary node not holding this number. The root always expresses the maximal trust for any number it holds. Then once the root sees its number on all its incoming nodes, all with maximal trust, it is certain that all nodes hold the same number. Now it can generate the next number.

The implementation In the protocol we assume that one node r is the root of the spanning tree. The identity of this node is determined beforehand, for example picking the node with identity 0. All non-root nodes u have access to their parent $\mathsf{P}(u)$ in the tree. For the purpose of this section it is assumed that this information is also stored in some non-volatile storage, but it will actually be maintained by a separate self-stabilizing spanning tree protocol (as discussed previously).

The mutual exclusion protocol appears in Prot. 4.3. In this protocol the externally visible state — i.e., the one readable by neighbouring processors — of each processor u is divided into two fields: num_u with values in $\{0, \ldots, n-1\}$, and $trust_u$ with values in $\{0, \ldots, D-1\}$. A node is privileged iff the corresponding predicate of Prot. 4.3 evaluates to true in the current configuration. A privileged node can choose to access its critical resource before taking the next step in the SSME protocol, in which case it loses its privilege.

Proof of correctness To prove correctness we proceed as follows. First we give a precise characterization of the legitimate configurations. We show that in these configurations the mutual exclusion property is satisfied, and that the privilege is passed fairly among all processors in the system. Then we show that once the system has reached a legitimate configuration, it will remain in a legitimate configuration as long as no further errors occur. We prove that

---

Node r

if $(\forall v \in \mathsf{In}(r) :: num_v = num_r \ \wedge \ trust_v = D - 1)$
then $num_r := (num_r + 1) \bmod n$
$trust_r := D - 1$

Privileged if $(\forall v \in \mathsf{In}(r) :: num_v = num_r = r \ \wedge \ trust_v = D - 1)$

Node $u \neq r$

$num_u := num_{\mathsf{P}(u)}$
if $(\forall v, w \in \mathsf{In}(u) :: num_v = num_w)$
then $trust_u := \min(D - 1, (trust_v + 1)_{v \in \mathsf{In}(u)})$
else $trust_u := 0$

Privileged if $num_{\mathsf{P}(u)} = u \neq num_u$

Protocol 4.3 *The first mutual exclusion protocol.*

---

the system is indeed self-stabilizing, by showing that the system will reach a legitimate configuration when started from an arbitrary initial configuration.

Recall that as a notational convention we write num_u^a for the value of num_u in configuration C^a. For the value of num_u in configuration C we simply write num_u. Define

$$u \equiv r \quad \triangleq \quad u \neq r \ \wedge \ num_u = num_r \ .$$

A configuration C is legitimate if it satisfies the following three conditions for each node u. First, the number held by u must equal the one held by r, or one less. If u holds the same number as r, then so must its parent and therefore all other nodes on the path from r to u in the spanning tree.

$$\begin{aligned} \mathsf{Proper}(u) \quad \triangleq \quad & num_u \in \{num_r, (num_r - 1) \bmod n\} \\ & \wedge \ (u \equiv r \Rightarrow num_{\mathsf{P}(u)} = num_r) \ . \end{aligned}$$

Second, if u and all its incoming nodes hold the same number as the root r, then the trust expressed by u should not be exaggerated.

$$\begin{aligned} \mathsf{Modest}(u) \quad \triangleq \quad & [(\forall v \in \mathsf{In}(u) :: num_v = num_r) \ \wedge \ u \equiv r] \\ & \Rightarrow trust_u \leq \min_{v \in \mathsf{In}(u)} trust_v + 1 \ . \end{aligned}$$

Third, if u holds the same number as the root r while some of its incoming nodes do not, the trust expressed by u should be 0.

$$\begin{aligned} \mathsf{Honest}(u) \quad \triangleq \quad & [(\exists v \in \mathsf{In}(u) :: num_v \neq num_r) \ \wedge \ u \equiv r] \\ & \Rightarrow trust_u = 0 \ . \end{aligned}$$

Note that according to these definitions $\mathsf{Proper}(r)$, $\mathsf{Modest}(r)$ and $\mathsf{Honest}(r)$ hold for the root r in any configuration C.

Our first proposition states that if the root advances in a legitimate configuration, all nodes hold the same value just prior to that step.

Proposition 4.5 *If* $(\forall v \in \mathsf{In}(r) :: num_v = num_r \ \wedge \ trust_v = D - 1)$ *for a legitimate configuration* C, *then* $(\forall v \in V :: num_v = num_r)$.

Proof By contradiction. Suppose there is a node $u \in V$ with $num_u \neq num_r$. Clearly $u \notin \mathsf{In}(r)$. Consider the shortest path $(v_0, v_1, \cdots, v_k, r)$ from v_0 to r such that $num_{v_0} \neq num_r$. Then $1 \leq k \leq D - 1$. Scuh v_0 exists by assumption; the path to r exists because the graph is strongly connected. Now by $\mathsf{Honest}(v_1)$ we have $trust_{v_1} = 0$, and for all $j > 1$ we have $trust_{v_{j+1}} \leq trust_{v_j} + 1$ by $\mathsf{Modest}(v_{j+1})$. Then $trust_{v_k} < k \leq D - 1$, and so for some $v \in \mathsf{In}(r)$ we have $trust_v \neq D - 1$. This is a contradiction, and the proposition follows. ◁

That our protocol is indeed a mutual exclusion protocol in legitimate configurations follows from the next lemma.

Lemma 4.6 (safety) *In any legitimate configuration* C, *at most one processor is privileged.*

Proof Let C be legitimate and take any node $u \neq r$. If u is privileged then $num_u \neq num_{\mathsf{P}(u)} = u$. By $\mathsf{Proper}(u)$ this is only possible if $num_{\mathsf{P}(u)} = num_r$ and $num_u \neq num_r$. We conclude that $u = num_r \neq num_u$. This shows that no other node $v \neq r$ can be privileged in C.

On the other hand $(\forall v \in \mathsf{In}(r) :: num_v = num_r \ \wedge \ trust_v = D - 1)$ if the root is privileged. Then by Prop. 4.5 we have $(\forall v \in V :: num_v = num_r)$, and so (by the previous paragraph) no node $u \neq r$ can be privileged. ◁

The next lemma implies that the protocol is deadlock-free and all its executions are infinite.

Lemma 4.7 (progress) *In any configuration C at least one processor can take a non-void step.*

Proof Let node u hold the minimal *trust* in configuration C. Then for all $v \in \mathsf{In}(u)$, $trust_v \geq trust_u$. If $trust_u < D - 1$, then if u (whether it equals r or not) takes a step $C \rightarrow C'$, then $trust_u \neq trust'_u$ and u would be able take a non-void step. If $trust_u = D - 1$ then, by minimality, for all nodes v we have $trust_v = D - 1$. If for some node $u \neq r$, $num_u \neq num_{\mathsf{P}(u)}$ then u is able to take a non-void step. If for all nodes $u \neq r$, $num_u = num_{\mathsf{P}(u)}$, then for all nodes u, $num_u = num_r$. Together with $trust_u = D - 1$ this implies that the root r is able to take a non-void step. ◁

Because we consider only fair executions, all processors take infinitely many steps in such an infinite execution. That the protocol is also fair with respect to passing the privilege among the processors is established by the next lemma.

Lemma 4.8 (fairness) *For every execution, if in a legitimate configuration C node u is privileged, then the next privileged node will be $(u + 1) \bmod n$.*

Proof If u is privileged in C, then by the proof of Lemma 4.6, $num_r = u$. The next number assumed by the root will be $(u + 1) \bmod n$, so only node $(u + 1) \bmod n$, if any, can become privileged. According to Prop. 4.5, just prior to this step of the root all nodes have $num_v = u$. If the root advances once more (to $(u + 2) \bmod n$), again by Prop. 4.5 just prior to that all nodes have $num_v = (u + 1) \bmod n$. Then in between those two steps of the root, node $(u + 1) \bmod n$ changes value from u to $(u + 1) \bmod n$ and becomes privileged in the process. ◁

Note that we can weaken the privilege-condition of r to

$$(\forall v \in \mathsf{In}(r) :: num_v = num_r \ \wedge \ trust_v = D - 1)$$

without violating the safety-property. In this case the root will be privileged every time it generates a new number. Then between the time that u is privileged and the time that $(u + 1) \bmod n$ is privileged, the root will be privileged as well (unless $(u + 1) \bmod n$ happens to equal r).

The next lemma states that once the system reaches a legitimate configuration, it will remain in a legitimate configuration for as long as no transient errors occur.

Lemma 4.9 (closure) *For any legitimate configuration C, if $C \to C'$ then C' is legitimate.*

Proof We split the proof in two cases.

Case 1, *Node r takes a step*: If $num_r = num'_r$, then in configuration C' clearly Proper(u) and Honest(u) still hold for all $u \in V$, and Modest(v) still holds for all v with $r \notin \mathsf{In}(v)$. For those w with $r \in \mathsf{In}(w)$, notice that r sets its *trust* to $D - 1$, and so still $trust'_w \leq trust'_r$ which shows Modest(w) for all those w as well.

If $num_r \neq num'_r$, $(\forall v \in \mathsf{In}(r) :: num_v = num_r \wedge trust_v = D - 1)$ and $num'_r = (num_r + 1) \bmod n$, by the protocol. Using Prop. 4.5 we arrive at $(\forall v \in V :: num_v = num_r)$. Then $(\forall v \neq r :: num'_v = (num'_r - 1) \bmod n)$ and thus in C' Proper(u) holds for all $u \in V$ and trivially Honest(u) and Modest(u) for all $u \in V$.

Case 2, *Node $v \neq r$ takes a step*: It is easily checked that in C', Proper(u) holds for all $u \in V$ and Modest(u) and Honest(u) hold for all u with $v \notin \mathsf{In}(u)$ or $num_u \neq num_r$. So it remains to show that Modest(u) and Honest(u) also hold for all u with $v \in \mathsf{In}(u)$ and $num_u = num_r$.

If $(\exists w \in \mathsf{In}(u) :: num_w \neq num_r)$, then by Honest($u$) also $trust_u = 0$. Then also $trust'_u = 0$ and trivially Honest(u) and Modest(u) in C'.

If $(\forall w \in \mathsf{In}(u) :: num_w = num_r)$, then by Modest($u$) and the fact that $v \in \mathsf{In}(u)$ we have $trust_u = trust'_u \leq trust_v$. Moreover, $num_v = num_r$ and thus $num_{\mathsf{P}(v)} = num_r$ by Proper(v), so $num'_v = num'_r$ by the protocol. This, in turn, implies $(\forall w \in \mathsf{In}(u) :: num'_w = num'_r)$ and thus Honest(u) in C'. By Modest(v), $trust_v \leq \min_{w \in \mathsf{In}(v)} trust_w + 1$. If v takes a step, then by the protocol $trust'_v = \min_{w \in \mathsf{In}(v)} trust_w + 1$. Hence $trust'_v \geq trust_v$ which shows that Modest(u) in C'. ◁

Lemma 4.10 (self-stabilization) *For any execution of the protocol starting in an arbitrary configuration C, the execution will eventually reach a legitimate configuration C'.*

Proof Start the protocol in configuration C, and wait until the root takes a step for the first time in configuration C^1 (this must eventually happen because the protocol does not deadlock by Lemma 4.7). In C^1 all $v \in \mathsf{In}(r)$ have $num_v = num_r$. Because there are n nodes and at least one $v \in \mathsf{In}(r)$, at least one value in $\{0, \ldots, n-1\}$ cannot occur on any node in C^1. Take the first

such value following num_r, and call this value a. Continue the protocol until r sets $num_r = a$ reaching configuration C^2. Because non-root nodes only copy existing values, in C^2 the root is the only node with $num_v = a$.

Now continue the execution until we reach a configuration C^3 in which the root advances once more, setting $num_r = (a+1) \bmod n$, and reaching configuration C'. We will show that C' is legitimate. First observe that $(\forall v \in \mathsf{In}(r) :: num_v^3 = a \ \wedge \ trust_v^3 = D-1)$. Also observe that, since in C^2 no non-root node has $num_v = a$, all nodes with $num_v = a$ take a step somewhere between C^2 and C^3. Now let $num_u^3 \neq a$ for some node u. Then for any configuration $C^2 \Rightarrow C \Rightarrow C^3$ we must have $num_u \neq a$ (because once $num_u = a$, all nodes on the path from the root to u have $num_v = a$ and hence $num_u = a$ at least until the root changes value). Again, like the proof of Prop. 4.5, consider the shortest path $u, v_1, \ldots, v_k, r$, with $k \leq D-1$, $v_i = \mathsf{P}(v_{i+1})$, and all intermediate nodes holding number a. Because $num_{v_1} = a$, it takes a step, setting $trust_{v_1} = 0$. Then $trust_{v_2} \leq 1$ because v_2 also takes a step after v_1 does, to copy a from v_1. By induction, $trust_{v_k} < k \leq D-1$. But then for some $u \in \mathsf{In}(r)$ we have $trust_u^3 < D-1$, contrary to assumption.

We conclude that $(\forall v \in V :: num_v^3 = a)$. This shows that in C' we have $num_r = (a+1) \bmod n$ and $(\forall v \neq r \in V :: num'_v = a)$, from which legitimacy of C' easily follows. ◁

4.5.2 The second protocol

In this section we present another protocol, also based on a spanning tree. The main difference between both protocols is that in legitimate configurations, the second protocol does not necessarily require the cooperation of all nodes in order to transfer the privilege from one node to another. In other words, it has a less strict synchronization. The downside of the comparison is that it is slightly less space-efficient, using $2n^2$ states per node.

Like the protocol in the previous section, this protocol works by distributing a number through a spanning tree. The number indicates the identity of the node that should get the privilege. When that node has consumed the critical section, it makes this fact known by setting a (boolean) flag. This flag can be seen as an acknowledge. When the root node finds that the flag has been set, it chooses the next number.

The number is distributed by letting each non-root node copy the number from its parent in the tree. Again, whenever a non-root node discovers that its identity is equal to the number held by its parent, it becomes privileged until it stores this number in its own state and raises its flag.

Unlike the number, the value of the flag is not spread via the edges of the tree. Instead, each node looks at the flags of *all* of its in-neighbours. If there is a neighbour that has the same number, it takes the logical *or* of this flag and its own value. By this mechanism, a flag value of T is diffused through the graph, and eventually reaches the root. When the root finds that one of its in-neighbours has set its flag (and has the same number), it chooses a new number and sets its flag to F.

Thus, it is not necessary for all nodes to take steps before the root can choose a new number. All that is needed is that the number be copied from the root down the path in the tree to the privileged node. When that node has finished the critical section, the flag value must be copied along *some* path from that node back to the root. A node that is not on one of those two paths does not need to update its state.

The root chooses its new number in a round-robin fashion. In contrast with the previous protocol, the numbers do not range from 0 to $n-1$ but from 0 to n^2-1. The necessity for this will become clear in the proof. A node i is privileged only if the number of its parent equals $i (\mathrm{mod} n)$.

The implementation We first introduce some notation. Let node r be the root of the spanning tree. The state of a node u consists of a number $num_u \in \{0,\ldots,n^2-1\}$, and a boolean $flag_u \in \{T,F\}$. For a node v, $\mathsf{P}(v)$ denotes its parent in the tree, and $\mathsf{Anc}(v)$ the set of ancestors of v. More formally, $\mathsf{Anc}(v) \triangleq \{\mathsf{P}(v)\} \cup \mathsf{Anc}(\mathsf{P}(v))$ where $\mathsf{Anc}(r) \triangleq \emptyset$. The function Dest indicates for a certain node the destined privileged node: $\mathsf{Dest}(v) = num_v \bmod n$. The protocol is presented in Prot. 4.4.

Proof of correctness To prove the correctness of the protocol, we first define the set of legitimate configurations. We then show that the legitimate configurations satisfy the properties of mutual exclusion, and that progress and fairness are guaranteed. Next we show that the set of legitimate configurations is closed under the steps taken by the nodes, and finally we prove self-stabilization.

Definition 4.11 *A configuration C is* legitimate *iff it satisfies*

$$\begin{aligned}
\mathsf{L1} &\triangleq (\forall v \in V, u \in \mathsf{Anc}(v) :: num_v \preceq num_u) \ , \\
\mathsf{L2} &\triangleq (\forall v \in V :: (\forall x :: num_v \prec x \prec num_r \Rightarrow x \bmod n \neq v)) \ , \\
\mathsf{L3} &\triangleq num_{\mathsf{Dest}(r)} \neq num_r \Rightarrow \\
&\quad\ (\forall v \in V :: num_v = num_r \Rightarrow \neg flag_v) \ , \ \textit{and}
\end{aligned}$$

Node r

if $\mathsf{Dest}(r) = r \ \vee \ (\exists v \in \mathsf{In}(r) :: num_v = num_r \ \wedge \ flag_v)$

then $num_r := num_r + 1 \bmod n^2$

$flag_r := F$

Privileged if $\mathsf{Dest}(r) = r \ \vee \ (\exists v \in \mathsf{In}(r) :: num_v = num_r \ \wedge \ flag_v)$

Node $v \neq r$

if $\mathsf{Dest}(\mathsf{P}(v)) = v$

then $flag_v := T$

else $flag_v := (\exists u \in \mathsf{In}(v) \cup \{v\} :: num_u = num_{\mathsf{P}(v)} \ \wedge \ flag_u)$

$num_v := num_{\mathsf{P}(v)}$

Privileged if $\mathsf{Dest}(\mathsf{P}(v)) = v \ \wedge \ num_{\mathsf{P}(v)} \neq num_v$

Protocol 4.4 *The second mutual exclusion protocol.*

$$\mathsf{L4} \ \triangleq \ num_{\mathsf{Dest}(r)} = num_r \ \wedge \ \mathsf{Dest}(r) \neq r \ \Rightarrow \ flag_{\mathsf{Dest}(r)} \ ,$$

where the ordering $\preceq$ *on numbers is defined as*

$$x \preceq y \triangleq (y - x \bmod n^2) \leq n \ ,$$

and $x \prec y$ *denotes* $(x \preceq y) \ \wedge \ (x \neq y)$.

Definition 4.11 can be intuitively understood as follows: since the numbers are distributed downward through the tree, the ancestors of a node hold "more recent" numbers. The root increases the number, so nodes on higher levels should hold "higher" numbers. Since the root needs the cooperation of the destined privileged node in order to increase the number, the number of a node can never "lag behind" more than n (L1). Also, it is prohibited that the root increases the number before the privileged node has seen this number. Suppose node v has a number that is different from the one the root holds. The ancestors of v hold numbers in the range from num_v to num_r. The numbers that lie strictly "in between" must not destine the privilege to v: if there is an ancestor u of v for which $\mathsf{Dest}(u) = v$ yet $num_u \neq num_v$, then it must be that $num_r = num_u$, otherwise u might overwrite this number (by copying the number from its parent) before v sees it. L2 excludes these configurations. Lastly, the flags must express an "acknowledge" by node $\mathsf{Dest}(r)$.

If this node has set its number to num_r (i.e., completed the critical section), then its flag must be T (L4). If it has not yet done so, then the flags of all nodes that do hold num_r should be F (L3).

Lemma 4.12 *If, in a legitimate configuration L, the root r is privileged, then*

1. $num_{\mathsf{Dest}(r)} = num_r$, *and*
2. *if* $\mathsf{Dest}(r) \neq r$ *then* $flag_{\mathsf{Dest}(r)} = T$.

Proof If r is privileged, then by definition of the protocol

$$(\exists v \in \mathsf{In}(r) :: num_v = num_r \wedge flag_v) \vee \mathsf{Dest}(r) = r \ .$$

Combined with L3, the first part of the lemma follows. Applying L4 then yields the second part. ◁

Lemma 4.13 *If, in a legitimate configuration L, a node $v \neq r$ is privileged, then*

1. $num_{\mathsf{P}(v)} = num_r$, *and*
2. $v = \mathsf{Dest}(r)$.

Proof If v is privileged, then by definition of the protocol

$$\mathsf{Dest}(\mathsf{P}(v)) = v \ \wedge \ num_{\mathsf{P}(v)} \neq num_v \ .$$

Since by L1 $num_v \preceq num_{\mathsf{P}(v)} \preceq num_r$, L2 implies $num_{\mathsf{P}(v)} = num_r$. The second part of the lemma follows directly from this. ◁

With Lemmas 4.12 and 4.13 it is also easy to prove that in a legitimate configuration a node can not loose its privilege while it is executing the critical section.

Theorem 4.14 (safety) *In any legitimate configuration, at most one node is privileged.*

Proof If some non-root node v is privileged, then $num_v \neq num_{\mathsf{P}(v)} = num_r$ and $v = \mathsf{Dest}(r)$ by Lemma 4.13. Hence the root node is not privileged (as that would contradict the first part of Lemma 4.12). Furthermore, if any other node u is privileged, then $u = v$, by the second part of Lemma 4.13. ◁

Theorem 4.15 (progress) *In an arbitrary execution, let the system be in configuration C. Eventually, the system will reach a configuration C' in which the root is privileged.*

Proof Suppose, to the contrary, that r never becomes privileged and therefore never takes a step again. Consider the set $A = \{v \mid num_v = num_r\}$ and the set $A' = \{v \mid v \in A \wedge flag_v = T\}$. At least $r \in A$. Unless $A = V$, there is always a node $v \neq A$ whose parent $\mathsf{P}(v)$ is in A. Then v's next step will insert $v \in A$. Hence, eventually $A = V$.

Then $d = \mathsf{Dest}(r) \in A$, and d has become privileged and has set its flag to T. So $d \in A'$. Unless $A' = V \setminus \{r\}$, there is always a node $v \neq A'$ one of whose in-neighbours $w \in \mathsf{In}(v)$ is in A'. Then v's next step will insert $v \in A'$. Hence eventually $A' = V \setminus \{r\}$. At this point, r becomes privileged. ◁

Theorem 4.16 (fairness) *In an arbitrary execution, let the system be in a legitimate configuration L. For any node v, the system will eventually reach a configuration L' in which v is privileged.*

Proof By repeatedly applying theorem 4.15, we know that the system will reach a legitimate configuration L'', in which $num_r = d$ with $d \bmod n = v$. Just prior to that, no node can hold $num_w = d$ because that would violate L2. Hence $num_v \neq d$, and because nodes only copy numbers and flags from nodes with the same number, the first node w to hold d with $flag_w = T$ must be $d \bmod n$. Then $d \bmod n = v$ is privileged before r takes another step. ◁

Theorem 4.17 (closure) *If the system goes from a configuration C to configuration C', and C is legitimate, then C' is legitimate as well.*

Proof The transition from C to C' must be one of the following:

1. The root takes a step (increases its number), or
2. The privileged node $\mathsf{Dest}(r) \neq r$ takes a step (copies its number from its parent and sets its flag to T), or
3. A non-root node $v \neq \mathsf{Dest}(r)$ takes a step (copies the number from its parent, and copies the flag from an incoming node that has the same number).

We show that in all cases the requirements for a legitimate configuration remain satisfied in C'.

Case 1: Suppose that L1 is violated in C' for a certain node w (i.e., we have $(num_r - num_w \bmod n^2) > n$). Since L1 holds in C and num_r is increased by 1 in the step to C', in C, $num_r = num_w + n \bmod n^2$. But since L2 also holds in C, all numbers x strictly between num_w and num_r have $x \bmod n \neq w$. Since there are $n - 1$ of those numbers it must be that $\mathsf{Dest}(r) = \mathsf{Dest}(w) = w$. Applying Lemma 4.12 results in $num_w = num_r$, which is a contradiction.

Suppose L2 is violated in C' for a node w. Since L2 holds in C and r increases its *num* by 1, L2 can only get violated in C' if $x = num_r$, i.e., if $num_r \bmod n = w$ and $num_w \neq num_r$. But then r cannot take a step according to Lemma 4.12.

L3 and L4 must hold in C', because there is no $v \neq r$ for which $num'_v = num'_r$ (since by L1 r holds the "largest" number in C).

Case 2: In C', $num'_{\mathsf{Dest}(r)} = num'_r$ (lemma 4.13), so $\mathsf{Dest}(r)$ satisfies L2. It is easily verified that L1, L3 and L4 are also satisfied in C'.

Case 3: As in the previous case, L1 and L4 trivially hold in C'. L3 must also be satisfied, otherwise it is already violated in C, by the node that v copied the state from. Finally, L2 is satisfied because $num_v \leq num_{\mathsf{P}(v)} = num'_v$ by L1 in C. Then in C', the range $num'_v < x < num'_r$ for x in L2 becomes smaller. ◁

We must now prove self-stabilization. According to theorem 4.15, the root will keep on increasing its number. We can view an execution as consisting of *rounds*, separated by a step in which the root chooses a new number. In a round, the non-root nodes may take steps, but the state of the root does not change.

The proof that the system stabilizes from any initial configuration is based on the presence of what may be called *unoccupied* numbers. Define $\mathsf{Occ} = \{x \mid (\exists v \in V, v \neq r :: num_v = x)\}$, the set of *occupied* numbers, i.e., numbers that are present in some non-root node. Suppose that the system starts a new round, and that the new value of num_r is not occupied. The root can not increase its *num* again until node $\mathsf{Dest}(r)$ has been privileged. The number held by the root is distributed through the tree until it reaches this node. After the critical section has been completed, the flag is set and the 'acknowledge' is diffused through the graph. At the end of this round, node $\mathsf{Dest}(r)$ has a *num* equal to num_r.

The proof is split in two parts. First we prove that from any configuration, the system will reach a configuration from which in the next n rounds, the root will choose an unoccupied value. We then show that the configuration reached after these n rounds is legitimate.

Theorem 4.18 (self-stabilization) *In an arbitrary execution, let the system be in configuration C.*

1. *Eventually, the system will reach a configuration C' in which*

$$\neg (\exists v \in V :: num_r \prec num_v) \ .$$

2. *From C', the system will eventually reach a legitimate configuration.*

Proof We prove both parts separately.

Case 1: First, observe that in any configuration, there exist at least n consecutive unoccupied numbers (since the set of numbers has size n^2, and the non-root nodes hold at most $n-1$ different numbers). In C, let $u_1, u_2, \ldots, u_n$ be such a row of consecutive numbers. Second, consider the way in which Occ changes during an execution. The only steps in which new numbers are added to Occ is when the root chooses a new number. Thus, in any execution from C, the numbers remain unoccupied until the round in which num_r is equal to u_1. Call the configuration just prior to the start of this round C'. In this configuration $num_r = u_1 - 1 \bmod n^2$, so the only numbers x for which $num_r \prec x$ are precisely $u_1, u_2, \ldots, u_n$.

Case 2: In configuration C', the root changes its number to u_1. Since u_1 is unoccupied, eventually node $v = u_1 \bmod n$ becomes privileged. At the end of this round, num_r is set to u_2, which is still an unoccupied number. We can repeat this argument for $u_2, u_3, \ldots, u_n$. Call the configuration that is reached after these n rounds C''. Between C' and C'', each non-root is privileged exactly once, and exactly one node is privileged in each round.

We can now show that C'' is a legitimate configuration:

▷ Let v be an arbitrary non-root node, with parent p. Immediately after v is privileged, $num_v = num_p$. In all steps thereafter, if num_v or num_p changes, then either num_v is set equal to num_p, or num_p is

set to the number of p's parent, which is a higher number (which can be proven by induction on p's depth in the tree). So $num_{p'} \leq num_p$ in C''. Also, in C'' all numbers held by a node are within the range $u_1, u_2, \ldots, u_n$. Together this yields L1.

- ▷ Immediately after some node v is privileged, $num_v = num_r$ and $\mathsf{Dest}(r) = v$. From then on until C'', num_r is incremented at most n times. Thus, L2 must hold in C''.
- ▷ In C'', there is no non-root node that has the same number as the root, hence L3 and L4 trivially hold. ◁

4.5.3 *About spanning trees and combining protocols*

Several self-stabilizing spanning-tree protocols have been published, for instance by Afek *et al.* [AKY90] for undirected graphs, and Dolev *et al.* [DIM93] for arbitrary communication graphs. Since this last result matches our needs, we briefly sketch the (adapted) spanning tree protocol of Dolev *et al.* [DIM93].

The self-stabilizing spanning tree protocol is presented in Prot. 4.5. It will build a breadth-first-search tree using the distance to the root of the spanning tree. The range of values that can be assumed by field $dist_u$ is $\{0, \ldots, D\}$, where D is an upper bound on the diameter of the graph. To break the symmetry, one of the nodes is selected root a-priori and forces its distance to 0. The other nodes read the distance of their neighbours, and add 1 to the minimal value found. A selection among the nodes with minimal distance must be made to determine the parent in the spanning tree. This selection must be deterministic to ensure that once a complete spanning tree is computed, it remains stable provided no further errors occur.

A subtle point that should not go unmentioned is related to storing the value for $\mathsf{P}(u)$. Technically speaking the identity of any of the incoming nodes cannot be determined, because the identity of a node is not part of the state readable by other processors. But in fact, we are not even interested in the *identity* of such a node: we only need to know which of the incoming nodes is the desired node. So in fact we only have to store a 'port-address', or something similar, in $\mathsf{P}(u)$. If Δ equals the maximal in-degree of any node in the graph, then storing $\mathsf{P}(u)$ requires $O(\Delta)$ values. The proof of the protocol is straightforward, and for more details we refer to [DIM93].

Like [DIM93] — where the notion of fair protocol combination is formally derived — we compose two protocols to obtain a SSME protocol for arbitrary directed graphs. By observing that the mutual exclusion part does not alter the part of the state used by the spanning tree protocol, it is easily

Node r

$dist_r := 0$

$\mathsf{P}(r) := nil$

Node $u \neq r$

$dist_u := 1 + \min\{dist_v \mid v \in \mathsf{In}(u)\}$

$\mathsf{P}(u) :=$ deterministically select $v \in \mathsf{In}(u)$
with $dist_v = \min\{dist_v \mid v \in \mathsf{In}(u)\}$

Protocol 4.5 *The Spanning Tree Protocol of Dolev et al. [DIM93].*

seen that any proof of this spanning tree protocol still holds for the combined case. After stabilization of the spanning-tree part of the combined protocol, this part will not influence the mutual exclusion part any more: the values stored in the $\mathsf{P}(u)$ will not change. Then after stabilization of the spanning-tree part, the proof of the mutual exclusion part can be used unaltered to prove stabilization and correctness of the whole protocol.

The overall space-complexity of the protocol is determined by multiplying the number of values stored in num_u, $trust_u$, $dist_u$, and $\mathsf{P}(u)$. This gives $nD^2\Delta(G)$ states per node in total for our first SSME protocol. A factor $\Delta(G)$ can be saved by observing that $\mathsf{P}(u)$ is really a function of the state of u and is only used and maintained locally. This means that $\mathsf{P}(u)$ does not have to be stored explicitly.

4.6 Conclusions and further research

In this chapter we investigated the complexity, in number of states per processor, of achieving self-stabilizing mutual exclusion on strongly connected directed graphs. Starting of with Tchuente's approach, reducing the number of states per processor needed to $O(n^3)$, we presented two protocols using $O(nD^2)$ and $O(n^2D)$ states per processor. It would be interesting to see whether the second protocol can be modified to work with $O(n)$ numbers, reducing its complexity to $O(nD)$. In any case, we have shown an exponential improvement in space-complexity over the best known previous protocols for mutual exclusion on directed graphs.

I can't relax
'Cause I haven't done a thing
And I can't do a thing
'Cause I can't relax.

THE COMSAT ANGELS
Independence Day

5 Self-Stabilizing Ring-Orientation Using Constant Space

Abstract The ring-orientation problem requires all processors on an anonymous ring to reach agreement on a direction along the ring. A self-stabilizing ring-orientation protocol eventually ensures that all processors on the ring agree on a direction, regardless of the initial states of the processors on which the protocol is started. In this chapter we present two uniform deterministic self-stabilizing ring-orientation protocols for rings with an odd number of processors using only a constant number of states per processor. The first protocol operates in the link-register model under the distributed daemon, and the second protocol operates in the state-reading model under the central daemon. Both protocols do not assume an upper-bound on the length of the ring, and are therefore applicable to dynamic rings. As an application of our techniques we are able to prove that under the central daemon on an odd-length ring, the link-register model and the state-reading model are equivalent in the sense that any self-stabilizing protocol for the one model can be transformed to an equivalent, self-stabilizing, protocol in the other model.

5.1 Introduction

On oriented rings, processors agree on a direction along the ring. Distributed algorithms on rings are more easily derived if it is known that the ring is oriented (cf. [ASW88]), and may be more efficient than similar algorithms for un-oriented rings (cf. [San84]). To orient a ring the processors must choose a left and right neighbour consistently around the ring, such that the left

neighbour of each processor considers that processor to be its right neighbour. Processors can distinguish between a first and second neighbour, but to exclude trivial solutions the processors are required to be otherwise identical.

A comprehensive study on uniform rings in the asynchronous message passing model has been published by Attiya *et al.* [ASW88]. They show that there is no deterministic protocol to orient even-length rings, and that there also cannot exist a protocol to orient rings of arbitrary length, if the protocols are required to terminate. Syrotiuk and Pachl [SP87] present a simple asynchronous ring-orientation protocol using message passing, that is only guaranteed to work for rings whose length is odd and bounded. These papers do not address self-stabilization.

Self-stabilizing protocols are protocols that will eventually satisfy their specification, regardless of the initial state they were started in. Self-stabilization was introduced by Dijkstra [Dij74, Dij82], and is a framework in which one can derive fault-tolerant protocols capable of recovering from transient errors. This type of error can change the state of certain processors, but leaves the processors themselves in working order. Now consider a self-stabilizing protocol running on a set of processors, and consider the state just after the last error. This could just as well have been the initial state of the protocol, so the protocol must attempt to recover from this error. As the protocol is self-stabilizing it will be able to do so, provided the next error does not occur too soon. Therefore if transient errors are infrequent enough, self-stabilizing protocols keep the system in a correct state most of the time.

Our interest in self-stabilizing ring-orientation protocols is threefold. First of all, Burns and Pachl [BP89a] have shown that deterministic self-stabilizing protocols can break symmetry on oriented rings of prime size. Recently, Itkis *et al.* [ILS95] constructed a constant space protocol for the same problem. Our results imply that the ring does *not* have to be oriented to achieve these results. Second, if a ring can be oriented deterministically, we are interested in the necessary cost of doing so. Finally, our self-stabilizing ring-orientation protocols allow us to show that two models of inter-processor communication frequently used in the literature on self-stabilization are in fact equivalent on odd-length rings.

Several models for inter-process communication and processor scheduling in self-stabilizing systems have been proposed in the literature. In the *state-reading* model [Dij74] processors communicate by reading each others state. Secondly, in the *link-register* model [DIM93] a processor uses separate shared registers to communicate with each of its neighbours. Thirdly, in the *message-passing* model processors communicate by sending messages

(a)synchronously to other processors, where each message may incur an unbounded but finite delay.

The scheduling of processor steps is also an important issue. The *central daemon* [Dij74] schedules one processor at a time. This processor then performs one step in which it reads the information it needs, does some local processing, and finally writes its new state before returning control to the daemon. The *distributed daemon* [Bur87] may schedule several processors concurrently, but it is assumed that all scheduled processors first read the information they need, before any of them is allowed to write. Finally, the *read/write daemon* [DIM93] allows arbitrary interleaving of processor steps, much like the interleaving semantics considered for wait-free shared memory constructions.

Self-stabilizing ring-orientation protocols were studied by Israeli and Jalfon [IJ93]. They prove that no uniform deterministic self-stabilizing ring-orientation protocols exist in (a) the link-register model under the distributed daemon for even-length rings, (b) the state-reading model for either (b1) even-length rings under the central daemon, or (b2) arbitrary rings under the distributed daemon. This leads them to construct a randomized self-stabilizing ring-orientation protocol in the link-register model under the distributed daemon for arbitrary rings. To complement their impossibility results, they also present a uniform deterministic self-stabilizing ring-orientation protocol to orient odd-length rings in the link-register model under the distributed daemon. This protocol assumes knowledge of an upper bound on the length of the ring; it uses a non-constant number of states per processor.

We present two uniform deterministic self-stabilizing ring-orientation protocols for odd length rings, both using only a constant number of states per processor. The first protocol operates in the link-register model under the distributed daemon. This protocol is an adaption of the general randomized Israeli-Jalfon protocol. Contrary to the deterministic Israeli-Jalfon protocol for odd-length rings, our protocol does not depend on the length of the ring. This implies that our protocol can be used on dynamic rings on which the number of processors may change over time (provided that the length of the ring is odd in between these changes). The second protocol operates in the state-reading model under the central daemon, and complements the impossibility results of Israeli and Jalfon. Note that these results do not contradict the impossibility results of Attiya *et al.* [ASW88], as self-stabilizing protocols can never be required to terminate. Nor do these results contradict the impossibility of uniform self-stabilizing mutual-exclusion on rings of non-prime length proven by Dijkstra [Dij82], as a cyclic symmetrical configuration is not

necessarily un-oriented.

The number of models encountered in the literature on distributed algorithms is overwhelmingly large. Some of these models are clearly distinct, other models may only differ significantly for certain classes of problems. It is a major challenge to explore these different models, and to find conditions or problem areas such that using either model yields the same result. These models are then called equivalent. We prove that under the central daemon for a class of graphs, including oriented rings, the state-reading and link-register model are equivalent in the sense that any self-stabilizing protocol for the one model can be transformed to an equivalent, self-stabilizing, protocol in the other model. Using our second protocol this proves that the link-register model and the state-reading model are equivalent, in the same sense, under the central daemon on odd-length rings. Our results extend those of Gouda *et al.* [GHR90] to system models often used in the literature on self-stabilization.

The structure of this chapter is as follows. Section 5.2 describes the model of a distributed system assumed throughout this chapter. Several formal definitions of self-stabilization and their ramifications are discussed in Sect. 5.3. Then in Sect. 5.4 we start with a brief description of the Israeli-Jalfon protocol and continue presenting our uniform deterministic self-stabilizing ring-orientation protocol in the link-register model under the distributed daemon for odd-length rings. Sect. 5.5 describes the uniform deterministic self-stabilizing ring-orientation protocol in the state-reading model under the central daemon for odd-length rings. We conclude this chapter showing the equivalence of the link-register and the state-reading model on odd-length rings under the central daemon in Sect. 5.6, and presenting some interesting directions of further research in Sect. 5.7.

5.2 The model

We consider an *anonymous ring* $R = (V_R, E_R)$, consisting of an odd number n of clockwise numbered *nodes* $q \in V_R = \{0, \ldots, n-1\}$ and *clockwise edges* $e \in E_R = \{pq \mid p, q \in V_R \wedge q = (p+1) \bmod n\}$. The nodes of the ring are numbered purely for notational convenience: as the ring is anonymous no node has access to its number. Two nodes p, q are *neighbours* if either pq or qp is an element of E_R. Neighbouring nodes can communicate with each other directly. Each node can distinguish a first and second neighbour. For neighbours p of q we define $port_q(p)$ to equal 1 if p is the first neighbour of

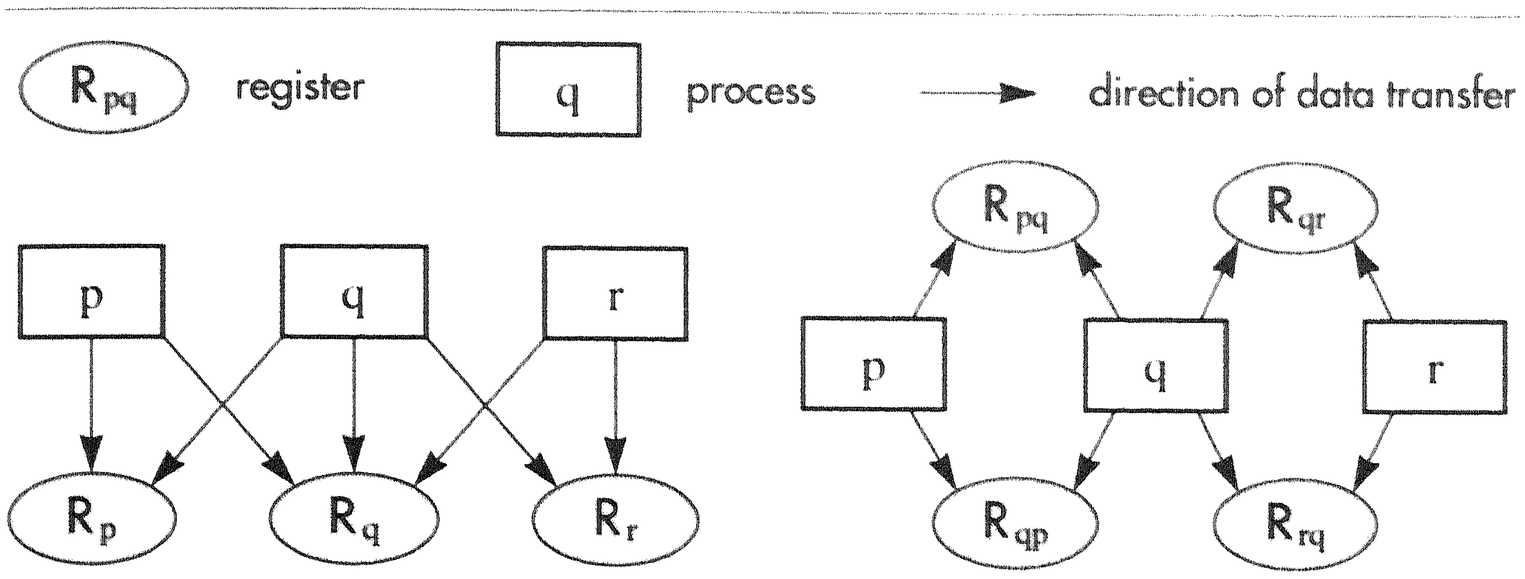

Figure 5.1 *Registers used by q to communicate with its neighbours.*

q, and 2 if p is the second neighbour of q[1]. In the remainder of this section, let nodes p and r be neighbours of a node q, with $port_q(p) = 1$.

In this chapter two models of communication are considered (see Fig. 5.1). In the *link-register* model, q will communicate with p and r using separate registers: R_{qp} is written by q and read by p, whereas R_{qr} is written by q and read by r In the *state-reading* model q stores its state in a register R_q readable by both p and r. The state-reading model is weaker than the link-register model, as in the state-reading model a processor q cannot introduce asymmetry in the states observed by p and r, whereas it can do so in the link-register model by writing different values to R_{qp} and R_{qr}.

The *state* s_q of a node q is comprised of its internal state and the contents of the registers it writes. The *configuration* C of the system is the Cartesian product $\prod_{q \in V_R} s_q$ over the states of all nodes in the ring. We write $C[q]$ for the state of node q in configuration C, and similarly $C[\mathsf{R}_q]$ for the value of register R_q in configuration C. Node q can update its state according to its *program* δ_q. Each *step* of node q in configuration C changes q's state to $\delta_q(C)$. In the link-register model, $\delta_q(C)$ is defined as $\delta_q(C[\mathsf{R}_{pq}], C[q], C[\mathsf{R}_{rq}])$. In the state-reading model, $\delta_q(C)$ is defined as $\delta_q(C[\mathsf{R}_p], C[q], C[\mathsf{R}_r])$. A *protocol* consists of a program δ_q for every node $q \in V_R$. A protocol is *uniform* if for all $p, q \in V_R$ we have $\delta_p = \delta_q$.

[1] Of course node q is not necessarily the first neighbour of node p if $pq \in E_R$.

A *schedule* is an infinite sequence $(P_i)_{i\geq 0}$ of *activations* $P_i \subseteq V_R$. A schedule is *fair* if each node $q \in V_R$ occurs in infinitely many activations P_i. Define the sequence $(t_i)_{i\geq 0}$ for a given schedule $(P_i)_{i\geq 0}$ setting $t_0 = 0$ and, for all $j > 0$, setting t_j to the minimal t such that $\bigcup_{i=t_{j-1}}^{t-1} P_i = V_R$. This sequence is unique, and partitions the schedule into *rounds* i, starting at t_i and ending at t_{i+1}, such that in each round each node is activated at least once. Thus a fair schedule is partitioned into infinitely many rounds. Under the *central daemon* one node is activated at a time to execute exactly one step: hence the central daemon only *allows* P_i that consist of exactly one node. Under the *distributed daemon* a set of nodes is simultaneously activated to concurrently execute exactly one step each. It allows arbitrary P_i. All nodes executing a step must have read the values in neighbouring registers before any node can be allowed to write the new value. Under both daemons activating P_i in configuration C yields a configuration C', denoted $C \rightarrow_{P_i} C'$, such that for all $q \notin P_i$ we have $C'[q] = C[q]$, whereas for all $q \in P_i$ we have $C'[q] = \delta_q(C)$.

A schedule $(P_i)_{i\geq 0}$ and an *initial configuration* C_0 induce an *execution* $E = (C_i)_{i\geq 0}$ such that for all $i \geq 0$ we have $C_i \rightarrow_{P_i} C_{i+1}$. We write $E(i)$ for the i-th configuration in execution E, and if $i \leq j$ we write $E(i) \Rightarrow_E E(j)$. An execution is fair if it is induced by a fair schedule[2]. We write $\mathcal{E}$ for the set of all fair executions allowed by the daemon under consideration. Let the schedule be partitioned into rounds as above. Then in execution E, round i starts in configuration $E(t_i)$, for which we write $\hat{E}(i)$.

We use some additional definitions in this chapter. A property X is called *stable* in configuration C (of execution E) if X holds in all configurations C' with $C \Rightarrow_E C'$. A property X is called stable from configuration C_f up to configuration C_l (of execution E) if this property holds in all configurations C' with $C_f \Rightarrow_E C' \Rightarrow_E C_l$. A *clockwise chain* is a sequence of nodes $q_0 \ldots q_k$, not necessarily $k < n$, such that for all i with $0 \leq i < k$, $q_i q_{i+1} \in E_R$. An *anti-clockwise chain* is a sequence of nodes $q_0 \ldots q_k$ such that for all i with $0 \leq i < k$, $q_{i+1} q_i \in E_R$. A *chain* is either a clockwise or an anti-clockwise chain.

5.3 About self-stabilization

A self-stabilizing protocol is a protocol that, when started in an arbitrary initial configuration, will eventually behave according to its specification. If we

[2] Note that we do allow stuttering (i.e., transitions $C \rightarrow_P C$) to occur in executions. In fact we need stuttering to make all executions infinite.

want to give a formal definition of self-stabilization, we first have to formalize what we mean by the specification of a protocol. In the early papers on self-stabilization the specification was viewed as describing the set of configurations $\mathcal{L}$, called the *legitimate configurations*, the protocol should be in. A mutual exclusion protocol, for instance, should always be in a configuration in which at most one node is executing its critical section. In this setting a protocol is *self-stabilizing* to a specification $\mathcal{L}$ if for every execution E the protocol will eventually reach a legitimate configuration $E(i)$, and once the configuration is legitimate it will remain legitimate forever: $(\forall E \in \mathcal{E}, \exists i \geq 0 :: E(i) \in \mathcal{L})$ and $(\forall C \in \mathcal{L}, \forall P :: C \rightarrow_P C' \Rightarrow C' \in \mathcal{L})$.

One drawback of configuration-based specifications becomes apparent if we again consider mutual exclusion. Usually it is required that the privilege—the node allowed to execute its critical section—is passed fairly among all nodes competing. This fairness-property cannot be expressed in a configuration-based specification, and is therefore not captured in the above definition of a self-stabilizing protocol.

Another, more general, way to view the specification of a protocol is as describing the set of behaviours $\mathcal{B} \subseteq \mathcal{E}$ the protocol should abide: i.e., any execution of the protocol should belong to its specification. In this setting a protocol is *self-stabilizing* to specification $\mathcal{B}$ if for all executions E of the protocol there exists an $i \geq 0$ such that all executions of the protocol starting in configuration $E(i)$ belong to $\mathcal{B}$:

$$(\forall E \in \mathcal{E}, \exists i \geq 0, \forall F \in \mathcal{E} : F(0) = E(i) :: F \in \mathcal{B}) \ .$$

In other words, a self-stabilizing protocol eventually cannot violate its specification.

Using this as a starting point, Burns *et al.* [BGM93] call a protocol *pseudo-stabilizing* to specification $\mathcal{B}$ if for all executions $E = (C_j)_{j \geq 0}$ of the protocol there exists an $i \geq 0$ such that $(E(j))_{j \geq i} \in \mathcal{B}$, i.e., if

$$(\forall E \in \mathcal{E}, \exists i \geq 0 :: (E(j))_{j \geq i} \in \mathcal{B}) \ .$$

In other words, a pseudo-stabilizing protocol eventually will not violate its specification. Pseudo-stabilizing protocols are weaker than self-stabilizing protocols. The first *will not* violate its specification, whereas the other *cannot* violate its specification. As the second statement depends on the assumption that the system will not be disrupted by another transient error, the difference seems artificial in practice. As Burns *et al.* [BGM93] observe that it is much easier to make pseudo-stabilizing protocols than to make self-stabilizing protocols, one might favour the first. However, pseudo-stabilizing protocols are not

required to stabilize within an certain amount of time. In fact, if $\mathcal{B}$ is closed under taking suffixes, one easily sees that if one can prove an upper bound on the stabilization time of a pseudo-stabilizing protocol, then this protocol is also self-stabilizing[3] [Tel94]. Thus pseudo-stabilizing protocols that are not self-stabilizing actually do not guarantee that the system will be legitimate even once.

One can build self-stabilizing protocols from scratch, or one can try to combine previous results to obtain more generally applicable protocols. Dolev *et al.* [DIM93] introduce *fair protocol combination* as a useful tool to construct a self-stabilizing protocol from two, simpler, self-stabilizing protocols. Informally speaking, fair protocol combination combines a master-protocol PM—that stabilizes to a certain specification provided certain external conditions hold—with a slave-protocol PS that stabilizes to executions in which exactly these conditions hold. The resulting protocol is a self-stabilizing version of PM without requiring those external conditions. In the combined protocol the steps of both protocols are taken alternately; the states of both protocols are merged so that the master protocol can read the state of the slave protocol. For more details we refer to Dolev *et al.* [DIM93]. Their construction of a self-stabilizing mutual exclusion protocol for arbitrary graphs is a good example of the use of fair protocol combination. It combines a self-stabilizing mutual exclusion protocol that only operates on tree-shaped graphs (the master protocol) with a self-stabilizing spanning tree protocol for arbitrary graphs (the slave protocol). The slave protocol ensures that the master protocol is eventually run a tree-shaped graph, which in turn guarantees that the combined protocol will eventually satisfy the mutual exclusion requirements.

5.4 Ring-orientation in the link-register model

In the ring-orientation problem it is required that all nodes agree on an orientation. That is, all nodes should choose a left and right neighbour such that the left neighbour of each node considers that node to be its right neighbour. Thus a self-stabilizing ring-orientation protocol must stabilize to the set of executions $\mathcal{B}_{RO} = \{E = (C_0 C_1 \ldots)\}$ where

$$(\forall q \in V_R, \exists p \in V_R :: left(q) = p \wedge right(p) = q \text{ is stable in } C_0 \text{ of } E)\,.$$

[3] Observe that if i is bounded, $(\forall E \in \mathcal{E}, \exists i : 0 \leq i < k :: (E(j))_{j \geq i} \in \mathcal{B})$, so if $\mathcal{B}$ is closed under taking suffixes, $(\forall E \in \mathcal{E} :: (E(j))_{j \geq k} \in \mathcal{B})$.

To obtain a uniform deterministic self-stabilizing ring-orientation protocol for odd-length rings, we use the randomized ring-orientation protocol of Israeli and Jalfon [IJ93]. This protocol is composed of two layers, combined using fair protocol combination, both operating in the link-register model under the distributed daemon. The lower layer is (what we will call) a randomized self-stabilizing *neighbour-ordering* protocol that will stabilize to a state in which any two neighbours agree on the order between them. The second layer is a deterministic self-stabilizing ring-orientation protocol assuming that all neighbours are mutually ordered, using only a constant number of states per node.

Intuitively the second layer of the Israeli and Jalfon protocol operates as follows. Initially certain nodes may hold a token, while others may be about to create tokens. If the ring is not oriented initially, at least one token is present or about to be created. However, once a node has created a token, it will never create a new token[4]. Each token has a fixed direction in which it travels around the ring. Whenever a node passes a token, this token directs the node. If two tokens meet, one of them will be eliminated based on the neighbour-ordering between the two nodes. Due to the elimination of opposite tokens, eventually all tokens will travel in the same direction around the ring, and thus the ring will eventually be oriented. For details we refer to [IJ93].

As an alternative for the lower layer we present a uniform deterministic self-stabilizing neighbour-ordering protocol for odd-length rings, using only a constant number of states per node, in the link-register model under the distributed daemon. Combining this with the second layer of the Israeli and Jalfon protocol, yields the desired self-stabilizing ring-orientation protocol for odd-length rings using a constant number of states per node.

5.4.1 Neighbour-ordering in the link-register model

In the neighbour-ordering problem it is required that eventually any two neighbours p and q agree on an order $<$ between them, and that once $p < q$ holds, $p < q$ remains to hold forever. That is, a self-stabilizing neighbour-ordering protocol must stabilize to the set of executions $\mathcal{B}_{NO} = \{E = (C_0 C_1 \ldots)\}$ where

$$(\forall\, pq \in E_R :: (p < q \text{ stable in } C_0 \text{ of } E) \vee (p > q \text{ stable in } C_0 \text{ of } E)) \ .$$

[4] This may seem to conflict with self-stabilization, but is easily achieved if one makes sure that no configuration in which a token may be generated by a node q can be the result of a step of q.

In this section we develop a uniform deterministic self-stabilizing neighbour-ordering protocol operating in the link-register model under the distributed daemon[5]. It is based on certain properties of odd-length rings, that we derive in the next few paragraphs.

Define for neighbours $p, q \in V_R$ the relations $\ll$ and $\equiv$ by

$$p \ll q \quad \text{if } port_p(q) < port_q(p) \text{ , and}$$
$$p \equiv q \quad \text{if } port_p(q) = port_q(p) \text{ .}$$

Label the edges $pq \in E_R$ (recall that E_R only contains clockwise edges) with $O \in \{\ll, \equiv, \gg\}$ such that $p\ O\ q$ if and only if pq is labelled O. If we consider chains $q_0 \ldots q_{k+1}$, then

$$q_0 \gg q_1 \equiv \cdots \equiv q_k \ll q_{k+1} \text{ implies } k \text{ is even, and} \tag{5.1}$$
$$q_0 \gg q_1 \equiv \cdots \equiv q_k \gg q_{k+1} \text{ implies } k \text{ is odd.} \tag{5.2}$$

This leads us to the following claim.

Claim 5.1 *For any ring R with edges labelled as defined above, the number of edges labelled $\equiv$ is even.*

Proof Let R be an arbitrary ring, and consider neighbouring nodes p, q, r and s (in that order). If $q \ll r$ it is easily seen that $port_q(r) = port_r(s) = 1$ and $port_r(q) = port_q(p) = 2$. Thus we can remove all edges not labelled $\equiv$ (merging their endpoints) and the remaining ring has the same number of $\equiv$-labelled edges as R has. But obviously if all edges in the remaining ring are labelled $\equiv$, the ring must have an even number of edges. ◁

From this claim it follows that any odd-length ring exhibits a certain asymmetry. The construction of our protocol will take this asymmetry as point of departure.

Corollary 5.2 *Let R be an odd-length ring, whose edges are labelled as defined above. Then there exists at least one edge labelled with $\ll$ or $\gg$, and the parity*

[5] Under the *central daemon* neighbour-ordering is easily achieved deterministically. Let each register R_{pq} contain a value in the set $\{0, 1\}$, and define $p < q$ if $R_{pq} < R_{qp}$. For any edge pq run the following protocol on p (and q, with p and q interchanged):

if $R_{pq} = R_{qp}$ **then** invert R_{pq}.

This protocol will stabilize in exactly 1 round.

of the number of edges labelled $\gg$ *is unequal to the parity of the number of edges labelled* $\ll$.

Now the straightforward approach to construct a self-stabilizing protocol for ring-orientation of odd-length rings might be one in which one generates tokens on $\ll$ or $\gg$-labelled edges, letting them travel in the direction of the $\ll$ or $\gg$. This will not work, however, because the parity-difference between $\ll$ and $\gg$-labelled edges on which it is based may be destroyed by two causes: initially extra tokens may be present on $\equiv$ labelled links, and each $\ll$-labelled edge has to generate infinitely many tokens, because it can never know it generated one. Apparently our previous corollary alone will not do: we are in need of an additional property of odd-length rings.

Definition 5.3 *Mark an arbitrary, non zero, number of edges originally labelled* $\equiv$ *with* $\bowtie$ *instead. Then a clockwise chain* $p_{l+1}p_l \ldots p_0q_0 \ldots q_rq_{r+1}$ *is called a* $\bowtie$-delimited chain $K^{\bowtie}$ *if the edges* $p_{l+1}p_l$, p_0q_0 *and* q_rq_{r+1} *are the only edges marked* $\bowtie$ *in* $K^{\bowtie}$.

Note that this definition also captures the case in which only one edge is marked $\bowtie$, because chains are allowed to span the ring more than once. This definition is used in the following lemma to expose yet another source of asymmetry on odd-length rings.

Lemma 5.4 *Let* R *be an odd-length ring, whose edges are labelled as defined above. Mark an arbitrary number of edges with* $\bowtie$ *according to the previous definition. Then there exists a* $\bowtie$*-delimited chain* $p_{l+1}p_l \ldots p_0q_0 \ldots q_rq_{r+1}$ *such that the parity of the number of edges between* p_l *and* p_0 *labelled* $\gg$ *is unequal to the parity of the number of edges between* q_0 *and* q_r *labelled* $\ll$.

Proof Let λ be the total number of edges labelled $\gg$ and let ρ be the total number of edges labelled $\ll$. By Corollary 5.2 we know that λ is odd (even) whereas ρ is even (odd). Given a ring R marked with $\bowtie$, consider the set $\{K_i^{\bowtie}\}$ of all $\bowtie$-delimited chains along R. For each $K_i^{\bowtie}$ let l_i be the number of $\gg$ labelled edges left of the middle $\bowtie$, and let r_i be the number of $\ll$ labelled edges right of the middle $\bowtie$.

As all edges marked $\bowtie$ were labelled with $\equiv$, and as all other edges occur exactly once left and exactly once right of the middle $\bowtie$ of some $K_j^{\bowtie}$, we see that $\sum l_i = \lambda$ and $\sum r_i = \rho$. Assume λ odd and ρ even (the other case leads to the same result). Then the number of odd l_i must be odd, whereas

the number of odd r_i must be even. Therefore there must be a chain $K_i^{\bowtie}$ for which the parity of l_i does not equal the parity of r_i. ◁

Now we are ready to give an informal description of the neighbour-ordering protocol. Each register R_{pq} holds a field $dir \in \{0,1\}$ such that $\mathsf{R}_{pq}.dir < \mathsf{R}_{qp}.dir$ if and only if $p < q$. If pq are un-ordered, i.e., if neither $p < q$ nor $p > q$, then we write $p = q$. If $p \neq q$, neither p nor q will try to order the edge pq. If $p = q$, node q can try to order pq by inverting $\mathsf{R}_{qp}.dir$, but under the distributed daemon both p and q might try to order pq simultaneously. In that case the net result will be zero, and if the daemon always schedules p and q simultaneously pq will never become ordered.

To avoid the above livelock schedule we propose the following. First we make sure that for any p and q with $p \gg q$, only q tries to order pq. By Corollary 5.2 at least one such pq exists, which means that eventually one pair of nodes will be ordered. To order the remaining edges pq with $p \equiv q$, we only allow nodes q that are already ordered with respect to their other neighbour r (i.e., nodes q with $q \neq r$) to try to order pq.

Now there are two cases to consider

1. $o = p = q \neq r$: p will not try to order pq so q must be allowed to do so, or
2. $o \neq p = q \neq r$: the situation for p and q is symmetric and livelock might still occur.

To enable q to tell case 1 and 2 apart, R_{pq} will store whether $o = p$ in a field $ord \in \{\neq, =\}$. To break the symmetry in case 2 we apply Lemma 5.4: the parity of $\gg$-labelled edges left of p must be unequal to the parity of $\ll$-labelled edges right of q for at least one edge pq in case 2. We let each node maintain the parity of $\gg$-labelled edges coming in from the other side in a field $parity \in \{0,1\}$, and only allow q to try to order pq if the parity it holds equals 1.

The protocol Let p and r be the neighbours of node q. In the neighbour-ordering protocol each register R_{qp} has the following fields:

- ▷ $youare \in \{1,2\}$, to encode the ordering $\ll$ between p and q. This field is always set such that $\mathsf{R}_{qp}.youare = port_q(p)$.
- ▷ $dir \in \{0,1\}$, to encode the desired node-ordering $<$ between p and q.
- ▷ $ord \in \{\neq, =\}$, to tell p whether $q = r$ or not.

```
1   R_qp.youare := port_q(p)                    (* force ≫' to correspond to ≫ *)
2   if q = r then R_qp.parity := 0              (* anchor the parity-chain *)
3   if q ≪' r ∧ q ≠ r then R_qp.parity := (R_rq.parity + 1) mod 2
                        (* increase parity for incoming ≪' from other side *)
4   if q ⋘' r ∧ q ≠ r then R_qp.parity := R_rq.parity
                                            (* pass on parity to other side *)
5   if q = r then R_qp.ord := '=' else R_qp.ord := '≠'
                                            (* pass ordering to other side *)
6   if p = q                        (* only try to order not yet ordered pairs *)
7   then if p ≫' q then invert R_qp.dir
                        (* note that only the head of ≫' directs the arc *)
8   else if p ≡' q ∧ q ≠ r ∧ R_pq.ord = '='
                then invert R_qp.dir        (* case ···o ≈ p = q ≠ r··· *)
9   else if p ≡' q ∧ q ≠ r ∧ R_pq.ord = '≠' ∧ R_qp.parity = 1
                then invert R_qp.dir        (* case ···o ≉ p = q ≠ r··· *)
```

Protocol 5.1 *Neighbour-ordering subroutine for node q to adjust* R_{qp}.

▷ *parity* $\in \{0,1\}$, holding the parity of the number of $\ll$ labelled edges coming in from the other side (i.e., through r).

Thus the protocol uses $2^4 \times 2^4$ states per node. Define for neighbouring nodes p and q

$$\begin{array}{lcll} p & \ll' & q & \text{if } \mathsf{R}_{pq}.youare < \mathsf{R}_{qp}.youare\ , \\ p & \equiv' & q & \text{if } \mathsf{R}_{pq}.youare = \mathsf{R}_{qp}.youare\ , \\ p & < & q & \text{if } \mathsf{R}_{pq}.dir < \mathsf{R}_{qp}.dir\ , \text{ and} \\ p & = & q & \text{if } \mathsf{R}_{pq}.dir = \mathsf{R}_{qp}.dir\ . \end{array}$$

These predicates can be evaluated locally at both p and q.

In the neighbour-ordering protocol all nodes run the same program, consisting of two subroutines. The subroutine for a node q with neighbours p and r to update the register used to communicate with p (i.e., R_{qp}) is presented in Prot. 5.1. Node q runs a similar subroutine to update R_{qr} (obtained by swapping p and r in the subroutine code). The subroutine consists of a set of guarded commands, denoted by **if**-statements. Whenever node q is scheduled by the distributed daemon to take a step, all commands (both for

updating R_{qp} and R_{qr}) whose guards are true are executed. At the start of each step, q reads R_{pq} and R_{rq} once, and q will write the new contents for R_{qp} and R_{qr} once at the end of each step.

Proof of correctness Throughout the proof, let $(P_i)_{i\geq 0}$ be an arbitrary fair schedule under the distributed daemon, let C_0 be an arbitrary initial configuration, and let E be the execution they induce.

Claim 5.5 *If $p < q$ holds in a configuration C of E, then $p < q$ is stable in C.*

Proof The only steps of p or q that can change $\mathsf{R}_{pq}.dir$ or $\mathsf{R}_{qp}.dir$ (and therefore the ordering between p and q) are those on line 7, 8 and 9. But these three steps are guarded by the condition $p = q$ (see line 6), so the result follows. ◁

Thus it remains to be shown that there exists an i such that for all $pq \in E_R$ we have $p \neq q$ in configuration C_i of execution E. We prove this by exhibiting an upper bound on the number of rounds of E after which a configuration satisfying this condition is guaranteed to be reached. This also provides us with an upper-bound on the stabilization time, measured in rounds, of the protocol. Recall that $\hat{E}(i)$ is the configuration at the start of round i in execution E.

Lemma 5.6 *For all $pq \in E_R$, if $p \not\equiv q$ then $p \neq q$ in $\hat{E}(2)$.*

Proof First observe that after round 0 each node has taken step 1 at least once, so we have $p \equiv q \Leftrightarrow p \equiv' q$, and similarly $p \gg q \Leftrightarrow p \gg' q$ and $p \ll q \Leftrightarrow p \ll' q$ for all $pq \in E_R$. Clearly these properties are stable in $\hat{E}(1)$. Take an arbitrary edge $pq \in E_R$ with $p \not\equiv q$. Assume $p \gg q$ (the case $p \ll q$ is handled similarly). Then $p \gg' q$ during round 1, which means that p cannot apply steps 7, 8 or 9 on R_{pq} during round 1. If $p \neq q$ in $\hat{E}(1)$, then we are done according to Claim 5.5. Otherwise, q will apply step 7 on R_{qp} in round 1, setting $p \neq q$. ◁

This leaves edges pq with $p \equiv q$. We first show that every node q faithfully conveys its order-relation with neighbour r to the other neighbour p.

Claim 5.7 *Let p and r be the neighbours of q. If $q \neq r$ is stable from $\hat{E}(i)$ to $\hat{E}(j)$, then $\mathsf{R}_{qp}.ord =$ '$\neq$' is stable from $\hat{E}(i+1)$ to $\hat{E}(j)$. Similarly, if $q = r$ is stable from $\hat{E}(i)$ to $\hat{E}(j)$, then $\mathsf{R}_{qp}.ord =$ '$=$' is stable from $\hat{E}(i+1)$ to $\hat{E}(j)$.*

Proof If $q \neq r$ is stable from $\hat{E}(i)$ to $\hat{E}(j)$, then q will apply step 5 in rounds i through $j-1$, setting $\mathsf{R}_{qp}.ord = \text{`}\neq\text{'}$. Thus $\mathsf{R}_{qp}.ord = \text{`}\neq\text{'}$ is stable from $\hat{E}(i+1)$ to $\hat{E}(j)$. The case for $q = r$ is handled similarly. ◁

The next claim shows that if for a certain edge rp we have $r = p$ for l rounds, then all nodes q with distance $k < l$ from p (without another equal pair of nodes in between) correctly store the parity of $\gg$ labelled edges (i.e., those pointing away from p) between p and q.

Claim 5.8 *Suppose $r = p_0$ is stable from $\hat{E}(i)$ to $\hat{E}(i+l)$, where $i \geq 2$ and $l \geq 0$. Pick $k < l$ such that on the chain $rp_0p_1 \ldots p_k$ along the ring we have $p_{i-1} \neq p_i$ for all i with $1 \leq i \leq k-1$. Define $\sigma_k = |\{p_i \mid 1 \leq i \leq k-1 \wedge p_{i-1} \gg p_i\}|$, i.e., the number of $\gg$ labelled edges between p_0 and p_{k-1}. Then the property $\mathsf{R}_{p_{k-1}p_k}.parity = \sigma_k \bmod 2$ is stable from $\hat{E}(i+k)$ up to $\hat{E}(i+l)$.*

Proof Proof by induction on k. In the base case $k = 1$ we have $\sigma_1 = 0$. Since $r = p_0$ holds during round i through $i+l-1$, p_0 applies step 2 in these rounds, setting $\mathsf{R}_{p_0p_1}.parity = 0$ as required. Thus this property is stable up to $\hat{E}(i+l)$.

Assume the induction hypothesis holds for $k < l-1$ and assume $p_{k-1} \neq p_k$. Then $\mathsf{R}_{p_{k-1}p_k}.parity = \sigma_k \bmod 2$ from $\hat{E}(i+k)$ up to $\hat{E}(i+l)$. Then there are two cases to consider. If $p_{k-1} \gg p_k$, then $p_{k-1} \gg' p_k$ stable in $\hat{E}(i+k)$ (as $i \geq 2$) and $\sigma_{k+1} = 1 + \sigma_k$. Now p_k takes step 3 in rounds $i+k$ through $i+l-1$, setting $\mathsf{R}_{p_kp_{k+1}}.parity$ to $(\mathsf{R}_{p_{k-1}p_k}.parity + 1) \bmod 2$ in these rounds as required. If $p_{k-1} \not\gg p_k$, then $p_{k-1} \not\gg' p_k$ stable in $\hat{E}(i+k)$ (as $i \geq 2$) and $\sigma_{k+1} = \sigma_k$. So p_k takes step 4 in rounds $i+k$ through $i+l-1$, setting $\mathsf{R}_{p_kp_{k+1}}.parity$ to $\mathsf{R}_{p_{k-1}p_k}.parity$ in these rounds as required. ◁

Theorem 5.9 *The neighbour ordering protocol stabilizes on an odd-length ring, under the distributed daemon, to a configuration in which for any two neighbours p, q, $p \neq q$. Furthermore, once $p < q$, then $p < q$ holds forever. The system stabilizes in $O(n^2)$ rounds.*

Proof By Lemma 5.6 in configuration $\hat{E}(2)$ we have for all edges $pq \in E_R$ with $p \not\equiv q$ that $p \neq q$. By Corollary 5.2 there exists at least one such edge.

Suppose that in configuration $\hat{E}(i)$ there exist x triplets o, p, q with $o = p = q$. By the above observation for at least one we actually have the quartet $o = p = q \neq r$. Suppose $o = p$ is stable from $\hat{E}(i)$ to $\hat{E}(i+2)$. Then by Claim 5.7, $\mathsf{R}_{pq}.ord = \text{`}=\text{'}$ and $\mathsf{R}_{qp}.ord = \text{`}\neq\text{'}$ are stable from $\hat{E}(i+1)$ to $\hat{E}(i+2)$.

Then in round $i+1$, p cannot apply any of the steps 7, 8 and 9. If in $\hat{E}(i+1)$, $p = q$ holds, q will apply 8 in round $i+1$, setting $p \neq q$. If in $\hat{E}(i+1)$, $p = q$ does not hold, then by Claim 5.5 in $\hat{E}(i+2)$ we have $p \neq q$. If $o = p$ is not stable from $\hat{E}(i)$ to $\hat{E}(i+2)$, then $o \neq p$ in $\hat{E}(i+2)$. In either case, the number of triplets $o = p = q$ in $\hat{E}(i+2)$ will be at most $x-1$. As the number of triplets is at most $n-1$ (n is the number of nodes), in configuration $\hat{E}(2n)$ there will only be edges pq with $p = q$ such that $o \neq p = q \neq r$.

Suppose that in configuration $\hat{E}(m)$, $m \geq 2n$, there are y edges pq with $p = q$ (and hence $o \neq p = q \neq r$). Now mark all edges $pq \in E_R$ with $p = q$ (and hence $p \equiv q$) with $\bowtie$. By Lemma 5.4 there exists at least one $\bowtie$-delimited chain $p_{l+1} = p_l \cdots p_0 = q_0 \cdots q_r = q_{r+1}$ such that the parity of the number of edges between p_l and p_0 labelled $\gg$ is unequal to the parity of the number of edges between q_0 and q_r labelled $\ll$. Note that for all edges pq between p_l and p_0 or between q_0 and q_r we have $p \neq q$.

Let $\mu = 1 + \max(l, r)$. Consider the above chain and suppose that $p_{l+1} = p_l$ and $q_r = q_{r+1}$ are stable from $\hat{E}(m)$ to $\hat{E}(m+\mu+1)$. Then using Claim 5.8, $\mathsf{R}_{p_0q_0}.parity \neq \mathsf{R}_{q_0p_0}.parity$ in round $m+\mu$. By Claim 5.7 also $\mathsf{R}_{p_0q_0}.ord = \text{`}\neq\text{'}$ and $\mathsf{R}_{q_0p_0}.ord = \text{`}\neq\text{'}$ in round $m+\mu$. If $p_0 = q_0$ in $\hat{E}(m+\mu)$, either p_0 or q_0 will apply step 9 in round $m+\mu$ setting $p \neq q$ in $\hat{E}(m+\mu+1)$. If $p_0 \neq q_0$ in $\hat{E}(m+\mu)$, then by Claim 5.5 $p_0 \neq q_0$ in $\hat{E}(m+\mu+1)$. In either case, the number of edges pq with $p = q$ in $\hat{E}(m+\mu+1)$ is at most $y-1$. Also, if $p_{l+1} = p_l$ or $q_r = q_{r+1}$ is not stable from $\hat{E}(m)$ to $\hat{E}(m+\mu+1)$, then the number of edges pq with $p = q$ in $\hat{E}(m+\mu+1)$ is at most $y-1$.

As $\mu \leq n$, and $y \leq \lfloor \frac{n}{2} \rfloor$ in $\hat{E}(2n)$, after round $2n + (n+1)\lfloor \frac{n}{2} \rfloor$ the system will certainly be in a legitimate configuration. ◁

5.5 Ring-orientation in the state-reading model

In this section we present a uniform deterministic self-stabilizing ring-orientation protocol for odd-length rings operating in the state-reading model under the central daemon, using only a constant number of states per node. This is the best we can hope for, as Israeli and Jalfon [IJ93] have already shown that such a protocol is impossible under the distributed daemon, and under the central daemon if the length of the ring is even. Most ring-orientation protocols operate by forwarding a token with a fixed direction around the ring. But in the state-reading model both neighbours of a node read the same state. If that node were to hold a token with a direction, it is not immediately obvious how to forward that token to the one neighbour it points to without possibly forwarding it to the other neighbour as well. In view of this observation it is

perhaps surprising that it is possible at all to orient a ring in the state reading model.

Intuitively the protocol operates as follows. Let each node have a *colour* taken from the set $\{0, 1\}$. Try to give neighbouring nodes alternating colours by inverting the colour of a node if it has the same colour as both its neighbours[6]. Of course on an odd-length ring this is never completely possible, so we are bound to end up with patterns like 001 and 110 around the ring. Call such patterns tokens. The idea is to make these tokens travel around the ring, orienting the nodes they visit. This requires us to impose a direction on tokens, for which we let each node store a *phase* taken from the set $\{+, -\}$. A token is directed if the nodes with equal colour have opposite phase: then the direction of the token points from the node with phase + to the node with phase −. Undirected tokens can be directed by letting the middle node in a token invert its phase.

Let us write 0_- for a node with colour 0 and phase −, and consider the pattern $0_+0_-1_-0_-$ around the ring. If we allow the second node to change its state to 1_+ we get the pattern $0_-1_+1_-0_-$: the token has moved one step in its direction. Now let a node also maintain its orientation, taken from $\{\leftarrow, \rightarrow\}$—for instance by storing the port of the neighbour the head of the arrow points to. If a token moves one step, the node changing colour is oriented into the direction of the token. Our protocol depends on the fact that a token always keeps the same direction, until it is eliminated. Then in the situation $0_+0_-1_+0_-$ it is unwise to allow the second node to change its state to 1_+ as this would yield the situation $0_+1_+1_+0_-$. Here the token becomes undirected, and thus may decide to invert its direction: $0_+1_-1_+0_-$. Therefore, in situations like $0_+0_-1_+0_-$ we must wait until the third node sets its phase to −.

We have already seen that in almost all configurations (except for the special case in which all nodes have the same colour) at least one token exists. If we can make sure that eventually all tokens travel in the same direction around the ring, the ring will eventually become oriented. Now consider what happens when two tokens with opposite direction meet. This situation is depicted in Fig. 5.2 (where the steps are taken from the complete description of the protocol in Prot. 5.2). Other nodes in this example may take a step

[6] This trick cannot be applied immediately under the distributed daemon, because the distributed daemon is free to schedule all nodes simultaneously. If all nodes in the ring have the same colour, then under such a schedule all nodes will simultaneously invert their colour.

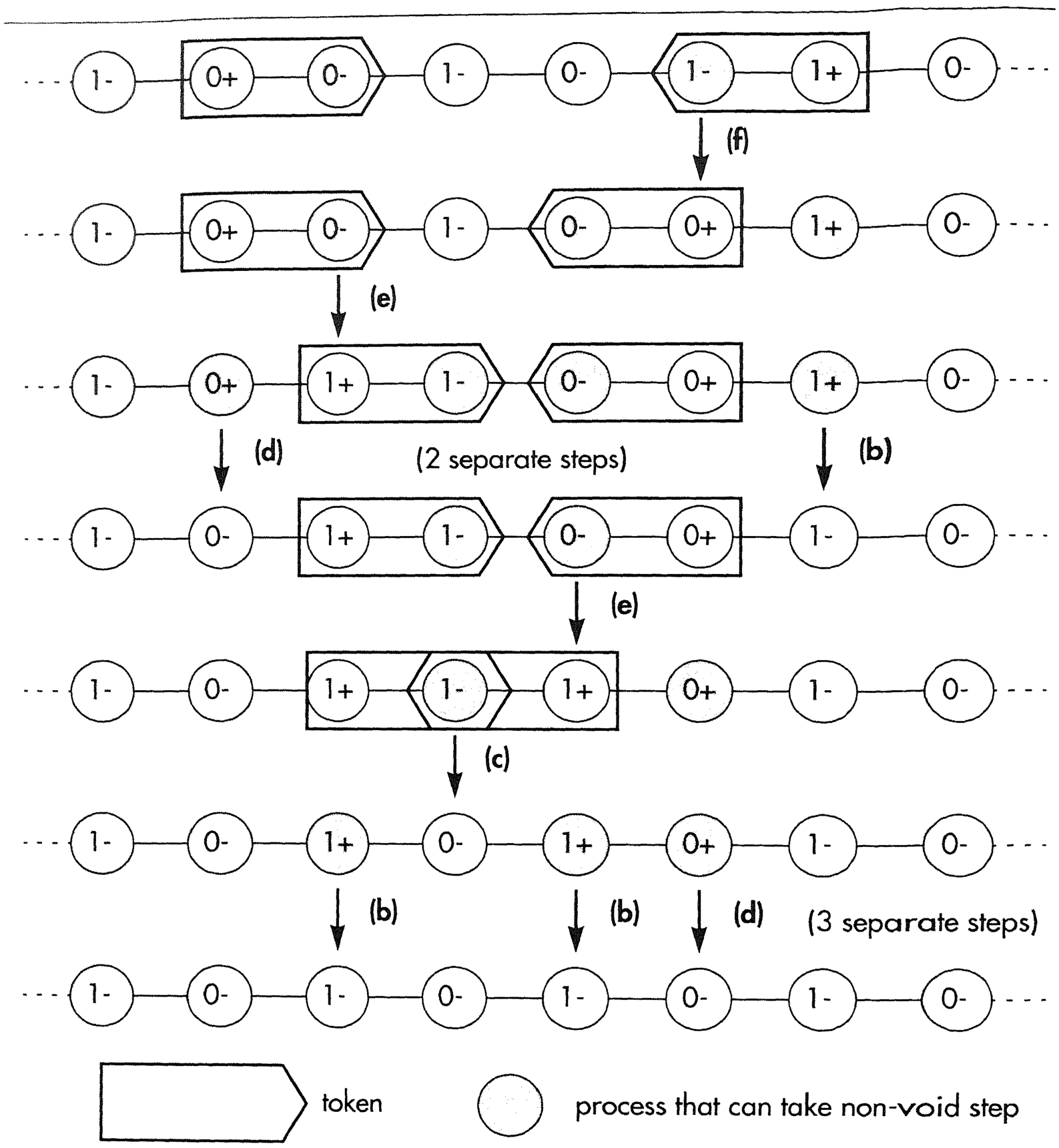

Figure 5.2 *Example of two tokens with opposite direction meeting.*

instead, but this leads to essentially the same situation. We see that both tokens are eliminated altogether.

δ	p	q	r	q'
(a)	0	0	0	1_-
(b)	0	1	0	1_-
(c)	1	1	1	0_-
(d)	1	0	1	0_-

δ	p	q	r	q'
(e)	0_+	0_-	1_-	$\overrightarrow{1_+}$
(f)	1_+	1_-	0_-	$\overrightarrow{0_+}$

δ	p	q	r	q'
(g)	0_-	0_-	1	0_+
(h)	0_+	0_+	1	0_-
(i)	1_-	1_-	0	1_+
(j)	1_+	1_+	0	1_-

Protocol 5.2 *Ring-orientation program for node q.*

5.5.1 The protocol

Let p and r be the neighbours of q. Each node q stores its whole state (i.e., its colour, phase and orientation) in register R_q. In the ring-orientation protocol, all nodes q run the same program, presented in Prot. 5.2 as a state-transition function δ. The tables list the applicable steps for a node q with neighbours p and r, depending on the state of p, q and r (i.e., the contents of R_p, R_q and R_r). The entries under q' list the new state of q after applying δ. To reduce the size, symmetric steps (with p and r interchanged) are not shown. So steps (e) through (j) actually represented two steps. If only some components of a state are specified, the other components may have an arbitrary value. If in the new state the orientation is not specified, it should equal the direction of q in the old state. In cases (e) and (f), the direction of q in the new state should be read as pointing from p to r. Node q can always determine this orientation because in these steps p and r hold opposite colours. The port of p and r is used to encode the direction.

Each table contains related steps. The first table lists the steps that try to colour the ring alternately, and to make sure that alternately coloured nodes have phase $-$. The second table lists the steps responsible for forwarding the token, and the third table lists the steps that break the symmetry in possible tokens by imposing one direction on them.

5.5.2 Proof of correctness

Throughout the proof, let $(P_i)_{i\geq 0}$ be an arbitrary fair schedule under the central daemon, let C_0 be an arbitrary initial configuration, and let E be the execution of the protocol they induce. If $n = 1$, the protocol is trivially correct, so in the remainder of the proof we assume $n \geq 3$. We prove correctness of the algorithm in stages. In each stage we define a set of configurations and

show that the protocol will converge to this set, when started in a configuration from the set of the previous stage. All sets are shown to be closed under transitions of the protocol. In the final stage the set of configurations will contain exactly those that are oriented. To describe the configurations in a set we use regular expressions.

Definition 5.10 *A configuration* matches *a regular expression L if the concatenation of the states of some chain (clockwise or anti-clockwise) of length n matches the regular expression. A regular expression L is* closed *under steps of the protocol if for all configurations C that match L, C' with $C \to_P C'$ for arbitrary allowed P matches L as well.*

In other words, a configuration matches a regular expression if we can cut the ring between a pair of nodes, and if the resulting chain, or string, of states matches the regular expression.

Define the regular expression L_1 by

$$\begin{aligned} L_1 &= (O_1 I_1)^* \ , \\ O_1 &= 0 \mid 0_+0_-0_+ \mid 0_-0_+ \mid 0_+0_- \ , \text{ and} \\ I_1 &= 1 \mid 1_+1_-1_+ \mid 1_-1_+ \mid 1_+1_- \ . \end{aligned}$$

Lemma 5.11 *Starting in an arbitrary configuration, we eventually reach a configuration matching L_1. Furthermore, L_1 is closed.*

Proof Consider 0-chains, i.e., chains of nodes coloured only with 0 that are delimited at both ends by 1-coloured nodes. If all nodes in the ring are coloured 0, only step (a) will be applicable by all nodes creating one 0-chain after the first node takes a step. For 0-chains define the width w as

$$\begin{aligned} w(0_+) &= 1.1 \ , \\ w(0_-) &= 1 \ , \text{ and} \\ w(0_1 \ldots 0_k) &= k + p(0_1 0_2) + p(0_k 0_{k-1}) \ . \end{aligned}$$

where the penalty p for the endpoints of the chain is given by

$$p(ab) = \begin{cases} 1.5 & \text{if } a = 0_- \ , \\ 1.6 & \text{if } ab = 0_+0_+ \ , \text{ and} \\ 0 & \text{otherwise} \ . \end{cases}$$

1-chains and their associated penalty and width are defined similarly. It is easily checked that all 0-chains (1-chains) K matching O_1 (I_1) are exactly those[7] for which $w(K) \leq 3.5$. So it remains to show that for any chain K with $w(K) > 3.5$, every non-void step taken by a node on that chain will decrease its width and will not create new chains with width greater than 3.5.

(a) Creates two 0-chains, with at least 2 0-coloured nodes less than the original chain, whereas the maximal penalty of the new endpoint for both chains is at most 1.6. Hence the width of both chains is less than the original.
(b) Does not change any 0-chains.
(c) Creates a 0-chain with width 1.
(d) If (d) is non-void, it changes 0_+, with width 1.1, to 0_- with width 1.
(e) Decreases the length of the chain by 1, but the endpoint may change from 000_- to 0_+0_+ resulting in an increase of penalty of at most 0.1.
(f) If 0_- in (f) is a complete 0-chain on its own, (f) changes this to chain 0_+0_- with width 3.5. Otherwise, it increases the length of the 0-chain (with endpoint 0_- and hence penalty 1.5) by 1 to a chain with endpoint 0_+0_- and hence penalty 0. So the width decreases by 0.5.
(g) Decreases the penalty of the endpoint from 1.5 to 0.
(h) Decreases the penalty of the endpoint from 1.6 to 1.5.
(i),(j) Do not change any 0-chains.

This completes the proof. ◁

All configurations matching the regular expression L_2 defined below only contain tokens in one direction.

$$L_2 = (O_2 I_2)^* \ , \qquad O_2 = 0 \mid 0_+0_- \ , \text{ and } \qquad I_2 = 1 \mid 1_+1_- \ .$$

Lemma 5.12 *From a configuration matching L_1, we eventually reach a configuration matching L_2. Furthermore, L_2 is closed.*

Proof First we prove closure. Let a node q with colour 0 take a step (the case of a node with colour 1 taking a step is handled similarly). Then q belongs to a chain matching O_2, because the current configuration matches L_2, and left and right of this chain there are chains matching I_2. We must show that any

[7] All chains of 4 or more 0's have width exceeding 4, $w(0_-00) \geq 4.5$, $w(0_+0_+) = 5.2$, $w(0_-0_+) = 3.5$, and $w(0_-0_-) = 5$.

step of q in any configuration matching L_2 yields a configuration matching L_2. That is, starting with a chain matching $I_2O_2I_2$ we get a chain matching $I_2O_2I_2$ or similar. This leads to the following case analysis (note that there are only two 0-chains matching O_2):

101	Becomes 10_1 by step (d), which matches $I_2O_2I_2$.
10_+0_1	Only node $0\_$ can move, but only if the right 1 has phase $-$. But for $1\_$ to match I_2 we must actually have the following: $10_+0\_1\_0$— matching $I_2O_2I_2O_2$—which becomes $10_+1_+1\_0$ after applying step (e). This string matches $I_2O_2I_2O_2$ again.

To prove that the system eventually reaches a configuration matching L_2, consider a configuration matching L_1. This configuration may fail to match L_1 for two reasons. First of all it may contain patterns like $0_+0\_0_+$ and $1_+1\_1_+$, and secondly it may contain tokens ($0_+0\_$ and $1_+1\_$) in opposite directions.

If it contains tokens in opposite directions, then there is at least one pair of opposite tokens pointing towards each other with no token in between. Then it is easily seen that after one round all nodes in between will have alternating colours (the only patterns in L_1 violating this are $0_+0\_0_+$ and $1_+1\_1_+$; steps (a) and (c) will colour these alternately). If some node in between takes a step, it will not change its colour, but will set its phase to $-$. After one round all these nodes will have phase $-$. In the next round, the tokens will move towards each other (because both have to apply step (e) or (f)). So the distance between them eventually becomes 0, i.e., a configuration like $0_+0\_1\_1_+$. If one of the middle nodes takes a step, this yields $0_+0\_0_+$ or $1_+1\_1_+$. This proves that eventually all opposing tokens are eliminated.

So we eventually reach a configuration in which all tokens have the same direction, and in which patterns $0_+0\_0_+$ and $1_+1\_1_+$ may occur. As these chains can only be created by tokens with opposite directions, no further chains of this kind will be created. These chains will eventually disappear, as the middle node can (and will) apply steps (a) or (c). Once all these chains have disappeared, we have reached a configuration matching L_2. ◁

All configurations matching the regular expression L_3 defined below are oriented.

$$L_3 = (O_3I_3)^* \ , \qquad O_3 = \overrightarrow{0} \mid \overrightarrow{0_+}\ \overrightarrow{0_-} \ , \text{ and} \qquad I_3 = \overrightarrow{1} \mid \overrightarrow{1_+}\ \overrightarrow{1_-} \ .$$

Lemma 5.13 *From a configuration matching L_2, we eventually reach a configuration matching L_3. Furthermore, L_3 is closed.*

Proof Closure is proven as in Lemma 5.12, noting that the orientation does not change in either case. To prove that the system eventually reaches a configuration matching L_3, note that because the ring has odd length, at least one token must exist, and that for at least one token we have 0_+0_-10 or 1_+1_-01. Thus the head of at least one token can eventually take a step, moving the token one position into the direction of the token, and directing the former head as well. Eventually one token must have travelled around the ring completely, at which time the ring is oriented. ◁

From Lemma 5.11, 5.12 and 5.13 we easily obtain the following theorem.

Theorem 5.14 *The ring-orientation protocol stabilizes, under the central daemon, to a configuration in which the ring is oriented, provided the length of the ring is odd.*

This theorem has a curious consequence. Dijkstra [Dij82] showed that no uniform deterministic self-stabilizing mutual exclusion protocol exists for rings of non-prime size. Burns and Pachl [BP89a] complemented this impossibility result with a uniform self-stabilizing mutual exclusion protocol for *oriented* rings of prime size, operating in the state-reading model under the central daemon. Combining the protocol of Burns and Pachl with our second ring-orientation protocol using fair protocol combination (cf. Sect. 5.3) proves the following theorem.

Theorem 5.15 *On* un-oriented *rings of prime size, self-stabilizing mutual exclusion can be achieved under the central daemon using a uniform protocol.*

Moreover, using the protocol of Itkis *et al.* [ILS95] instead, the total number of states per processor can even be kept constant.

5.6 On the equivalence of self-stabilizing system models

An overwhelming amount of models for distributed systems can be found in the literature, some of which only differ on seemingly minor points. This diversity is caused by the fact that slight alterations to a model may have

a huge effect on the (im)possibility or (in)efficiency of certain protocols[8]. A major challenge is to explore these different models, and find conditions or application areas under which these models are equivalent. In the area of self-stabilization, research in this area has already been started by Gouda *et al.* [GHR90]. In this section we will show that on oriented rings the link-register and state-reading model are equivalent. Using the results of the previous sections, we also show that, as a consequence, the link-register model and the state-reading are eventually equivalent on odd-length rings under the central daemon.

What do we mean by equivalence between models? Several definitions seem to be appropriate (cf. [GHR90, LV93b]). Intuitively two models are equivalent if they are of equal strength: whatever is possible in one model, is also possible in the other, and vice versa. We adopt the following formal definitions.

Definition 5.16 *Protocol* P_1 simulates *protocol* P_2 *if there exists a recursive mapping* μ *such that for all executions* E_1 *of* P_1, $\mu(E_1)$ *is an execution of* P_2. *Protocol* P_1 eventually simulates *protocol* P_2 *if there exists a mapping* μ *such that for all executions* E_1 *of* P_1, *a suffix of* $\mu(E_1)$ *is an execution of* P_2. *Protocols* P_1 *and* P_2 *are* (eventually) equivalent *if both (eventually) simulate each other. System model* M_1 (eventually) simulates *system model* M_2 *if for all deterministic protocols* P_2 *on* M_2 *there exists a deterministic protocol* P_1 *on* M_1 *that (eventually) simulates* P_2. *System models* M_1 *and* M_2 *are* (eventually) equivalent *if both (eventually) simulate each other.*

We choose to show simulation of M_2 by M_1 by giving a general method to convert a protocol in M_2 to a simulating protocol in M_1 and describing the mapping μ from executions in M_1 to M_2. As in the definition, both models are equivalent if we can show that both simulate each other.

Theorem 5.17 *Let* $G = (V, E)$ *be an undirected graph, where each node* $p \in V$ *labels its edges* $pq \in E$ *with* $lab_p(q)$. *Suppose there exists a function* f *such that for all* $pq \in E$ *we have* $lab_p(q) = f(lab_q(p))$, *and suppose that for all* $pq, pr \in E$ *with* $pq \neq pr$ *we have* $lab_p(q) \neq lab_p(r)$. *Then on* G, *the state-reading model and the link-register model are equivalent.*

[8] As exemplified by the large body of literature on how to reach agreement in the presence of faults.

Proof Consider an arbitrary node q in G. A protocol in the state-reading model is easily transformed into an equivalent protocol in the link-register model, by changing all writes to R_q into writes of the same value to all registers R_{qp} with $qp \in E$, and changing all reads from R_p, for some $qp \in E$, to reads from R_{pq}. Initially, for all $qp \in E$, the contents of R_{qp} should equal the contents of R_q. Then μ maps all executions in the link-register model to executions in the state-reading model by mapping the contents of R_{qp} for all $qp \in E$ (that by construction always hold the same value) to the contents of R_q.

A protocol in the link-register model is transformed into an equivalent protocol in the state-reading model as follows. Split, in the state-reading model, the register R_q into as many fields $\mathsf{R}_q.to[\cdot]$ as there are edges $qp \in E$. Replace a write to R_{qp} by a write of the same value to $\mathsf{R}_q.to[lab_q(p)]$. Change reads from R_{qp} to reads from $\mathsf{R}_q.to[f(lab_p(q))]$. Then μ maps all executions in the state-reading model to executions in the link-register model by mapping the contents of $\mathsf{R}_q.to[lab_q(p)]$ to the contents of R_{qp}. The result of applying μ is indeed an execution in the link register model because (i) if $qp \neq qr$ then $lab_q(p) \neq lab_q(r)$ so no write is overwritten by a wrong write, and (ii) $lab_p(q) = f(lab_q(p))$ so a value read from R_{qp} equals $\mathsf{R}_q.to[f(lab_p(q))] = \mathsf{R}_q.to[lab_q(p)]$ which equals the value written by a write to R_{qp}. ◁

Oriented rings, and also cliques and hypercubes with sense of direction have a labelling as in the above theorem. From the self-stabilizing ring-orientation protocol presented in the previous section, we easily obtain the following corollary.

Corollary 5.18 *For odd-length rings, the link-register and state-reading model are eventually equivalent under the central daemon.*

Simply combine the simulation in the proof of Theorem 5.17 with the ring-orientation protocol for the state-reading model. Note that the simulation of the state-reading protocol by the link-register protocol ensures that after one step of q after an error, the contents of R_{qp} and R_{qr} are equal. The ring-orientation protocol requires a central daemon.

Observe that a similar corollary cannot be obtained for odd-length rings under the distributed daemon, because Israeli and Jalfon already showed that no ring in the state-reading model can be oriented deterministically under the distributed daemon.

5.7 Conclusions and further research

We have shown that uniform deterministic self-stabilizing ring-orientation protocols using only a constant number of states per node exist for odd-length rings, both in the link-register model under the distributed daemon, and in the state-reading model under the central daemon. Further research might be directed at deriving similar protocols for other graphs with a regular structure, like cliques or hypercubes.

We have also shown that the link-register model and the state-reading model are eventually equivalent on odd length rings under the central daemon, showing that a self-stabilizing protocol designed for the one model can be transformed to an equivalent, self-stabilizing, protocol for the other model. We are unaware of similar theorems exploring the relation between the central daemon and the distributed daemon.

5.8 Acknowledgements

The author wishes to thank Marina Papatriantafilou and Philippas Tsigas for the discussions on this problem while they were visiting the CWI in Amsterdam.

Tanzen nutzlos in der Sonne
Macht dir nichts aus; Sie machen garaus
Ein Bild für die Götter
Schönen Gruß vom Schnitter

EINSTÜRZENDE NEUBAUTEN
Halber Mensch

6 Self-Stabilization of Wait-Free Shared Memory Objects

Abstract It is an interesting question whether one can device highly fault tolerant distributed protocols that tolerate both processor failures as well as transient memory errors. To answer this question we consider self-stabilizing wait-free shared memory objects. As a main contribution, we propose a general definition of self-stabilizing wait-free shared memory objects that expresses safety guarantees even in the face of processor failures. We prove that within this framework one cannot construct a self-stabilizing single-reader single-writer regular bit from single-reader single-writer safe bits. This impossibility result leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit as the minimal hardware needed to achieve self-stabilizing wait-free interprocess communication and synchronization. Based on this hardware, adaptations of well known wait-free implementations of regular and atomic shared registers are proven to be self-stabilizing. This paper is among the first to consider the interplay between processor failures and transient memory errors in a single failure model.

6.1 Introduction

The importance of reliable distributed systems can hardly be exaggerated. In the past, research on fault tolerant distributed systems has focused either on system models in which processors fail, or on system models in which the memory is faulty. In the first model a distributed system must remain operational while a certain fraction of the processors is malfunctioning. When constructing shared memory objects like, for instance, atomic registers, this is-

sue is addressed by considering *wait-free* constructions which guarantee that any operation executed by a single processor is able to complete even if all other processors crash in the meantime. Originally, research in this area focussed on the construction of atomic registers from weaker (safe or regular) ones [VA86, Lam86, PB87, LTV89, IS92]. Later attention shifted to stronger objects (cf. [AH90, Her91] and many others).

In the second model a distributed system is required to overcome arbitrary changes to its state within a bounded amount of time. If the system is able to do so, it is called *self-stabilizing*. Self-stabilizing protocols have been extensively studied in the past. Originating from the work of Dijkstra on self-stabilizing mutual exclusion on rings [Dij74], several other self-stabilizing protocols have been proposed for particular problems, like mutual exclusion on other topologies [BGW89, BP89a, DIM93], the construction of a spanning-tree [AKY90], orienting a ring [IJ93, Hoe94a], and network synchronization [AKM+93]. Another approach focuses on the construction of a 'compiler' to automatically transform a protocol belonging to a certain class to a similar, self-stabilizing, one [KP90, AKM+93]. For a general introduction to self-stabilization see [Sch93, Tel94].

To develop truly reliable systems both failure models must be considered together. We briefly summarize recent results that address this issue. Anagnostou and Hadzilacos [AH93] show that no self-stabilizing, fault-tolerant, protocol exists to determine, even approximately, the size of a ring. Gopal and Perry [GP93] present a 'compiler' to turn a fault-tolerant protocol for the synchronous rounds message-passing model into a protocol for the same model which is both fault-tolerant and self-stabilizing. A combination of self-stabilization and wait-freedom in the construction of clock-synchronization protocols is presented in [DW93, PT94].

Another approach to combining processor and memory failures is put forward by Afek *et al.* [AGM+92, AMT93] and Jayanti *et al.* [JCT92]. They analyse whether shared objects do or do not have wait-free (self)-implementations from other objects of which at most t are assumed to fail. Objects may fail by giving responses which are incorrect, or by responding with a special error value, or even by not responding at all. In so-called gracefully degrading constructions, operations on the 'high-level' object during which more than t 'low-level' objects fail are required to fail in the same manner as these 'low-level' objects.

We are interested in exploring the relation between self-stabilization and wait-freedom in shared memory objects. A shared memory object is a data structure stored in shared memory which may be accessed concurrently

by several processors through the invocation of operations defined for it. Self-stabilizing wait-free objects occur naturally in distributed systems in which both processors and memory may be faulty. We give a general definition of self-stabilizing wait-free shared memory objects, and focus on studying the self-stabilizing properties of wait-free shared registers. Single-writer single-reader safe bits—traditionally used as the elementary memory units to build these registers with—are shown to be too weak for our purposes. Focusing on registers, being the weakest type of shared memory objects, allows us to determine the minimal hardware needed for a system to be able to converge to legal behaviours after transient memory faults, as well as to remain operative in the presence of processor crashes.

Shared registers are shared objects reminiscent of ordinary variables, that can be read or written by different processors concurrently. They are distinguished by the level of consistency guaranteed in the presence of concurrent operations ([Lam86]). A register is *safe* if a read returns the most recently written value, unless the read is concurrent with a write in which case it may return an arbitrary value. A register is *regular* if a read returns the value written by a concurrent or an immediately preceding write. A register is *atomic* if all operations on the register appear to take effect instantaneously and act consistent with a sequential execution. Shared registers are also distinguished by the number of processors that may invoke a read or a write operation, and by the number of values they may assume. These dimensions imply a hierarchy with single-writer single-reader ($1W1R$) binary safe registers (a.k.a. bits) on the lowest level, and multi-writer multi-reader ($nWnR$) l-ary atomic registers on the highest level. A *construction* or *implementation* of a register is comprised of i) a data structure consisting of memory cells called *sub-registers* and ii) a set of read and write procedures which provide the means to access it.

Li and Vitányi [LV91] and Israeli and Shaham [IS92] were the first to consider self-stabilization in the context of shared memory constructions. Both papers implicitly call a shared memory construction self-stabilizing if for every *fair* run started in an arbitrary state, the object behaves according to its specification except for a finite prefix of the run. We feel that this notion of a self-stabilizing object does not agree well with the additional requirement that the object is wait-free. On wait-free shared objects, a single processor can make progress even if all other processors have crashed. This definition of self-stabilization, on the other hand, only guarantees recovery from transient errors in fair runs (in which no processors crash). Moreover, both Li and Vitányi [LV91] and Israeli and Shaham [IS92] do not consider the possibility of

pending operations. And it is exactly these pending operations that make the definition and implementation of self-stabilizing shared objects an intricate problem.

Our contribution in this chapter is threefold. First, in Sect. 6.2, we propose a general definition of a self-stabilizing wait-free shared memory object, that ensures that all operations after a transient error will eventually behave according to their specification even in the face of processor failures. Second, in Sect. 6.3, we prove that within this framework one cannot construct a self-stabilizing single-reader single-writer regular bit from single-reader single-writer safe bits—which have traditionally been used as the basic building blocks in wait-free shared register implementations. This impossibility result leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit, which, from a hardware point of view, resembles a flip-flop with its output wire split in two (cf. Sect. 6.4). Using this bit as a basic building block, we formally prove, as a third contribution, that adaptations of well known wait-free implementations of regular and atomic shared registers are self-stabilizing (cf. Sects. 6.4.1, 6.4.2, and 6.4.3). This shows that our definition of self-stabilizing wait-free shared objects is viable—in the sense that it is neither trivial nor impractical. Section 6.5 concludes this chapter with directions for further research.

6.2 Defining self-stabilizing wait-free objects

In the definition of shared memory objects we follow the concept of *linearizability* (cf. [HW90, Her91]), which we, for the sake of self containment, briefly paraphrase here. Consider a distributed system of n sequential processors. A shared memory object is a data-structure stored in shared memory that may be accessed by several processors concurrently. Such an object defines a set of *operations* $\mathcal{O}$ which provide the only means for a processor to modify or inquire the state of the object. The set of processors that can invoke a certain operation may be restricted. Each operation $O \in \mathcal{O}$ takes zero or more parameters p on its invocation and returns a value r as its response ($r = O(p)$). Each such operation execution is called an *action*, and takes a non-zero amount of time to complete. We denote by $t_I(A) \geq 0$ the invocation time of an action A and by $t_R(A) > t_I(A)$ its response time (on the real time axis).

When implementing compound objects from lower level ones, the operations of the compound object are implemented by means of a sequential procedure that can invoke an operation on one of the primitive objects or do some local computations. The implementation must be such that from

the known properties of the primitive objects and the order of the steps taken by the procedure, correct behaviour of the compound object can be proven. Processors are sequential and, therefore, cannot invoke an action if their previously invoked action has not responded yet.

To model processor crash failures we introduce for each processor p the *crash action* ψ_p. No invocation or response of an action at processor p, nor another crash action ψ_p may occur later than the time $t(\psi_p)$ processor p crashes. In terms of the implementation of an object, no sub-operations may be executed by p after the crash action ψ_p either. Note that these crash actions are only needed in the definitions that are to follow. Because we study asynchronous systems, these crash actions are not observable by any processor.

The desired behaviour of an object is described by its *sequential specification* S. This specifies the set of possible states of the object, and for each operation its effect on the state and its (optional) response. We write $(s, r = O(p), s') \in S$ if invoking O with parameters p in state s changes the state of the object to s' and returns r as its response[1].

A *run* over the object is a tuple $\langle \mathcal{A}, \rightarrow \rangle$ with actions $\mathcal{A}$ and partial order $\rightarrow$ such that for $A, B \in \mathcal{A}$, $A \rightarrow B$ iff $t_R(A) < t_I(B)$. Similarly, $\psi_p \rightarrow A$ iff $t(\psi_p) < t_I(A)$. If two actions A, B are incomparable under $\rightarrow$, i.e., if neither $A \rightarrow B$ nor $B \rightarrow A$, they are said to *overlap*. Then we write $A \| B$. Runs have infinite length, and capture the real time ordering — as could be observed externally without a stopwatch — between actions invoked by the processors. An implementation of a shared object is *wait-free* if in all runs each invocation of an action A is followed by a matching response after finite time (i.e., when $0 < t_R(A) - t_I(A) < \infty$), unless the processor p invoking A crashed at time $t(\psi_p)$ such that $0 < t(\psi_p) - t_I(A) < \infty$.

A *sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$ over the object is an infinite sequence $s_1 A_1 s_2 A_2 \ldots$, where $\bigcup_i A_i = \mathcal{A}$, s_i a state of the object as in its sequential specification, and $\Rightarrow$ a total order over $\mathcal{A}$ defined by $A_i \Rightarrow A_j$ if and only if $i < j$. A run $\langle \mathcal{A}, \rightarrow \rangle$ *corresponds* with a sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ if the set of actions $\mathcal{A}$ is the same in both runs, and if $\Rightarrow$ is a total extension of $\rightarrow$ (i.e., $A \rightarrow B$ implies $A \Rightarrow B$). Stated differently, the sequential execution corresponding to a run is a run in which no two actions are concurrent but in which the 'observable' order of actions in the run is preserved. Note that there may be more than one sequential execution corresponding to a single

[1] Although the notation might imply otherwise, we consider $O(p)$ and $O(q)$ to be *different* operations.

run, because the order between two concurrent actions can be fixed either way.

Definition 6.1 *A run* $\langle \mathcal{A}, \rightarrow \rangle$ *over an object is* linearizable w.r.t. sequential specification S, *if there exists a corresponding sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$, *such that* $(s_i, A_i, s_{i+1}) \in S$ *for all* i.

An object is linearizable w.r.t. its sequential specification S if all possible runs over the object are linearizable w.r.t. S. Informally speaking, an object is linearizable w.r.t. to specification S if all actions appear to take effect instantaneously and act according to S. The challenge in implementing such linearizable objects is to avoid all non-linearizable runs.

6.2.1 Adding self-stabilization

Li and Vitányi [LV91] and Israeli and Shaham [IS92] were the first to consider self-stabilizing wait-free constructions. Both papers implicitly use the following straightforward definition of a self-stabilizing wait-free object.

Definition 6.2 *A shared wait-free object is self-stabilizing if every* fair *run (in which all operations on all processors are executed infinitely often) that is started in an arbitrary state, is linearizable except for a finite prefix.*

A moment of reflection shows that assuming fairness may not be very reasonable for wait-free shared objects. The above definition requires that after a transient error *all* processors cooperate to repair the fault. This clearly violates the wait-free property which states that processors can make sensible progress even if other processors have crashed. This observation leads us to the following stronger, still informal, definition of a self-stabilizing wait-free shared object.

Definition 6.3 *A shared wait-free object is self-stabilizing, if every run started in an arbitrary state is linearizable except for a bounded finite prefix.*

Let us develop a formal version of this definition. To model self-stabilization we need to allow runs that start in an arbitrary state; in particular we have to allow runs in which a subset of the processors start executing an action at an arbitrary point within its implementation. Such runs model the case in which transient memory errors occur during an action, or, rather, the

case where alteration of the program counter by the transient error forces the processor to jump to an arbitrary point within the procedure implementing the operation. For such so called *pending* actions A, and for such actions alone, we set $t_I(A) = 0$.

Slow pending actions can carry the effects of a transient error arbitrarily far into the future[2]. Hence we can only say something meaningful about that part of a run after the time that all pending actions have finished, or the processors on which these pending actions run have crashed.

An action A *overlaps a pending action* iff there exists a pending action B on some processor p (hence, by definition, $t_I(B) = 0$) such that $t_I(A) < t_R(B)$ and $t_I(A) < t(\psi_p)$. Define $\mathsf{count}(A)$ equal 0 for all actions A overlapping a pending action, and define $\mathsf{count}(A)$ equal to i if A is executed as the i-th action of a certain processor not overlapping a pending action. As a special case—to be used later—for all actions A with $\mathsf{count}(A) = 0$ for which there exists a B with $\mathsf{count}(B) = 1$ and $A \| B$ set $\mathsf{count}(A) = \odot$ instead (where $0 < \odot < 1$). With these definitions, actions A with $\mathsf{count}(A) = \odot$ overlap both with pending actions and with actions not overlapping any pending actions. As each processor executes sequentially, and actions are unique, count is well-defined.

We are now ready to present the formal definition of a self-stabilizing wait-free shared memory object.

Definition 6.4 *A run* $\langle \mathcal{A}, \rightarrow \rangle$ *is* linearizable w.r.t. sequential specification S after k processor actions, *if there exists a corresponding sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$, *such that for all* i, *if* $\mathsf{count}(A_i) > k$ *then* $(s_i, A_i, s_{i+1}) \in S$.

This definition allows all actions overlapping pending actions, and the first k actions of each processor *not* overlapping pending actions to behave arbitrarily (even so far as to allow e.g. a read action to behave as a write action or vice versa). However, the effect of such an arbitrary action should be globally consistent. For instance, all fast processors must agree — after k operations — on the arbitrary behaviour of the slow processors. In particular, for $k = 0$

[2] A pending action may carry something "malicious" in its local state that might disorder the system at any time after it is used to modify a sub-register. Consider for instance the Vitányi-Awerbuch register (cf. [VA86] and Sect. 6.4.3). If a pending action writes a huge tag only to the last register S_{in} in the row, no later action except one executed by processor n will see this tag. If writes after the pending action use a lower tag, they will be ignored by actions of processor n, even if these writes occur strictly before the actions of processor n.

and in the absence of pending actions, the definition implies that all actions agree on the effect of the transient error on the state of the object. For example, for a 0-stabilizing shared register all reads that occur immediately after a transient error should return the same value.

Definition 6.5 *A shared object is* k-stabilizing wait-free *with sequential specification* S *if all its runs are wait-free and linearizable w.r.t.* S *after* k *processor actions.*

In the above definition the stabilization delay k is taken to be independent of the type of operations performed by a processor, while one might very well feel that the difficulty of stabilizing different types of operations on the same object may vary. Indeed, preliminary versions of this definition were more fine-grained and included separate delays for different types of operations (e.g. allowing the first k_w writes and the first k_r reads performed by a processor on a read/write register to be arbitrary). It turns out that this amount of detail is really unnecessary, essentially because different types of operations on a shared object already need to reach some form of agreement on the state of the object.

Definition 6.5 is general and considers all atomic objects whose behaviour is described by a sequential specification. Of course, we want to know whether such self-stabilizing shared objects exist and how they can be implemented. Traditionally, in the non self-stabilizing case, atomic objects have been built using several layers. Starting with safe registers as the minimal hardware available for interprocess communication, regular registers, multi-reader registers and finally atomic registers were constructed. We take a similar approach. Therefore, we need to define when safe and regular registers are self-stabilizing.

A register is a shared object on which read operations R and write operations $W(v)$ are defined. For a run $\langle \mathcal{A}, \rightarrow \rangle$ over such a register, identify the set $\mathcal{R}$ of read actions that do not act as writes, and identify the set $\mathcal{W}$ of write actions plus read actions that do act as writes. For $R \in \mathcal{R}$ define $\mathrm{val}(R)$ as the value returned by read action R and for $W \in \mathcal{W}$ define $\mathrm{val}(W)$ as the value 'actually' written by action W. Also for $W \in \mathcal{W}$ and $R \in \mathcal{R}$ define W *directly precedes* R, $W \rightleftarrows R$, if $W \rightarrow R$ and if there is no $W' \in \mathcal{W}$ such that $W \rightarrow W' \rightarrow R$. If no such write exists, we take the imaginary initial write $W_\perp$ responsible for writing the arbitrary initial value $\mathrm{val}(W_\perp)$. Define the *feasible* writes of a read R as all $W \in \mathcal{W}$ such that $W \rightleftarrows R$ or $W \| R$.

A write $W(v)$ on a register behaves correctly if $\mathsf{val}(W(v)) = v$. A read R on a *safe* register behaves correctly if $R \in \mathcal{R}$ and $\mathsf{val}(R) = \mathsf{val}(W)$ for a write W with $W \rightleftarrows R$, or there is a write W such that $W \| R$. A read R on a *regular* register behaves correctly if $R \in \mathcal{R}$ and $\mathsf{val}(R) = \mathsf{val}(W)$ for some feasible write of R.

Definition 6.6 *A safe or regular register is* k-stabilizing wait-free *if all its runs are wait-free and for all its runs the set of actions can be partitioned in sets* $\mathcal{R}$ *and* $\mathcal{W}$ *such that only actions* A *with* $\mathsf{count}(A) \leq k$ *behave incorrectly. Such a register is simply* stabilizing wait-free *if all its runs are wait-free and for all its runs only pending write actions* W *and read actions* R *overlapping pending writes behave incorrectly* without *behaving as a write* (*i.e.,* $R \in \mathcal{R}$)[3].

6.3 Some impossibility results

The impossibility results to be presented in this section help to set the scene for the actual constructions of stabilizing shared registers in the sections to come. First we prove that stabilizing $1W1R$ safe bits[4] are not strong enough to implement k-stabilizing (or stabilizing) wait-free regular registers. This immediately implies that stronger self-stabilizing objects, like atomic registers, cannot be implemented using such safe registers, either. Then we show that, when implemented using stabilizing safe registers, reads of stabilizing regular registers overlapping a pending write cannot behave correctly. Hence the construction of a stabilizing $1W1R$ regular bit from a stabilizing $1W2R$ safe bit in Prot. 6.1 is optimal. Finally, we show that 0-stabilizing $nWnR$ atomic registers — and similar multi-user objects — cannot be constructed from stabilizing $1W1R$ registers (of arbitrary type). This shows that the construction of a 1-stabilizing $nWnR$ atomic register from stabilizing $1W1R$ regular registers in Prot. 6.3 cannot be made to stabilize faster. We start with a few introductory remarks before presenting the theorems.

It is a common convention to view the scheduling of processor steps as being chosen by an *adversary*, who seeks to force the protocol to behave incorrectly. The adversary is in control of (i) choosing the configuration of the system after a transient error and (ii) scheduling the processors' steps in

[3] Actually, for safe registers a read overlapping a pending write never behaves incorrectly, because safeness allows it to return an arbitrary value.

[4] All objects considered in this chapter are wait-free; for brevity we will not always explicitly mention this when referring to an object.

a run.

We assume that the value of the register depends only on the state of its implementation, i.e., for any configuration, the value of the register in that configuration is the value that would be returned by the $k+1$-th read of a sequence of reads taking place in a time interval during which no other processors are active. Similarly, we assume that the behaviour of an operation solely depends on the configuration in which it is executed and not in its position in the run. If a read behaves as a write in some state, it must behave as a write in that state wherever that state occurs in the run.

Theorem 6.7 shows that stabilizing $1W1R$ safe bits are not strong enough to build wait-free stabilizing $1W1R$ regular registers. The gist of the argument runs as follows. Since the writer cannot read the sub-registers it writes, in order to know their contents and converge into correct stabilized behaviours, it has to rely on information that either is local or is passed to it through shared sub-registers that can be written only by the reader. The same reasoning holds for the registers written by the reader. The adversary can set the system in a state in which this information is inconsistent. Subsequently, by scheduling the sub-actions of both the reader and the writer on the same sub-register to be concurrent, the adversary can destroy the information propagation because of the weak consistency that safeness guarantees.

Theorem 6.7 *There exists no deterministic implementation of a k-stabilizing wait-free $1W1R$ binary regular register using stabilizing $1W1R$ binary safe sub-registers.*

Proof Suppose that such an implementation exists. Since we look for a contradiction we may safely restrict attention to runs with no pending actions.

Any implementation of a $1W1R$ binary regular register from stabilizing $1W1R$ safe binary sub-registers must use two sets of sub-registers (that can be considered as two "big" sub-registers): one (S_W) that is written by the writer and read by the reader and one (S_R) that is written by the reader and read by the writer. Then the whole state of the implementation is described by a configuration $C = (lr, lw, sr, sw)$, where lr, lw denote the reader's and writer's local states, and sr, sw denote the contents of S_R, S_W, respectively.

A read action on the regular register may involve several sub-reads of S_W; however, in the course for a contradiction, attention may be restricted to runs in which all those sub-reads observe the same value of S_W. Then the value returned by each read is determined by a *reader function* $F(lr, sw)$. Furthermore, let $F^x(lr, sw)$ denote the value returned by the x-th read of a

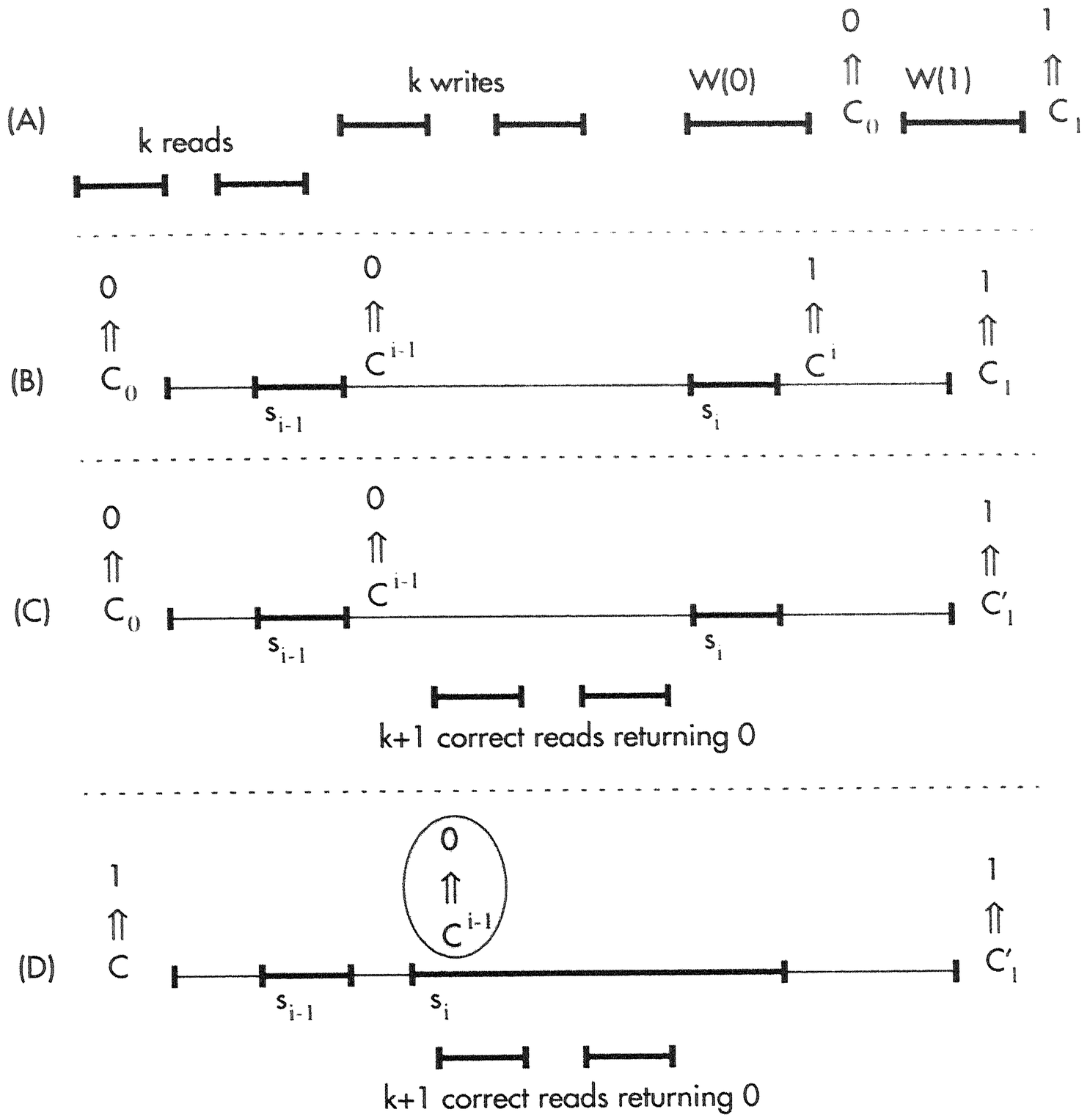

Figure 6.1 *The runs constructed in the proof of Theorem 6.7.*

sequence of non-interfered reads that start from a configuration with local state lr while all reads find $S_W = sw$. In Fig. 6.1 $C \Rightarrow v$ denotes $F^{k+1}(C) = v$.

Consider an arbitrary initial configuration and a run in which the following actions are sequentially scheduled (see Fig. 6.1 (A)): k reads, k writes (writing arbitrary values), a write action $W(0)$ writing 0 and a write action $W(1)$ writing 1. After $W(0)$ the system configuration is $C_0 = (lr, lw_0, sr, sw_0)$, while after $W(1)$ the system configuration is $C_1 = (lr, lw_1, sr, sw_1)$.

Since the register is k-stabilizing by assumption, $W(0)$ and $W(1)$ must behave correctly and hence the last of $k+1$ reads starting in C_0 must return 0 (so $F^{k+1}(lr, sw_0) = 0$) and the last of $k+1$ reads starting in C_1 must return 1. Therefore, during $W(1)$ the writer performs some sub-writes $s_1, \ldots, s_m$ on bits of S_W in that order.

Let us write $sw \uparrow s_1 \ldots s_i$ to denote the value of S_W after sub-writes $s_1, \ldots, s_i$ have been applied, while S_W held sw initially. With this definition, $F^{k+1}(lr, sw_0 \uparrow s_1 \ldots s_m) = 1$. Hence, during $W(1)$ there is an s_i $(1 \leq i \leq m)$, such that

$$F^{k+1}(lr, sw^{i-1} = sw_0 \uparrow s_1 \ldots s_{i-1}) = 0 \text{ , and} \tag{6.1}$$

$$F^{k+1}(lr, sw^{i} = sw_0 \uparrow s_1 \ldots s_{i}) = 1 \text{ .} \tag{6.2}$$

Let $C^{i-1} = (lr, lw_0, sr, sw^{i-1})$ and $C^i = (lr, lw_0, sr, sw^i)$. See Fig. 6.1 (B).

Now consider a run in which $k+1$ reads are started in C^{i-1} just before $W(1)$ starts s_i. Because the register is k-stabilizing, all these behave correctly, i.e., return 0 and do not behave as a write (see Fig. 6.1 (C)).

Now let lr' be the local state of the reader after those reads and sw'_1 the contents of S_W after completion of W_1 in this run (configuration C'_1). Again because the register is k-stabilizing its value in C'_1 must be 1, and hence

$$F^{k+1}(lr', sw'_1) = 1 \text{ .} \tag{6.3}$$

Now let the adversary set the system in configuration $C = (lr, lw_0, sr, sw^i)$, i.e., differing from C_0 only in the contents of S_W, and schedule again a $W(1)$ action. By our assumption and Eq. (6.2), the value of the register in C equals 1. The writer, however, observes the same state as in C_0 and hence performs the same sequence of subwrites.

The adversary can again schedule $k+1$ reads, now overlapping s_i instead of just preceding it (see Fig. 6.1 (D)). Because the bit written by s_i is safe, and sw^{i-1} and sw^i only differ in the bit written by s_i, all these $k+1$ reads may observe sw^{i-1} instead of sw^i. Hence the $k+1$ reads observe C^{i-1} as in case (C), and $W(1)$ continues as in the previous schedule in which these reads did execute, yielding configuration C'_1. We know from the previous paragraph and our initial assumption that none of these reads perform as a write, while the $k+1$-th read returns 0 (Eq. (6.1)). But the only feasible writes are a write of 1 (recall Eq. (6.3)) and the write of the initial value, which is 1 (Eq. (6.2)). This is a contradiction, as the $k+1$-th read must return the value of a feasible write. ◁

The next theorem states that we cannot construct stabilizing regular

registers from stabilizing safe registers such that even the reads overlapping a pending write behave correctly (either returning the value written by the pending write or the value initially present as created by the transient error). This shows that Def. 6.6 is tight, and that Protocols 6.1 and 6.2 to be presented next are optimal.

Theorem 6.8 *There exists no deterministic implementation from stabilizing wait-free safe registers of a stabilizing wait-free* $1W1R$ *regular register whose reads overlapping a pending write must behave correctly.*

Proof We prove this by contradiction, showing that if such an implementation exists using x registers, there must also exist an implementation using $x - 1$ registers. Because for any communication between two processors at least 1 register is necessary, we arrive at a contradiction.

Suppose the writer writes at least one safe register. Let the adversary generate an error such that a pending write W starts writing an arbitrary value to a register S, after which the pending write W finishes. The adversary sets the value of S just after the error equal to the value written by W. Then, whatever the contents of the other registers are, the configuration before and after W is the same, hence the value of the register (as returned by a non-interfered read) at the time of the error and the value of the register immediately after write W is the same. Call this value v. Hence, by regularity, every (non pending) read action overlapping only W must return v.

Now let the adversary schedule a read action R that starts and completes within the interval of time W writes the safe register S. Read R only overlaps the pending write, and hence must return v. Then either R does not read S, or it reads an arbitrary value from S (because R is concurrent with the write to register S, and because S is safe). Either way, the value R returns cannot depend on the value read from S, but must depend only on the other registers it reads. Because the contents of those registers where chosen completely arbitrary, the value of S is shown never to influence the outcome of a read. But then the writer never needs to write S, and hence the number of registers it writes can be decreased by one. ◁

Theorem 6.9 *For* $n > 0$ *and* $l > 1$ *there does not exist a deterministic implementation of a* 0*-stabilizing wait-free* $nWnR$ *l-ary atomic register from stabilizing* $1W1R$ *registers (of arbitrary type).*

Proof Assume all subregisters are $1W1R$. Then reads of the atomic register

executed by different processors must obtain a value by reading disjoint sets of subregisters. For every processor and each possible return value v of a read, the adversary can set the configuration of the processor (i.e., local state plus values read from the subregisters) such that if this processor executes its first read without any interference, this read returns v.

Suppose, to the contrary, that such a 0-stabilizing wait-free implementation of a $nWnR$ l-ary atomic register A exists. W.l.o.g. assume that 0 and 1 are among the values stored in register A. Let p_0 and p_1 be two processors accessing A, and let the adversary set the configuration of p_0 such that its first read on its own will return 0, while the configuration of p_1 is such that its first read on its own will return 1. Consider all runs over A with only two actions: a read R_0 executed by p_0 and a read R_1 executed by p_1, both nonpending. Then according to Def. 6.5 in all such runs R_0 and R_1 must return the same value. Also, there is a run (the one where R_0 executes alone) where 0 is the return value, and there is a run (the one where R_1 executes alone) where 1 is the return value.

This construction now can be used to solve the two processor consensus problem[5] as follows. Consider two copies A^0 and A^1 of the above 0-stabilizing wait-free implementation of a $nWnR$ l-ary atomic register. Let the initial configuration of A^0 be such that read $R_0(\mathsf{A}^0)$ (of processor p_0) on its own returns 0, and the read $R_1(\mathsf{A}^0)$ (of processor p_1) on its own returns 1. Similarly, let the initial configuration of A^1 be such that read $R_0(\mathsf{A}^1)$ (of processor p_0) on its own returns 1, and the read $R_1(\mathsf{A}^1)$ (of processor p_1) on its own returns 0.

The protocol for p_i, where $i \in \{0, 1\}$, to propose a value $v \in \{0, 1\}$ simply is to read and return $R_i(\mathsf{A}^{i \oplus v})$[6]. To see that this protocol solves 2-processor consensus consider the following two cases.

If p_0 and p_1 propose complementary values v and $1 \oplus v$ respectively, then both read the *same* register A^v, and by the observation of the second to last paragraph both must return the same value. This value is either v or $1 \oplus v$, both of which are proposed.

If p_0 and p_1 propose the same value v, they will *not* read the same register, and hence these reads will execute on their own. If p_i proposes v, it reads $R_i(\mathsf{A}^{i \oplus v})$ which, according to the initializations described in the previous

---

[5] In the consensus problem each process starts with a private input value, and all processes must, at some point, irrevocably decide on some output value. All processes must decide on the same value, and this value must have been the input value to some process.

[6] $\oplus$ denotes the bitwise exclusive or.

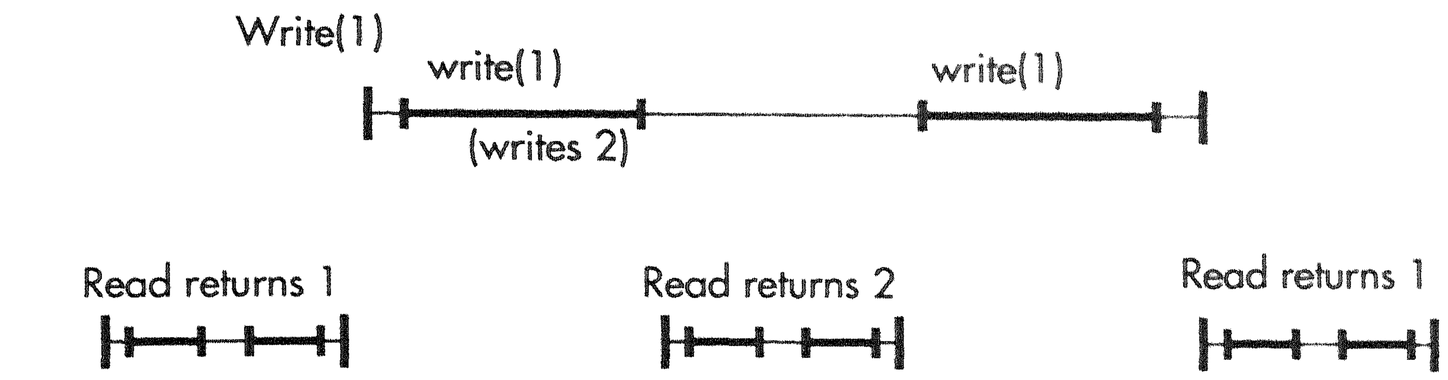

Figure 6.2 *Repeating actions does not make an object 1-stabilizing.*

paragraph, will return v if executed on its own. Hence both processors decide v as required.

Loui and Abu-Amara [LAA87] showed that deterministically solving 1-resilient consensus using read/write registers is impossible. We conclude that therefore a deterministic implementation from stabilizing $1W1R$ registers of a 0-stabilizing wait-free $nWnR$ l-ary atomic register does not exist. ◁

It is straightforward to generalize Theorem 6.9 and its proof to similar multi-user objects that are known not to be strong enough to solve consensus (e.g. regular registers and atomic snapshot memories).

Using Theorem 6.9 and the construction of a 1-stabilizing wait-free $nWnR$ atomic register from stabilizing $1W1R$ regular ones in Prot. 6.3 we arrive at the following corollary.

Corollary 6.10 *There does not exist a general deterministic method to transform a k-stabilizing wait-free shared object into a* 0*-stabilizing wait-free object using only read/write registers.*

We also would like to point out that a k-stabilizing shared object cannot simply be transformed into a 1-stabilizing one by executing each operation k times in a row. Consider the counterexample in Fig. 6.2 for $k = 2$. The first read (before the write) returns 1, as well as the third read after the write. Then the write must have behaved as a write of 1. This contradicts with the fact that the second read returns 2 due to the fact that the first subwrite behaves arbitrary and writes 2 instead of 1.

6.4 Self-stabilizing constructions of shared registers

To construct complex self-stabilizing shared memory objects, single-reader single-writer stabilizing safe-bits are not strong enough according to Theorem 6.7. This impossibility result is based on the fact that the processors cannot read their own shared variables and that any information exchanged between them can be destroyed by the weakness of the safeness property. However, if we assume the existence of stabilizing *dual-reader* single-writer safe bits, this reasoning does not apply: to know the value of its own shared bits the writer can simply read them. We believe there is no fundamental difference between assuming that a $1W1R$ safe bit exists and assuming that a $1W2R$ safe bit exists. After all, the first models a flip-flop with a single output wire, whereas the latter models a flip-flop with its output wire split in two.

We prove in the next sections that if these $1W2R$ safe bits are used as basic building blocks in some well-known wait-free constructions of shared registers, the resulting constructions do become, after minor modifications, self-stabilizing.

A few words on the programming notation are in order. We use := to denote assignment, *name* to denote local variables, NAME to denote shared variables, and **name** for keywords. We choose not to mention the *Read*() or *Write*() actions on a shared variable explicitly. Instead, R := v is 'syntactic sugar' for *Write*(R, v). Similarly, if R occurs in an expression, *Read*(R) is implied. As usual, both actions may take some time to complete.

6.4.1 A stabilizing $1W1R$ regular bit

Protocol 6.1 presents the adaptation of Lamport's [Lam86] construction of a $1W1R$ regular bit from a $1W1R$ safe bit, into a stabilizing wait-free one using a wait-free stabilizing $1W2R$ safe bit. Instead of relying on a local copy of the value of the bit, the writer now actually reads the bit (over its second 'wire') to determine whether its value has to be changed. We proceed by proving correctness of the protocol.

Theorem 6.11 *Protocol 6.1 implements a wait-free stabilizing* $1W1R$ *regular binary register using one wait-free stabilizing* $1W2R$ *safe binary register.*

Proof Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run of reads R and writes W over the regular bit. By Prot. 6.1 this induces an order $\rightarrow$ over the safe bit S. Write R(S) (W(S)) for the read from (write to) the safe bit S performed by read R (write

S: stabilizing $1W2R$ safe bit

operation *Read*() : {0, 1}
1 **return** S ;

operation *Write*(v : {0, 1})
2 **if** S $\neq v$
3 **then** S := v ;

Protocol 6.1 *A stabilizing 1W1R regular bit.*

W) on the regular bit. Let $w_\perp$ be the initializing write of safe bit S, and set $\mathsf{val}(W_\perp) = \mathsf{val}(w_\perp)$ for the initializing write of the regular bit. If $\langle \mathcal{A}, \rightarrow \rangle$ has a pending write W, set $\mathsf{val}(W)$ to the value of S just after W; this is the value an interference free read starting after W will read. For all other, non-pending, writes set $\mathsf{val}(W(v)) = v$.

According to Def. 6.6 it remains to show that in $\langle \mathcal{A}, \rightarrow \rangle$ all reads R not overlapping a pending write return the value written by a feasible write. Reads that do overlap a pending write may behave arbitrary, but clearly never can behave as writes. Protocol 6.1 is trivially wait-free.

Claim 6.12 *If R does not overlap a pending write and $R(\mathsf{S})$ is interference free and $W \rightleftarrows R(\mathsf{S})$ then* $\mathsf{val}(W) = \mathsf{val}(R(\mathsf{S}))$.

Proof Let $W \rightleftarrows R(\mathsf{S})$ and let $\mathsf{val}(R(\mathsf{S})) = a$. If $W = W_\perp$ or W is a pending write, then $\mathsf{val}(W) = a$ by definition. Otherwise, note that a write $W(x)$ reads S by $R'(\mathsf{S})$ without interference. Either $\mathsf{val}(R'(\mathsf{S})) = x$ so W does not write S or $\mathsf{val}(R'(\mathsf{S})) = \neg x$ and W does write x to S by $W(\mathsf{S})$.

In the first case, as S is stabilizing and $R'(\mathsf{S})$ and $R(\mathsf{S})$ are not pending, and do not overlap a pending write, and there cannot be another write to S between $R'(\mathsf{S})$ and $R(\mathsf{S})$, $\mathsf{val}(R(\mathsf{S})) = \mathsf{val}(R'(\mathsf{S})) = x = \mathsf{val}(W)$.

In the second case, as S is stabilizing and $W(\mathsf{S})$ not pending, and $R(\mathsf{S})$ is not pending and does not overlap a pending write, and $W(\mathsf{S}) \rightleftarrows R(\mathsf{S})$, $\mathsf{val}(R(\mathsf{S})) = \mathsf{val}(W(\mathsf{S})) = x = \mathsf{val}(W)$. ◁

Consider a read R not overlapping a pending write. For this read, $\mathsf{val}(R) = \mathsf{val}(R(\mathsf{S}))$.

If $R(\mathsf{S})$ is interference-free, then by Claim 6.12, for a write W with $W \rightleftarrows R(\mathsf{S})$ we have $\mathsf{val}(W) = \mathsf{val}(R(\mathsf{S})) = \mathsf{val}(R)$ and W is feasible for R.

If $R(\mathsf{S})$ is interfered, there is a write $W(x)$ with $W \| R$ writing S and W cannot be pending by assumption. W reads S by $R'(\mathsf{S})$ and $\mathsf{val}(R'(\mathsf{S})) = \neg x$.

By Claim 6.12 and the fact that $R'(\mathsf{S})$ cannot overlap another write (because it is executed by one), for W' with $W' \rightleftarrows W$ we must have $\mathsf{val}(W') = \neg x$. As $W \| R$, then $W' \rightleftarrows R$ or $W' \| R$, so both W and W' are feasible for R. One of these writes writes 0 and the other writes 1, so $\mathsf{val}(R)$ equals the value written by a feasible write. ◁

6.4.2 A stabilizing $1W1R$ l-ary regular register

Protocol 6.2 presents the adaptation of Lamport's [Lam86] construction of a $1W1R$ l-ary regular register from l $1W1R$ regular bits, into a wait-free stabilizing one using l $1W1R$ wait-free stabilizing regular bits. It differs from the original construction in that a check is added to detect that the end of the bit array is reached without reading a 1 in any of the bits, in which case a default value $(l-1)$ is returned. We prove correctness of the protocol below.

Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run over the regular l-ary register. By Prot. 6.2 this induces an order $\rightarrow$ on the actions on the regular bits $\mathsf{S}^0, \ldots, \mathsf{S}^{l-1}$ Number the writes consecutively, writing W^i for the write with index i. Let W^0 be the pending write if it exists, and $W_\perp$ otherwise. Let us write $R(\mathsf{S}^v)$ for the read of S^v by read R, and let us write $W(\mathsf{S}^v)$ for the write to S^v by a write W. The index of $W(\mathsf{S}^v)$ equals the index of W (and the index of $w_{v,\perp}$ always equals 0). For reads R of a subregister S^v not overlapping a pending write, define $\pi(R)$ to be the largest index i such that W^i (of S^v) is feasible for R and $\mathsf{val}(W^i) = \mathsf{val}(R)$.

Consider the values of all S^v just after W^0 (i.e., the value read by an non-interfered read starting after W^0). Set $\mathsf{val}(W^0)$ to the minimal v such that $\mathsf{S}^v = 1$, setting $\mathsf{val}(W^0) = l-1$ if no such v exists. For all other writes $W(v)$ set $\mathsf{val}(W(v)) = v$. We first prove the following claim:

Claim 6.13 *Let R be a read not overlapping a pending write. If $W^i(u) \rightarrow R$ then $\pi(R(\mathsf{S}^v)) \geq i$ for all $v \leq u$. If R reads S^{w+1} then $\pi(R(\mathsf{S}^w)) \leq \pi(R(\mathsf{S}^{w+1}))$.*

Proof The first part easily follows from Prot. 6.2 as, for $i > 0$, $W^i(u)$ writes to all registers S^v with $v \leq u$.

For the second part, note that if R reads S^{w+1}, then $\mathsf{val}(R(\mathsf{S}^w)) = 0$. If $\pi(R(\mathsf{S}^w)) = i > 0$ (if $i = 0$ we are done immediately) then $\mathsf{val}(W^i(\mathsf{S}^w)) = 0$ and hence (by Prot. 6.2 and the fact that W^i for $i > 0$ is not pending) W^i must write to S^{w+1} as well. Now by $W^i(\mathsf{S}^w) \not\leftarrow R(\mathsf{S}^w)$ and $W^i(\mathsf{S}^{w+1}) \rightarrow W^i(\mathsf{S}^w)$ and $R(\mathsf{S}^w) \rightarrow R(\mathsf{S}^{w+1})$ we get $W^i(\mathsf{S}^{w+1}) \rightarrow R(\mathsf{S}^{w+1})$ and hence $i \leq \pi(R(\mathsf{S}^{w+1}))$. This shows $\pi(R(\mathsf{S}^w)) = \pi(R(\mathsf{S}^{w+1}))$. ◁

$S^0 \ldots S^{l-1}$: stabilizing
$1W1R$ regular bit

operation *Write*($v : \{0, \ldots, l-1\}$)
1 $S^v := 1$;
2 **while** $v \neq 0$
3 **do** $v := v - 1$; $S^v := 0$;

operation *Read*() : $\{0, \ldots, l-1\}$
$w : \{0, \ldots, l\}$;
4 $w := 0$;
5 **while** $S^w = 0 \wedge w < l$
6 **do** $w := w + 1$;
7 **if** $w = l$ **then return** $l - 1$;
8 **else return** w ;

Protocol 6.2 *A stabilizing 1W1R l-ary regular register.*

Theorem 6.14 *Protocol 6.2 implements a stabilizing* $1W1R$ *l-ary regular register using l stabilizing* $1W1R$ *regular binary registers.*

Proof According to Def. 6.6 we have to show that in every run $\langle \mathcal{A}, \rightarrow \rangle$ over the register all reads not overlapping a pending write return the value written by a feasible write. Clearly reads that do overlap a pending write behave arbitrary but do not behave as writes. Protocol 6.2 is trivially wait-free.

First consider a read R with $\mathrm{val}(R(S^v)) = 1$ for some v, so $\mathrm{val}(R) = v$. Let $\pi(R(S^v)) = i$. Then $W^i \not\leftarrow R$ and $\mathrm{val}(W^i) = v$ — because W^i wrote 1 to S^v if $i > 0$; if $i = 0$ then by Claim 6.13 $\pi(R(S^u)) = 0$ for all $u < v$ and by Prot. 6.2 all this reads must return 0, hence $\mathrm{val}(W^0) = v$ too. W^i is not a feasible write for R only if there exists a $W^j(w)$ such that $W^i \rightarrow W^j(w) \rightarrow R$ (and so $i < j$). If $w \geq v$, then by Claim 6.13 $\pi(R(S^v)) \geq j > i$, and if $w < v$, then using Claim 6.13 inductively $i < j \leq \pi(R(S^w)) \leq \pi(R(S^v))$. This contradicts the assumption that $\pi(R(S^v)) = i$.

Now consider a read R where, for all v, $\mathrm{val}(R(S^v)) = 0$. Then for this read $\mathrm{val}(R) = l - 1$. Because all writes write 1, if anything, to S^{l-1}, and $\mathrm{val}(R(S^{l-1})) = 0$, we have $\pi(R(S^{l-1})) = 0$. Then, using Claim 6.13 inductively, $\pi(R(S^v)) = 0$ for all v, and so, for all v, the value of S^v just after W^0 equals 0. Hence, $\mathrm{val}(W^0) = l - 1$ by definition. By a similar argument as before, W^0 is feasible for R. ◁

6.4.3 A 1-stabilizing $nWnR$ l-ary atomic register

Protocol 6.3 presents the adaptation of the Vitányi-Awerbuch [VA86, AKK+88] multi-writer atomic register construction from $1W1R$ multi-valued regular registers, into a wait-free 1-stabilizing $nWnR$ l-ary atomic register using n^2

$S_{11} \ldots S_{nn}$: stabilizing $1W1R$ regular: $\mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
(with fields *tag*, *id*, and *val*)

operation $Write_i(v : \mathcal{V})$
$max : \mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
1 $max := \max_{1 \le j \le n} S_{ji}$;
for $j := 1$ **to** n
2 **do** $S_{ij} := \langle max.tag + 1, i, v \rangle$;

operation $Read_i() : \mathcal{V}$
$max : \mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$;
1 $max := \max_{1 \le j \le n} S_{ji}$;
for $j := 1$ **to** n
2 **do** $S_{ij} := max$;
3 **return** $max.val$;

Protocol 6.3 *A 1-stabilizing nWnR l-ary atomic register.*

wait-free stabilizing $1W1R$ ∞-ary regular registers. To make this construction self-stabilizing the values have become part of the labels used to determine the most recently written value. In the original construction this is not necessary because any value stored with a tag, processor id pair always corresponds to the value written by that processor during that invocation. The main difficulty — and difference — in the proof of correctness exists in showing that the first few actions of a processor behave in line with Def. 6.4. We prove correctness of the protocol below.

In the protocol, $\mathcal{V}$ is the domain of values read and written by the multi-writer register. The construction uses n^2 regular stabilizing regular registers S_{ij} written by processor i and read by processor j. These registers store a *label* consisting of an unbounded *tag*, a processor *id* with values in the domain $\{1, \ldots, n\}$, and a *value* in $\mathcal{V}$. Labels are lexicographically ordered by $\leq$.

The sequential specification of an atomic register simply states that a write updates the state to be the value written, whereas a read returns the state of the register. Let $\langle \mathcal{A}, \rightarrow \rangle$ be a run of the above protocol. In the remainder of the proof, $\langle \mathcal{A}, \rightarrow \rangle$ is the above run with actions A with $\mathsf{count}(A) = 0$ (thus $\mathsf{count}(A) \neq \odot$) removed. We will show that for this "sub-run" there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ such that $\Rightarrow$ is an extension of $\rightarrow$ and for all writes $W(v)$ with $\mathsf{count}(W(v)) > 1$ set the state of the register to v, and for all reads R with $\mathsf{count}(R) > 1$, R returns the current state of the register. As for actions A, B with $\mathsf{count}(A) = 0$ and $\mathsf{count}(B) \neq 0$ in the original run either $A \rightarrow B$ or $A \| B$, we can prepend all these A to $\langle \mathcal{A}, \Rightarrow \rangle$ such that the resulting sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ corresponds to the original run

$\langle \mathcal{A}, \rightarrow \rangle$ and satisfies Def. 6.4.

We are going to partition $\mathcal{A}$ into a set $\mathcal{R}$ of actions that behave as reads and a set $\mathcal{W}$ of actions that behave as writes. To this end, define

$$\begin{aligned} \mathcal{F} &= \{A \in \mathcal{A} \mid \mathsf{count}(A) = \odot \vee \mathsf{count}(A) = 1\}, \\ \mathcal{R}^- &= \{A \in \mathcal{A} \mid \mathsf{count}(A) > 1 \text{ and } A \text{ is a read}\}, \text{ and} \\ \mathcal{W}^- &= \{A \in \mathcal{A} \mid \mathsf{count}(A) > 1 \text{ and } A \text{ is a write}\}. \end{aligned}$$

Then $\mathcal{F}$ corresponds to the set of actions that, according to Def. 6.4, may behave arbitrary. We further subdivide $\mathcal{F}$ into actions $\mathcal{F}_W$ that seem to behave as a write and actions $\mathcal{F}_R$ that seem to behave as a read, making sure that no two apparent writes write the same label (because the remainder of the proof, especially the definition of the reading function π depends on this).

Define for $A \in \mathcal{A}$, $\mathsf{label}(A)$ as the label written by A, and for a set of actions $\mathcal{F}$, $\mathsf{label}(\mathcal{F}) = \{\mathsf{label}(F) \mid F \in \mathcal{F}\}$. Set $\Lambda = \mathsf{label}(\mathcal{F}) \setminus \mathsf{label}(\mathcal{W}^-)$, the set of labels not written by a real write in $\mathcal{W}^-$, and let $\mathcal{F}_W$ be an arbitrary subset of $\mathcal{F}$ such that

(F1) $\mathsf{label}(\mathcal{F}_W) = \Lambda$,
(F2) For all $A, B \in \mathcal{F}_W$, if $\mathsf{label}(A) = \mathsf{label}(B)$ then $A = B$, and
(F3) For all $A \in \mathcal{F}_W$ and $B \in \mathcal{F}$, if $\mathsf{label}(A) = \mathsf{label}(B)$ then $t_1(A) < t_1(B)$.

Now set $\mathcal{F}_R = \mathcal{F} \setminus \mathcal{F}_W$ and define $\mathcal{W} = \mathcal{W}^- \cup \mathcal{F}_W$ and $\mathcal{R} = \mathcal{R}^- \cup \mathcal{F}_R$.

Lemma 6.15 *If $A \rightarrow B$ then $\mathsf{label}(A) \leq \mathsf{label}(B)$. If $B \in \mathcal{W}^-$ this inequality is strict.*

Proof Let A be performed by processor i and B be performed by processor j. If $A \rightarrow B$, then the write to S_{ij} by A precedes the read of S_{ij} by B. Because we only consider actions with $\mathsf{count} \neq 0$, the write to S_{ij} is not pending, and by $A \rightarrow B$ the read of S_{ij} does not overlap a pending write. Then the write of A to S_{ij} or a later write by action C of i to S_{ij} is a feasible write to the read of S_{ij} of B—hence this read returns the value written to S_{ij} by processor i during action A or the later action C. Since processor i both reads and writes from S_{ii}, and $\mathsf{count}(A) \neq 0$, $\mathsf{label}(A) \leq \mathsf{label}(C)$. Therefore the read of S_{ij} by B returns a label greater than or equal to $\mathsf{label}(A)$. B picks the maximum of all labels read, so if B is a read, $\mathsf{label}(A) \leq \mathsf{label}(B)$ and if B is a write, then $\mathsf{label}(A) < \mathsf{label}(B)$. ◁

The next lemma basically shows that every read returns a value writ-

ten by some write. Uniqueness of this write is established in Lemma 6.17.

Lemma 6.16 *For all $R \in \mathcal{R}$ there exists a $W \in \mathcal{W}$ such that* $\mathsf{label}(W) = \mathsf{label}(R)$ *and $R \not\leftarrow W$.*

Proof Define $A \rightsquigarrow B$ iff $\mathsf{label}(A) = \mathsf{label}(B)$ and $B \not\leftarrow A$. Then $A \rightsquigarrow A$. Let $R \in \mathcal{R}$ be arbitrary, and pick a $B \in \mathcal{A}$ such that $B \rightsquigarrow R$ and for no $A \in \mathcal{A}$, $A \neq B$, $A \rightsquigarrow B$. If $B \in \mathcal{W}$ we are done, so assume $B \in \mathcal{R}$.

Suppose $\mathsf{count}(B) > 1$. Then there is an operation C on the same processor with $\mathsf{count}(C) = 1$ and $C \rightarrow B$. If $\mathsf{label}(C) = \mathsf{label}(B)$ then $C \rightsquigarrow B$, while if $\mathsf{label}(C) < \mathsf{label}(B)$ (the only other possible case according to Lemma 6.15) then the contents of the register from which B obtains $\mathsf{label}(B)$ — note that since B is a read it writes the maximal label it reads — has changed after C read that same register. This register then is written by an operation D with $\mathsf{label}(D) = \mathsf{label}(B)$ before B reads it. Then $D \not\leftarrow C$, which, as $\mathsf{count}(C) = 1$, implies $\mathsf{count}(D) \neq 0$ and hence $D \rightsquigarrow B$. This contradicts the assumption that there is no A such that $A \rightsquigarrow B$.

We conclude that $\mathsf{count}(B) \leq 1$ and hence $B \in \mathcal{F}$. This implies $\mathsf{label}(B) \in \mathsf{label}(\mathcal{F})$. So either there exists a $W \in \mathcal{W}^-$ such that $\mathsf{label}(W) = \mathsf{label}(B) = \mathsf{label}(R)$, or $\mathsf{label}(B) \in \Lambda$ and by $(F1)$ there exists a $W' \in \mathcal{F}_W$ with $\mathsf{label}(W') = \mathsf{label}(B) = \mathsf{label}(R)$. In the first case, by Lemma 6.15, $R \not\leftarrow W$ as required. In the second case, since $B \in \mathcal{F}$ we must have by $(F3)$, $t_l(W') < t_l(B)$. Then as $B \rightsquigarrow R$ implies $R \not\leftarrow B$, this in turn implies $R \not\leftarrow W'$. ◁

The next lemma shows that different write actions write different labels. Together with the previous lemma this establishes that the maximal label read by a read action uniquely determines the write action that wrote the value this read returns.

Lemma 6.17 *For all $W, W' \in \mathcal{W}$ if* $\mathsf{label}(W) = \mathsf{label}(W')$ *then $W = W'$.*

Proof There are three cases

$W, W' \in \mathcal{W}^-$: By Prot 6.3 W and W' must be executed by the same processor (or else their *id*-fields differ). If $\mathsf{label}(W) = \mathsf{label}(W')$, and using Lemma 6.15, neither $W \rightarrow W'$ nor $W' \rightarrow W$. Hence $W = W'$.

$W \in \mathcal{W}^-, W' \in \mathcal{F}_W$: Then $\mathsf{label}(W) \notin \Lambda$, and $\mathsf{label}(W') \in \Lambda$ by $(F1)$. This is a contradiction.

$W, W' \in \mathcal{F}_W$: If $\mathsf{label}(W) = \mathsf{label}(W')$, then by $(F2)$ we have $W = W'$. ◁

Define for a particular run $\langle \mathcal{A}, \rightarrow \rangle$ a *reading function* $\pi : \mathcal{R} \mapsto \mathcal{W}$ by $\pi(R) = W$ if $\mathsf{label}(R) = \mathsf{label}(W)$ and $W \in \mathcal{W}$. This is a proper definition by the next lemma.

Lemma 6.18 *For all $R \in \mathcal{R}$, $\pi(R)$ is defined and unique, $R \not\rightarrow \pi(R)$, and R returns the value written by $\pi(R)$.*

Proof That $\pi(R)$ is defined and $R \not\rightarrow \pi(R)$ follows from Lemma 6.16. That it is unique follows from Lemma 6.17. If $\pi(R) \in \mathcal{W}^-$, then (stretching notation somewhat) $\mathsf{label}(\pi(R)).val$ equals the value written by $\pi(R)$. If $\pi(R) \in \mathcal{F}_W$ we define the (arbitrary) value written by $\pi(R)$ to equal $\mathsf{label}(\pi(R)).val$ ◁

We now show that every run $\langle \mathcal{A}, \rightarrow \rangle$ with the above reading function π is atomic. Define for $W \in \mathcal{W}$ its *clan* $[W]$ by

$$[W] = \{W\} \cup \{R \in \mathcal{R} \mid \pi(R) = W\} ,$$

and let $\Gamma = \{[W] \mid W \in \mathcal{W}\}$ be the set of all clans. Define $\rightarrow'$ over Γ by

$$[W] \rightarrow' [W'] \iff (\exists A \in [W], B \in [W'] :: A \rightarrow B) \quad .$$

Lemma 6.19 *For all $W \in \mathcal{W}$ and $A, B \in [W]$ we have $\mathsf{label}(A) = \mathsf{label}(B)$. Also if $W \neq W'$, then for all $A \in [W], B \in [W']$ we have $\mathsf{label}(A) \neq \mathsf{label}(B)$.*

Proof The first part follows from the definition of $[W]$ and $\pi(R)$. The second part follows from Lemma 6.17. ◁

Lemma 6.20 *$\rightarrow'$ is an acyclic partial order over Γ.*

Proof Suppose not. Then there exists a chain

$$[W_1] \rightarrow' [W_2] \rightarrow' \cdots \rightarrow' [W_m] \rightarrow' [W_1]$$

with $m > 1$, and $W_i \neq W_j$ if $i \neq j$. This implies that for all i with $1 \leq i \leq m$ there exist actions $A_i, B_i \in [W_i]$ such that $A_i \rightarrow B_{i+1}$ (where $m + 1 = 1$). By Lemma 6.15 and 6.19 $\mathsf{label}(A_i) \leq \mathsf{label}(B_{i+1}) = \mathsf{label}(A_{i+1})$. We conclude that $\mathsf{label}(A_1) = \mathsf{label}(A_2)$, contrary to Lemma 6.19. ◁

Applying these lemmas and the results of [AKK+88] we arrive at the following theorem.

Theorem 6.21 *Protocol 6.3 implements a* 1*-stabilizing* $nWnR$ *l-ary atomic register using* n^2 *stabilizing* $1W1R$ ∞*-ary regular registers.*

Proof Define a total order $\Rightarrow$ over $\mathcal{A}$ extending $\rightarrow$ as follows. First extend $\rightarrow'$ over Γ to a total order $\Rightarrow'$ (according to Lemma 6.20, this is possible). Now for $A \in [W]$ and $B \in [W']$ let $A \Rightarrow B$ if $[W] \Rightarrow' [W']$ (a). This extends $\rightarrow$ because if $A \rightarrow B$, then by the definition of $\rightarrow'$, $[W] \rightarrow' [W']$ and thus $[W] \Rightarrow' [W']$. For $A, B \in [W]$ fix an arbitrary extension $\Rightarrow$ of $\rightarrow$ such that for the only writer $W \in [W]$ we have for all other $C \in [W]$ that $W \Rightarrow C$ (b). This is an extension of $\rightarrow$ because by Lemma 6.18, $C \not\rightarrow W$.

Now $\Rightarrow$ is a total order over all actions $\mathcal{A}$ such that for all $R \in \mathcal{R}$ we have $\pi(R) \Rightarrow R$ by Lemma 6.18 and (b). Moreover, there does not exists a $W \in \mathcal{W}$ such that $\pi(R) \Rightarrow W \Rightarrow R$, because by (a) and the fact that $R \notin [W]$ by Lemma 6.19, either $W \Rightarrow [\pi(R)]$ or $R \Rightarrow [W]$. Hence $W \Rightarrow \pi(R)$ or $R \Rightarrow W$.

We conclude that all reads R return the value written by the most recently preceding write in the sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$. ◁

6.5 Further research

Our results are a first, but important, step towards exploring the relation between self-stabilization and wait-freedom in the construction of shared objects. This is a new area of research and there are still a lot of interesting questions in this new area that remain unanswered. First of all, this chapter is a proposal for a reasonable and general definition of self-stabilizing wait-free shared objects. The impossibility proofs and the constructions presented here are evidence for the viability of our approach, They show that our definitions are neither trivial nor impractical, but further research is necessary to assess their true value. Second, the construction of the 1-stabilizing $nWnR$ atomic register uses unbounded time-stamps to invalidate old values. We would like to know whether this is necessarily so, or if the space requirements of a k-stabilizing atomic register can be bounded. Finally, following the work of Aspnes and Herlihy [AH90], it is an interesting venture to classify, based on their sequential specification, all k-stabilizing shared memory objects that can be constructed from k'-stabilizing atomic registers, and to provide a general method to do so.

If it breaks
You get to keep both pieces.

CHAT(8)
Manual Page

7 Optimal Resiliency Against Mobile Faults

Abstract In this chapter we consider a model where malicious agents can corrupt hosts and move around in a network of processors. We consider a family of mobile-fault models $\mathsf{MF}(\frac{t}{n-1}, \rho)$. In $\mathsf{MF}(\frac{t}{n-1}, \rho)$ there are a total of n processors, the maximum number of mobile faults is t, and their *roaming pace* is ρ (for example, $\rho = 3$ means that it takes an agent at least 3 rounds to "hop" to the next host). We study in these models the classical testbed problem for fault-tolerant distributed computing: Byzantine agreement.

It has been shown that if $\rho = 1$, then agreement cannot be reached in the presence of even one fault, unless one of the processors remains uncorrupted for a certain amount of time. Subject to this proviso, we present a protocol for $\mathsf{MF}(\frac{1}{3}, 1)$, which is optimal. The running time of the protocol is $O(n)$ rounds, also optimal for these models.

7.1 Introduction

We consider a model where malicious agents can corrupt hosts and move around in a network of processors. The importance of this issue is rapidly growing with the current trend of universalization of computer networking and open systems, and, specially, with the advent of the so-called "agent technology." Previous works have, in one way or another, addressed the issue of dynamic faults. The following is a review of these results. Perhaps the first article to consider a model with malicious faults covering different fractions of the network at different points in time is due to Reischuk [Rei85]. The author designed a (sub-optimal) Byzantine agreement protocol able to toler-

ate them as long as they remain stationary for a given interval of time. More recently, Ostrovsky and Yung [OY91] proposed randomized methods to withstand the attack of mobile viruses. Canetti and Herzberg [CH94] use cryptographic techniques to achieve secure computation in a system where different sets of servers may be broken into at different times, but not all at once. In [FGY93], Franklin and Yung address the issue of privacy in a system with mobile eavesdroppers. Finally, Garay [Gar94] presents agreement protocols that improve on the work of Reischuk [Rei85].

However, one common characteristic in all the above results is that it is assumed that the faults (viruses, agents) are able to change position from one host to the next independently of when messages are sent in the network. In contrast, in this chapter we make the more natural assumption that faults are able to "travel" from one processor to another *only when messages are sent*. This is in indeed the case with network viruses (see, for example, [KW91]).[1]

For this model, we evaluate the classical testbed problem for fault-tolerant distributed computing: Byzantine agreement (*BA*) [LSP82]. Loosely stated, the problem requires processors in a communication network, some of which may be faulty, and all of which start the computation with an initial value, to decide on the same value. Moreover, it is required that if the initial value among the nonfaulty processors is the same, then that must be the decision value. In fact, a distinct characteristic of the mobile fault environments is that once a global state is reached, then special efforts must be dedicated in order to maintain it. This leads us to consider a variant of the problem (defined in Sect. 7.2) that is more adequate for these environments.

In this chapter we review impossibility results for this model, and present a protocol that requires the optimal number of processors to achieve (and maintain) consistency in a network, in the presence of the "fastest" mobile faults. (Indeed, as pointed out in [Gar94], the speed with which the faults can move is a fundamental parameter in this model.)

Finally, we emphasize that our model and work necessarily represent an abstraction of things, in terms of the faults' power of disruption and coordination, as well as of the "clean" synchronous structure and "integral" fault speed assumption we derive our bounds on. The former assumptions we find necessary for the analysis of scenarios where things can really go

[1] However, one important distinction is that we will assume that the total number of agents in the network is upper-bounded by a parameter t, as opposed to "combinatorial explosion"-type situations.

wrong (e.g., life-critical applications; cryptographic settings; etc.). The latter simplifications we consider important for the understanding of the basic intricacies and issues involved in faulty-agent mobility; once the basic facts are established, similar results can hopefully be extended and derived with lesser efforts for variants and/or weaker versions of the model.

The rest of the chapter is organized as follows. In Sect. 7.2 we define more formally the *Mobile Fault* model MF, as well as *MBA*, the version of the agreement problem that is adequate for these new environments. In Sect. 7.3 we present the negative results, while in Sect. 7.4 we present the protocol that tolerates the optimal number of fast-moving faults.

7.2 Mobile fault model

We are given a network of n processors numbered 1 through n that may communicate only by exchanging messages via a reliable point-to-point channel. In order to be able to measure and quantify more cleanly the issue of mobility, we shall consider the standard model (see, for example, [LSP82]) of synchronous computation. In such a model, the network computation evolves as a series of *rounds*, during which processors can send one message to all other processors, receive the messages, and perform some local computation.

We assume that in the network there are (malicious) agents that can "corrupt" the processors. The semantics of corruption is as follows: In order to understand the worst possible disruption that can be caused in such a setting, we assume that the malicious agents are able to take complete control over the application being run, and even wipe out their memories. Furthermore, the agents are *mobile*, meaning that they can leave their current "hosts," and move on to "infect" new, different processors. Specifically, given the computation's round structure, we make the natural assumption that the agents will "travel" or move with the "Send" or "Broadcast" operations. We will also assume that the total number of agents in the network is bounded by a parameter t.

A consequence of the assumptions above is that, in the worst case, processors which are left by the agents and come "back to life" are unable to contribute in any meaningful way to the on-going computation, and will require the help of the currently correct processors in order to reconstruct the state of the execution (including the code of the protocol being executed!). This model is very powerful in terms of the agents' disruptive capabilities but, as mentioned before, we are interested in understanding what might be required in worst-case scenarios (e.g., life-critical applications, security settings,

etc.). These assumptions can be weakened in several ways. For example, one could assume that some form of secure, tamper-proof memory is available.

We shall refer to these agents as *faults*, or simply *agents*. Given an assignment of the faults to the processors at any given round, we will draw from the virus terminology the term *cured* to refer to the processors that were occupied by the agent in the previous round, but no longer. We will refer to the remaining processors not possessed by the agents as *nonfaulty*, or *noninfected*.

We use the formalism of [Gar94] to describe mobile fault environments. A fundamental parameter in such an environment is the *roaming pace* of the faults ρ, $[\rho] = \frac{\#\ rounds}{\#\ nodes}$, with which the agents can travel in the network. For convenience, we will normalize to the number of rounds that it takes to move from one processor/node to another. Thus, $\rho = 3$ indicates that it takes an agent 3 rounds to "hop" to the next nonfaulty processor. Further, for simplicity, we only consider the case of $\rho \in \mathbf{N}$, the set of natural numbers. We will sometimes refer to the case of $\rho = 1$ as "full speed," the case of $\rho = 2$ as "half speed," etc.

We can now more formally define the *Mobile Fault* model as a family MF of models indexed by the tuple $(\frac{t}{n-1}, \rho)$. In $\mathsf{MF}(\frac{t}{n-1}, \rho)$ there are a total of n processors, the maximum number of mobile faults is t, and their roaming pace is ρ. (Granted that at first sight, this notation is not the prettiest. However, it does allow us to express in a handy manner that, for example, the number of faults is less than a third — of the size of the network — and that they move at half speed.) Notice that, in contrast with the traditional models of static faults (for example, [PSL80]) in which the only restriction is the bound on the maximum number of faults, there is no guarantee in the mobile-fault model that a processor that is initially correct will remain uninfected forever.

Intuitively, in this type of environment, as opposed to the traditional environment of faults that are static, things have more of a continuous, longer execution flavour, since one cannot expect the (currently) noninfected processors to reach meaningful conclusions after a few rounds of communication. This is particularly illustrated by the classical problem of achieving consistency in such an environment. Recall the *Byzantine Agreement* problem [LSP82], which can be defined as follows. At the offset of the computation, each processor p holds an initial value $v_p \in \{0, 1\}$. Regardless of the behaviour of the faulty processors, required is a distributed protocol satisfying the following conditions:

Decision Every nonfaulty processor eventually irreversibly decides on a value $d_p \in \{0, 1\}$.

Agreement The nonfaulty processors decide on the same value.

Validity If the initial values v_p of all nonfaulty processors are identical, then $d_p = v$ for all nonfaulty processors p.

However, notice that in MF, if there is a point in time when the three conditions are satisfied, the fact that the faults can continue to move along will make this global condition disappear. Thus, in this chapter we consider *Mobile-Fault* Byzantine Agreement (*MBA*), a version of the problem that was presented in [Gar94] that is adequate for the mobile-fault environment. *MBA* is defined similarly to the *BA* problem above, together with the additional condition:

Consistency Maintenance Once Agreement is achieved among the currently noninfected processors, it is preserved among the (possibly different) noninfected processors.

This condition captures the contrast of MF with respect to the traditional models, in that if there is a moment (interval) in time in which a consistency state is achieved among the currently noninfected processors, then special efforts will have to be dedicated to *maintaining* such a state, by having the processors that participated in the decision "pass the token" to the processors that re-join the computation after the agents have left them.

Albeit being more stringent, Consistency Maintenance resembles the correctness condition of *self-stabilizing* protocols (e.g., [Dij74]). In our case, however, it is required that the correctness (consistency) state be maintained once it is reached, even if faults continue to occur. We will sometimes abuse the language and say that *MBA* is "achieved," meaning that consistency is reached, and from that point on, maintained.

7.3 Impossibility results

It is well known that in the case of t static (arbitrary) faults, *BA* is achievable whenever the size of the network $n > 3t$ [LSP82]. It turns out that in the case of faults that may move, there is a tight relationship between their speed (pace) and the number of faults that can be tolerated. The following, somewhat curious result from [Gar94], also applies to our model where the agents

can only move with the messages:

Proposition 7.1 [Gar94] *In* $\mathtt{MF}(\cdot, 1)$, BA *can only be achieved if* $t = 0$.

The proof of this proposition is an extension of the lower bound of Dolev and Strong [DS82], which shows that $t + 1$ rounds of communication are needed in the classical setting of static faults with a maximum of t faults. The basic idea is that by having a single mobile agent "hopping" from one processor to the next, one can construct executions that exhibit *serial faultiness*, in the sense that at each round a new processor misbehaves. Assuming now that *BA* can be achieved in k (not necessarily $< n$) rounds, techniques similar to those of [DS82] can be used to show that all executions are equivalent — in the sense of the currently noninfected processors deciding on the same value — to that in which the source does not send any value, a contradiction.

The above result is fairly extreme: Can't even tolerate one fault if it moves at full speed! One may wonder if the situation gets any better by slowing the faults down. This is what the following proposition — a generalization of Prop. 7.1 — characterizes:

Proposition 7.2 [Gar94] *In* $\mathtt{MF}(\cdot, \rho)$, BA *is possible only if* $t < \rho$.

Roughly speaking, the proposition is proved by the t agents implementing serial faultiness through coordination. Note that one can now view the case of $\rho \geq t + 1$ as corresponding to the classical setting of static faults, in the sense that if the faults remain immobile for at least that long, then protocols exist that are guaranteed to yield agreement in that much time. Given Prop. 7.1 and 7.2, the following condition constitutes one way of overcoming the above difficulties:

(*1*) At least one processor remains uncorrupted.

We call the reader's attention to the diversity of contexts in which this condition reveals itself as necessary (e.g., failure detection [CT91], cryptographic protocols [CH94], etc.). Compare this also with the "clean round" concept of [DM90].

In practice, condition *1* captures situations where not all places in the network are equally accessible to the faults. We also note that if the identity of the processor that cannot be corrupted by the faults is known beforehand

to all processors, then the problem we are trying to solve becomes trivial. We will thus assume that this is not the case. The condition does not specify the length of the period during which the processor cannot be corrupted. In the next section we present a protocol that requires a processor to remain uncorrupted for $O(n)$ rounds of communication.

Another consequence of the *1* condition is that a direct application of the so-called full information protocols (e.g., [BNDD+87, LSP82]) where all correct processors send their views to all other processors, and this for $t + 1$ rounds, will not work here. An essential requirement for these protocols to work is that of the "pigeon-hole" principle: A time is needed where a "clean round" (i.e., no new faults) occurs [DM90], and this cannot be guaranteed given the mobility of the faults. Indeed, as pointed out in [Gar94], in contrast with the classical setting of static faults, solutions to *MBA* necessitate a time proportional to n, the size of the system, as opposed to t. Specifically:

Corollary 7.3 *Every* MBA *protocol requires n rounds of communication in its worst-case run.*

Finally, the lower bound on the number of processors, as a function of the number of faults ($n > 3t$ [PSL80]), also holds for MF systems satisfying *1*. In the next section, we present a protocol for $\mathsf{MF}(\frac{1}{3}, 1)$ (i.e., optimal number of processors and faults at full speed) that relies on *1* to achieve *MBA* in $3n$ rounds.

7.4 Agreement protocols for mobile faults

Given condition *1* and the observation from the previous section on the failure in MF of the classical full information protocols, a natural type of solution to look at is the *Phase King* paradigm of Berman and Garay [BG89], since its correctness relies on the eventual existence of a good processor (the "King"). The execution of a protocol adhering to this paradigm is divided into *phases*, each with a different king. In a phase, the following steps take place:

1. Round(s) of exchange of messages and computation among all processors; and
2. all processors listen to the King.

The purpose of 1 is to eliminate possible discrepancies in the configuration of existing values, and discover the unique value, if such exists. In 2, processors

that are not "overwhelmingly convinced" of the existence of a unique value, trust the king of the phase and adopt his value. The reader is referred to [BG89] for further details.

However, a direct use of the existing Phase King protocols for the static-fault models is not possible, since even if unanimity is achieved by the currently correct processors at a given round, that state may disappear as the agents move along. This brings us to the concept of *reconstruction* of information, the subject of the next section.

7.4.1 Network memory

The basic idea is to the use of the network as a collective memory device. Cryptographic settings — where not all parties can be trusted — are the natural places where these techniques originated (e.g., [Sha79]). More recently, the concept of network memory has been utilized by Ostrovsky and Yung [OY91] and Garay [Gar94]. The concept can be briefly formulated as follows. Let a network *object* $\mathcal{O}$ be defined by the tuple $<$*round number*, *processor id*, *data*$>$. Then, given a big enough fraction of noninfected processors at all times, it is possible for a correct processor to store $\mathcal{O}$ in the network by sending copies to all processors, maintain it for a period of time (say, k rounds), and reconstruct it whenever needed. In this section we will be interested in storing objects for just one round. Specifically, we establish the following simple fact for $\mathsf{MF}(\frac{1}{3}, 1)$:

Lemma 7.4 *Let p be a noninfected processor at round r. Then in $\mathsf{MF}(\frac{1}{3}, 1)$, every correct processor (cured or noninfected) can reconstruct at $r+1$ any object stored by p at r.*

Proof In round r, p is correct, and stores m in the network by sending a copy of m to all processors. This message is received in round r by at least $2t+1$ correct processors: at least $t+1$ that remained correct since last round, plus at least t more, either because they remained correct, or because they became cured. In round $r+1$, all of these processors echo $\mathcal{O}$. The new object becomes:

$$\mathcal{O} = \begin{cases} < r, p, m > & \text{if received } m \text{ from at least} \\ & 2t+1 \text{ processors;} \\ < r, p, \perp > & \text{otherwise.} \end{cases}$$

Since in round $r + 1$ at most t of the messages come from processors that were infected in round r, all the correct processors at round $r + 1$ receive m with multiplicity at least $2t + 1$, and are thus able to reconstruct m. ◁

Note that there is no guarantee that objects stored by infected processors will be univocally reconstructed, but this will be of no consequence to us. Lemma 7.4 can be generalized to reconstruct objects stored, say, k rounds back. Notice though that in MF, per the argument above, it is imperative that the copies of the object be continuously refreshed.

7.4.2 A protocol with optimal resiliency

We are now ready to introduce MOPT, a protocol that is able to cope with mobile faults, for model MF($\frac{1}{3}$, 1). The protocol is shown in Prot. 7.1. It consists of *phases*, each consisting of three rounds of message exchange. There are three values that get transmitted at any given time: $\perp$ (for "undecided"), 0, and 1; we assume that $\perp < 0$. In "universal exchange 1," all the currently correct processors send their values V to all processors, and store the received values in vector MV. (We assume that the cured processors have the ability to receive messages. These would also include the protocol itself and the global state of the execution — e.g., round number — which we omit for clarity.) Each processor (both noninfected and cured) then checks if there exists a value $\in \{0, 1\}$ that is very popular among the received values. If not, the processor stays undecided. The structure of "universal exchange 2" is similar to that of the first exchange, except that its purpose is the discovery of a unique value, if such exists. Again, it could be that in this exchange up to t agents decide to move to new hosts. The cured processors get up to speed by computing the values of $D[\cdot]$ based on the received MV's. In the "reconstruction and king's broadcast" exchange, processors re-send their MV's from the previous round, which they store in table *ECHO*. Processors that became cured in this round use that introspective ability to execute procedure RECONSTRUCT, shown in Prot. 7.2. Finally, if a correct (cured or noninfected) is either still undecided, or not overwhelmingly convinced about the existence of a possibly unique value, it "listens" to processor k, the king of the phase. Since all processors are supposed to send their "view" (i.e., MV) from the last round, every processor is able to reproduce the king's computation (i.e., V_k). Note that in MOPT the reconstruction procedure is only used in the last exchange.

We now address the correctness of the protocol. We first define the following potential function:

protocol MOPT;

$V := v_p$;

for $r := 0$ **to** ∞ **do**

(* universal exchange 1 *)

broadcast(V);

receive($MV[\cdot]$);

for $j := 0$ **to** 1 **do** $C[j] :=$ # of j's in MV;

$$V := \begin{cases} 0 & \text{if } C[0] \geq n - t \\ 1 & \text{if } C[1] \geq n - t \\ \perp & \text{otherwise} \end{cases};$$

(* universal exchange 2 *)

broadcast(V);

receive($MV[\cdot]$);

for $j :=\perp$ **to** 1 **do** $D[j] :=$ # of j's in MV;

$$V := \begin{cases} 0 & \text{if } D[0] > t \\ 1 & \text{if } D[1] > t \\ \perp & \text{otherwise} \end{cases};$$

(* reconstruction and king's broadcast *)

broadcast($MV[\cdot]$);

forall i

do $ECHO[i, \cdot] := MV[\cdot]$ received from i;

if *cured*

then RECONSTRUCT;

$k := (r \bmod n) + 1$; (* k is the phase's king *)

if $(V =\perp) \vee (D[V] < n - t)$

then $V := \max(0, V_k)$; (* $V_k = MV[k]$ is value received from k *)

od

Protocol 7.1 *A protocol for* MF($\frac{1}{3}$, 1)*; code for processor* p.

$\Phi_v(r) \triangleq$ the number of correct processors p for which, at the end of phase r, $V_p = v$.

Observe that the Consistency Maintenance condition of *MBA* enforces on functions Φ the property that whenever they reach the "ceiling" of $n - t$, they should irrevocably remain there. The next lemma shows that this is indeed the case for MOPT. For simplicity we assume $n = 3t + 1$.

procedure RECONSTRUCT;
    forall i
    do if $\exists\ v \in \{\perp, 0, 1\}$ s.t. $ECHO[\cdot, i] = v$ at least $n - t$ times
        then $MV[i] = v$
        else $MV[i] = \perp$;
for $j := \perp$ **to** 1 **do** $D[j] :=$ # of j's in MV;

$$V := \begin{cases} 0 & \text{if } D[0] > t \\ 1 & \text{if } D[1] > t \\ \perp & \text{otherwise} \end{cases} ;$$

Protocol 7.2 *Procedure* RECONSTRUCT *for* MF$(\frac{1}{3}, 1)$.

Lemma 7.5 (Consistency Maintenance) *In* MF$(\frac{1}{3}, 1)$, *if in phase* $f \geq 0$ *of* MOPT $\Phi_v(f) \geq 2t + 1$ *holds for some* $v \in \{0, 1\}$, *then* $\Phi_v(r) \geq 2t + 1$ *holds at all phases* $r > f$.

Proof Assume $\Phi_v(f) \geq 2t + 1$ holds for some $v \in \{0, 1\}$. In universal exchange 1 of phase $f + 1$ at least $2t + 1$ v's get sent, and since for each fault that moves there is a cured processor, each correct (cured or noninfected) processor has $C[v] = 2t + 1 \geq n - t$ and $C[\overline{v}] \leq t < n - t$. Thus, all of these processors assign v to V.

In universal exchange 2, at least $2t + 1$ processors send v, and again for at least $2t + 1$ cured and noninfected processors $D[v] \geq n - t$ and also $D[w] \leq t$, for $w \in \{\perp, \overline{v}\}$, assigning v to V.

During the reconstruction and king's broadcast, of the $2t + 1$ correct processors from the previous round, at least $t + 1$ processors remain uninfected, and therefore ignore the king's broadcast. Of the remaining t correct processors from the previous round, again for each new infected processor there is a cured processor. Each cured processor executes procedure RECONSTRUCT, and it follows from Lemma 7.4 that each cured processor computes $D[v] = 2t + 1 \geq n - t$, and $V = v$, thus also ignoring the king's broadcast. This yields at least $2t + 1$ correct processors $V = v$ at the end of phase $f + 1$, and thus $\Phi_v(f + 1) \geq 2t + 1$. Now the situation repeats itself, and we are done. ◁

Theorem 7.6 *In model* MF$(\frac{1}{3}, 1)$ *satisfying condition 1, protocol* MOPT *achieves*

Mobile Byzantine Agreement in $3n$ *rounds of communication.*

Proof We first consider Validity. The configuration is unanimous at the start of phase 0 with at least $2t + 1$ noninfected processors holding value $v \in \{0, 1\}$. Similar arguments to those of Lemma 7.5 allows us to establish that $\Phi_v(0) \geq 2t + 1$, and using Lemma 7.5 for the following phases yields the theorem.

We now address Agreement. Let I be the phase corresponding to the processor that cannot be corrupted by the faults. In particular, I will be sending the same MV in the "reconstruction and king's broadcast" exchange as it did in "universal exchange 2." Two cases are possible after the last exchange of phase I:

Case 1: $D_p[V_p] < n - t$ or $V_p = \perp$ for every correct (cured or noninfected) p. Then all processors execute the assignment to V. As I is correct by assumption, all correct processors receive the same MV from I, and compute the same V_I. If $V_I = \perp$, they all assign 0 to V.

Case 2: $D_p[V] \geq n - t$ and $V_p \neq \perp$ for at least some correct (cured or noninfected) p. Lemma 7.4 and the fact that there are no more than t corrupted processors guarantee that $|C_q[v] - C_r[v]| \leq t$ and $|D_q[v] - D_r[v]| \leq t$, for every non-faulty q, r and $v \in \{\perp, 0, 1\}$. Therefore, if a correct processor p computes $D_p[V_p] \geq n - t$ after the second universal exchange (or reconstructs and computes $D_p[V_p] \geq n - t$ during the reconstruction and king's broadcast), then $D_q[V_p] > t$ and $D_q[\overline{V_p}] \leq t$ for all other correct processors q, including the king I. As a result, each correct processor q has $V_q = V_I$ at the end of the phase, either because it ignores I's value, or because of accepting it.

In either case, all correct processors have the same value in V at the end of phase I, and Lemma 7.5 applies.

Regarding the round complexity, observe that I's turn becomes at phase n at the latest. ◁

7.5 Final remarks

In this chapter we have studied the problem of t malicious agents moving around in a network of n processors. In contrast with previous works, the agents in our model can only move when messages are sent in the network.

Subject to the condition that one of the processors remains uncorrupted for a certain amount of time, we have presented a protocol to reach and maintain agreement among the uncorrupted processors that tolerates the maximal number of bad agents (namely, $t < \frac{n}{3}$), in $3n$ communication rounds. Notice the gap between the running time of our protocol and that of Corollary 7.3. We also point out that if randomization is allowed, then the above condition can be dropped, and techniques similar to those of [FM88] can be used in a straightforward manner to reach agreement in $O(1)$ expected time, but only tolerating up to $t < \frac{n}{6}$ mobile agents. Can this be improved?

Acknowledgements

We thank Mark Moir for useful comments and suggestions.

Multas per gentes et multa per aequora vectus
advenio has miseras, frater, ad inferias ...

C. VALERIUS CATULLUS
Carmina, CI

8 Optimal Routing Tables

Abstract The optimal space used to represent routing schemes in communication networks is established, both for worst-case static networks and on the average for all static networks. Several factors may influence the cost of representing a routing scheme for a particular network. It is therefore unavoidable that we first describe several reasonable models in which to measure this cost. Failure to do so in the past has obfuscated previous results.

We show that, in most models, for almost all graphs $\Theta(n^2)$ bits are necessary and sufficient for shortest path routing. By 'almost all graphs' we mean the Kolmogorov random graphs which constitute a fraction of $1 - 1/n^c$ of all graphs on n nodes, where $c \geq 3$ is an arbitrary fixed constant. In contrast, there is a model that rises the average case lower bound to $\Omega(n^2 \log n)$ and another model where the average case upper bound drops to $O(n \log^2 n)$. This clearly exposes the sensitivity of such bounds to the model under consideration. Furthermore, if paths have to be short, but need not be shortest (i.e., if the stretch factor may be larger than 1), our other upper bounds indicate that much less space is needed on average, even in the more demanding models.

For worst-case static networks we prove a $\Omega(n^2 \log n)$ lower bound for shortest path routing, for those models where the nodes in the network are labelled $1, \ldots, n$. This lower bound holds even for all stretch factors < 2.

Throughout, we use the incompressibility method based on Kolmogorov complexity.

8.1 Introduction

A universal *routing strategy* for static communication networks will, for every network, generate a *routing scheme* for that particular network. Such a routing scheme comprises a *local routing function* for every node in this network. The routing function of node u returns for every destination $v \neq u$ an edge incident to u on a path from u to v. This way, a routing scheme describes a path, called a *route*, between every pair of nodes u, v in the network. The *stretch factor* of a routing scheme equals the maximum ratio between the length of a route it produces, and the shortest path between the endpoints of that route. The stretch factor of a routing strategy equals the maximal stretch factor attained by any of the routing schemes it generates. If the stretch factor of a routing strategy equals 1, it is called a *shortest path routing strategy* because then it generates for every graph a routing scheme that will route a message between arbitrary u and v over a shortest path between u and v.

In a *full information* shortest path routing scheme, the routing function in u must, for each destination v return all edges incident to u on shortest paths from u to v. These schemes allow alternative, shortest, paths to be taken whenever an outgoing link is down.

We consider point to point communication networks on n nodes described by an undirected graph G. The nodes of the graph initially have unique labels taken from $\{1, \ldots, n\}$. Edges incident to a node v with degree $d(v)$ are connected to *ports*, with fixed labels $1, \ldots, d(v)$, by a so called *port assignment*. This coincides with the minimal local knowledge a node needs to route: a) a unique identity to determine whether it is the destination of an incoming message, b) the guarantee that each of its neighbours can be reached over a link connected to exactly one of its ports, and c) that it can distinguish these ports.

The space requirements of a routing scheme is measured as the sum over all nodes of the number of bits needed on each node to encode its routing function. If the nodes are not labelled with $\{1, \ldots, n\}$—the minimal set of labels—we have to add to the space requirement, for each node, the number of bits needed to encode its label. Otherwise, the bits needed to represent the routing function could be appended to the original identity yielding a large label that is not charged for but does contain all necessary information to route.

The cost of representing a routing function at a particular node depends on the amount of (uncharged) information initially there. Moreover, if we are allowed to relabel the graph and change its port assignment before

generating a routing scheme for it, the resulting routing functions may be simpler and easier to encode. On a chain, for example, the routing function is much less complicated if we can relabel the graph and number the nodes in increasing order along the chain. We list these assumptions below, and argue that each of them is reasonable for certain systems. We start with the three options for the amount of information initially available at a node.

I Nodes do not initially know the labels of their neighbours, and use ports to distinguish the incident edges. This models the basic system without prior knowledge.

  IA The assignment of ports to edges is fixed and cannot be altered. This assumption is reasonable for systems running several jobs where the optimal port assignment for routing may actually be bad for those other jobs.

  IB The assignment of ports to edges is free and can be altered before computing the routing scheme (as long as neighbouring nodes remain neighbours after re-assignment). Port reassignment is justifiable as a local action that usually can be performed without informing other nodes.

II Nodes know the labels of their neighbours, and know over which edge to reach them. This information is for free. Or, to put it another way, an incident edge carries the same label as the node it connects to. This model is concerned only with the additional cost of routing messages beyond the immediate neighbours, and applies to systems where the neighbours are already known for various other reasons[1].

Orthogonal to that, the following three options regarding the labels of the nodes are distinguished.

α Nodes cannot be relabelled. For large scale distributed systems relabelling requires global coordination that may be undesirable or sim-

[1] We do not consider models that give neighbours for free and, at the same time, allow free port assignment. For, given a labelling of the edges by the nodes they connect to, the actual port assignment doesn't matter at all, and can in fact be used to represent $d(v) \log d(v)$ bits of the routing function. Namely, each assignment of ports corresponds to a permutation of the ranks of the neighbours — the neighbours at port i moves to position i. There are $d(v)!$ such permutations.

ply impossible.

β Nodes may be relabelled before computing the routing scheme, but the range of the labels must remain $1, \ldots, n$. This model allows a bad distributions of labels to be avoided.

γ Nodes may be given arbitrary labels before computing the routing scheme, but the number of bits used to store its label are added to the space requirements of a node. Destinations are given using the new, complex, labels[2]. This model allows us to store additional routing information, e.g. topological information, in the label of a node. This option is justified for centrally designed interconnect networks for multiprocessors and communication networks.

These two orthogonal sets of assumptions IA, IB, or II, and α, β, or γ, define the nine different models we will consider in this chapter.

8.1.1 Summary of our results

We determine the optimum space used to represent shortest path routing schemes on almost all graphs, namely the Kolmogorov random graphs which constitute a fraction of at least $1 - 1/n^3$ of all graphs. These bounds straightforwardly imply the same bounds for the average case over all graphs. For an overview of the results, refer to Table 8.1[3].

We prove that for almost all graphs $\Omega(n^2)$ bits are necessary to represent the routing scheme, if relabelling is not allowed and nodes know their neighbours (II $\wedge$ α) or nodes do not know their neighbours (IA $\vee$ IB)[4]. Partially matching this lower bound, we show that $O(n^2)$ bits are sufficient to represent the routing scheme, if the port assignment may be changed or if nodes do know their neighbours (IB $\vee$ II). In contrast, for almost all graphs, the lower

[2] In this model it is assumed that a routing function cannot tell valid from invalid labels, and that a routing function always receives a valid destination label as input. Requiring otherwise makes the problem harder.

[3] In this table, arrows indicate that the bound for that particular model follows from the bound found by tracing the arrow. In particular, the average case lower bound for model IA $\wedge$ β is the same as the IB $\wedge$ γ bound found by tracing $\rightarrow$ and $\downarrow$. The reader may have guessed that a ? marks an open question.

[4] We write A $\vee$ B to indicate that the results hold under model A or model B. Similarly, we write A $\wedge$ B to indicate the result holds only if the conditions of both model A and model B hold simultaneously. If only one of the two 'dimensions' is mentioned, the other may be taken arbitrary (i.e., IA is a shorthand for (IA $\wedge$ α) $\vee$ (IA $\wedge$ β) $\vee$ (IA $\wedge$ γ)).

	no relabelling (α)	*permutation* (β)	*free relabelling* (γ)
worst case — *lower bounds*			
port assignment free (IB)	$\rightarrow$	$\Omega(n^2 \log n)$ [GP96]	?
neighbours known (II)	$\Omega(n^2 \log n)$	$\Omega(n^2)$ [FG95]	$\Omega(n^{7/6})$ [PU89]
average case — *upper bounds*			
port assignment fixed (IA)	$O(n^2 \log n)$	$\leftarrow$	$\leftarrow$
port assignment free (IB)	$O(n^2)$	$\leftarrow$	$\leftarrow$
neighbours known (II)	$O(n^2)$	$\leftarrow$	$O(n \log^2 n)$
average case — *lower bounds*			
port assignment fixed (IA)	$\Omega(n^2 \log n)$	$\rightarrow$	$\downarrow$
port assignment free (IB)	$\downarrow$	$\rightarrow$	$\Omega(n^2)$
neighbours known (II)	$\Omega(n^2)$	?	?

Table 8.1 *Size of shortest path routing schemes: overview of results.*

bound rises to $\Omega(n^2 \log n)$ bits if both relabelling and changing the port assignment is not allowed (IA $\wedge$ α). And, again for almost all graphs, the upper bound drops to $O(n \log^2 n)$ bits if nodes know the labels of their neighbours and nodes may be arbitrarily relabelled (II $\wedge$ γ).

Full information shortest path routing schemes are shown to require, on almost all graphs, $\Omega(n^3)$ bits to be stored, if relabelling is not allowed (α). This matches the obvious upper bound for all graphs. For stretch factors larger than 1 we obtain the following results. When nodes know their neighbours (II), for almost all graphs, routing schemes achieving stretch factors s with $1 < s < 2$ can be stored using a total of $O(n \log n)$ bits[5]. Similarly, for almost all graphs in the same models (II), $O(n \log \log n)$ bits are sufficient for routing with stretch factor ≥ 2. Finally, for stretch factors $\geq 6 \log n$ on almost all graphs again in the same model (II), the routing scheme occupies only $O(n)$ bits.

For worst case static networks we prove, by construction of explicit graphs, a $\Omega(n^2 \log n)$ lower bound on the total size of any routing scheme with stretch factor < 2, if nodes may not be relabelled (α). The techniques

[5] For Kolmogorov random graphs, which have diameter 2 by Lemma 8.6, routing schemes with $s = 1.5$ are the only ones possible in this range.

used throughout are incompressibility arguments based on Kolmogorov complexity, [LV93].

8.1.2 Comparison with related work

Previous upper- and lower bounds on the total number of bits necessary and sufficient to store the routing scheme in worst-case static communication networks are due to Peleg and Upfal [PU89], and Fraigniaud and Gavoille [FG95]. Our bounds are stronger, because they apply to the average case as well.

In [PU89] it was shown that for any stretch factor $s \geq 1$, the total number of bits required to store the routing scheme for some n-node graph is at least $\Omega(n^{1+1/(2s+4)})$ and that there exist routing schemes for all n-node graphs, with stretch factor $s = 12k + 3$, using $O(k^3 n^{1+1/k} \log n)$ bits in total. For example, with stretch factor $s = 15$ we have $k = 1$ and their method guarantees $O(n^2 \log n)$ bits to store the routing scheme. The lower bound is shown in the model where nodes may be arbitrarily relabelled and where nodes know their neighbours (II $\wedge$ γ). Free port assignment in conjunction with a model where the neighbours are known (II) can, however, not be allowed. Otherwise, each node would gain $n \log n$ bits to store the routing function in (see the footnote to model II).

Fraigniaud and Gavoille [FG95] showed that for stretch factors $s < 2$ there are routing schemes that require a total of $\Omega(n^2)$ bits to be stored in the worst case if nodes may be relabelled by permutation (β).

Kranakis *et al.* in [KKU95a, KKU95b, KK96] independently use Kolmogorov complexity to obtain results on interval routing, bounded degree graphs, full information routing, shortest path families, and an alternative proof for the result of Fraigniaud and Gavoille [FG95]. In particular, they show that for each n *there exist* graphs on n nodes (actually about a $1/2^{n/2}$th fraction of all such graphs), which may not be relabelled (α), that require $\Omega(n^3)$ bits to store a *full information* shortest path routing scheme.

Finally, Gavoille and Pérennès [GP96] recently showed that there are routing schemes that require a total of $\Omega(n^2 \log d)$ bits to be stored in the worst case for some graphs with maximal degree d, if nodes may be relabelled by permutation and the port assignment may be changed (IB $\wedge$ β).

To the best of our knowledge, Jan van Leeuwen was the first to formulate explicitly the question of what exactly is the optimal size of the routing functions, and he recently drew also our attention to this group of problems. See also [FLMS95].

8.2 Kolmogorov complexity

The Kolmogorov complexity, [Kol65], of x is the length of the *shortest* effective description of x. That is, the *Kolmogorov complexity* $C(x)$ of a finite string x is simply the length of the shortest program, say in FORTRAN (or in Turing machine codes) encoded in binary, which prints x without any input. A similar definition holds conditionally, in the sense that $C(x|y)$ is the length of the shortest binary program which computes x given y as input. It can be shown that the Kolmogorov complexity is absolute in the sense of being independent of the programming language, up to a fixed additional constant term which depends on the programming language but not on x. We now fix one canonical programming language once and for all as reference and thereby $C()$.

For the theory and applications, see [LV93]. Let $x, y, z \in \mathbb{N}$, where $\mathbb{N}$ denotes the natural numbers. Identify $\mathbb{N}$ and $\{0,1\}^*$ according to the correspondence $(0,\epsilon), (1,0), (2,1), (3,00), (4,01), \ldots$. Hence, the length $|x|$ of x is the number of bits in the binary string x. Let $T_1, T_2, \ldots$ be a standard enumeration of all Turing machines. Let $\langle \cdot, \cdot \rangle$ be a standard invertible effective bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. This can be iterated to $\langle \langle \cdot, \cdot \rangle, \cdot \rangle$.

Definition 8.1 *Let U be an appropriate universal Turing machine such that $U(\langle \langle i, p \rangle, y \rangle) = T_i(\langle p, y \rangle)$ for all i and $\langle p, y \rangle$. The* Kolmogorov complexity *of x given y (for free) is*

$$C(x|y) = \min\{|p| : U(\langle p, y \rangle) = x, p \in \{0,1\}^*\} \ .$$

8.3 Kolmogorov random graphs

One way to express irregularity or *randomness* of an individual network topology is by a modern notion of randomness like Kolmogorov complexity. A simple counting argument shows that for each y in the condition and each length n there exists at least one x of length n which is *incompressible* in the sense of $C(x|y) \geq n$, 50% of all x's of length n is incompressible but for 1 bit ($C(x|y) \geq n - 1$), 75% of all x's is incompressible but for 2 bits ($C(x|y) \geq n - 2$) and in general a fraction of $1 - 1/2^c$ of all strings cannot be compressed by more than c bits, [LV93].

Definition 8.2 *Each graph $G = (V, E)$ on n nodes $V = \{1, 2, \ldots, n\}$ can be coded by a binary string $E(G)$ of length $n(n-1)/2$. We enumerate the $n(n-1)/2$ possible edges uv in a graph on n nodes in standard lexicographical order without*

repetitions and set the ith bit in the string to 1 if the i-th edge is present and to 0 otherwise. Conversely, each binary string of length $n(n-1)/2$ encodes a graph on n nodes. Hence we can identify each such graph with its corresponding binary string.

Definition 8.3 *Let δ be a simply described recursive function over the natural numbers, such as $\log n, \log\log n, \sqrt{n}$. That is,*

$$C(\delta) := \min\{C(T) : T \text{ is a Turing machine computing } \delta\}$$

is bounded by a small fixed constant. An individual graph G on n nodes is δ-random *(in contrast to a randomly generated graph) if it satisfies*

$$C(E(G)|n) \geq n(n-1)/2 - \delta(n) \ . \tag{8.1}$$

Elementary counting shows that *a fraction* of at least

$$1 - 1/2^{\delta(n)}$$

of all graphs on n nodes has that high complexity, [LV93]. We need the notion of self-delimiting binary strings.

Definition 8.4 *We call x a* prefix *of y if there is a z such that $y = xz$. A set $A \subseteq \{0,1\}^*$ is* prefix-free, *if no element in A is the prefix of another element in A. A 1:1 function $E : \{0,1\}^* \to \{0,1\}^*$ (equivalently, $E : \mathbb{N} \to \{0,1\}^*$) defines a* prefix-code *if its range is prefix-free. A simple prefix-code we use throughout is obtained by reserving one symbol, say 0, as a stop sign and encoding*

$$\begin{aligned} \bar{x} &= 1^{|x|}0x \ , \\ |\bar{x}| &= 2|x| + 1 \ . \end{aligned}$$

Sometimes we need the shorter prefix-code x':

$$\begin{aligned} x' &= \overline{|x|}x \ , \\ |x'| &= |x| + 2\lceil\log(|x|+1)\rceil + 1 \ . \end{aligned}$$

We call $\bar{x}$ or x' a self-delimiting *version of the binary string x.*

We can effectively recover both x and y unambiguously from the binary strings $\bar{x}y$ or $x'y$. For example, if $\bar{x}y = 111011011$, then $x = 110$ and $y = 11$. If $\bar{x}\bar{y} = 1110110101$ then $x = 110$ and $y = 1$. The self-delimiting form $x' \ldots y'z$ allows the concatenated binary sub-descriptions to be parsed and unpacked into the individual items $x, \ldots, y, z$; the code x' encodes a separation delimiter for x, using $2\lceil\log(|x|+1)\rceil$ extra bits, and so on, [LV93].

Lemma 8.5 *The degree d of each node of a δ-random graph satisfies*

$$|d - (n-1)/2| = O\left(\sqrt{(\delta(n) + \log n)n}\right) \ .$$

Proof Assume that the deviation of the degree d of a node u in G from $(n-1)/2$ is at least k. From the lower bound on $C(E(G)|n)$ corresponding to the assumption that G is random, we can estimate an upper bound on k, as follows.

Describe $G = (V, E)$ given n as follows. We can indicate which edges are incident on node u by giving the index of the interconnection pattern (the characteristic sequence of the set $V_u = \{v \in V - \{u\} : uv \in E\}$ in $n-1$ bits where the v-th bit is 1 if $v \in V_u$ and 0 otherwise) in the ensemble of

$$m = \sum_{|d-(n-1)/2| \geq k} \binom{n-1}{d} \leq 2^n e^{-k^2/(n-1)} \tag{8.2}$$

possibilities. The last inequality follows from a general estimate of the tail probability of the binomial distribution, with s_n the number of successful outcomes in n experiments with probability of success $0 < p < 1$ and where $q = 1 - p$. Namely, Chernoff's bounds, [LV93], pp. 127-130, give

$$\Pr(|s_n - np| \geq k) \leq 2e^{-k^2/4npq} \ . \tag{8.3}$$

To describe G it then suffices to modify the old code of G by prefixing it with

- ▷ A description of this discussion in $O(1)$ bits;
- ▷ the identity of node u in $\lceil \log(n+1) \rceil$ bits;
- ▷ the value of d in $\lceil \log(n+1) \rceil$ bits, possibly adding non-significant 0's to pad up to this amount;
- ▷ the index of the interconnection pattern in $\log m + 2 \log\log m$ bits in self-delimiting form;[6]

followed by the old code for G with the bits in the code denoting the presence or absence of the possible edges which are incident on node u deleted.

Clearly, given n we can reconstruct the graph G from the new description. The total description we have achieved is an effective program of

$$\log m + 2\log\log m + O(\log n) + n(n-1)/2 - (n-1)$$

[6] From now on we write simply '$\log n$' for '$\lceil \log(n+1) \rceil$' in cases where the difference clearly doesn't matter.

bits. This must be at least the length of the shortest effective binary program, which is $C(E(G)|n)$ satisfying Eq. (8.1). Therefore,

$$\log m + 2\log\log m \geq n - 1 - O(\log n) - \delta(n) \ .$$

Since we have estimated in Eq. (8.2) that

$$\log m \leq n - 1 - (k^2/(n-1))\log e \ ,$$

it follows that

$$k = O\left(\sqrt{(\delta(n) + \log n)n}\right) \ .$$

◁

Lemma 8.6 *All $o(n)$-random graphs have diameter 2.*

Proof The only graphs with diameter 1 are the complete graphs which can be described in $O(1)$ bits, given n, and hence are not random. It remains to consider $G = (V, E)$ is an $o(n)$-random graph with diameter greater than 2. Let u, v be a pair of nodes with distance greater than 2. Then we can describe G by modifying the old code for G by prefixing it with

- ▷ A description of this discussion in $O(1)$ bits;
- ▷ The identities of $u < v$ in $2\log n$ bits;
- ▷ The old code $E(G)$ of G with, for each w with $uw \in E$, all bits representing presence or absence of an edge wv between w and v deleted. We know that all the bits representing such edges must be 0 since the existence of any such edge shows that uw, wv is a path of length 2 between u and v contradicting the assumption that u and v have distance > 2. This way we save at least $n/4$ bits, since we save bits for as many edges wv as there are edges uw, that is, the degree of u which is $n/2 \pm o(n)$ by Lemma 8.5.

Since we know the identities of u and v, and the nodes adjacent to u (they can be obtained from $E(G)$ because $u < v$) we can reconstruct G from this discussion and the new description, given n. Since by Lemma 8.5 the degree of u is at least $n/4$, the new description of G, given n, requires at most

$$n(n-1)/2 - n/4 + O(\log n)$$

bits, which contradicts Eq. (8.1) from some n onwards. ◁

Lemma 8.7 *Let c be a fixed constant. If G is $c \log n$-random then from each node u all other nodes are either directly connected to u or are directly connected to one of the least $(c+3)\log n$ nodes directly adjacent to u.*

Proof Given u, let A be the set of the least $(c+3)\log n$ nodes directly adjacent to u. Assume, by way of contradiction, there is a node w of G that is not directly connected to a node in $A \bigcup \{u\}$. We can describe G as follows.

- ▷ A description of this discussion in $O(1)$ bits;
- ▷ A literal description of u in $\log n$ bits;
- ▷ A literal description of the presence or absence of edges between u and the other nodes in $n-1$ bits;
- ▷ A literal description of w and the presence or absence of edges between w and the other nodes in $\log n + n - 2 - (c+3)\log n$ bits (by omitting the bits corresponding to the least $(c+3)\log n$ nodes directly adjacent to u);
- ▷ The encoding $E(G)$ with the edges incident with nodes u and w deleted, saving at least $2n-2$ bits.

Altogether the resultant description has

$$n(n-1)/2 + 2\log n + 2n - 3 - (c+3)\log n - 2n + 2$$

bits which contradicts the $c \log n$-randomness of G by Eq. (8.1). The lemma is proven. ◁

8.4 Upper bounds

In this section we show how one can route messages over Kolmogorov random graphs with routing schemes that can be stored efficiently. Specifically we show that in general (i.e., on almost all graphs) one can use shortest path routing schemes occupying at most $O(n^2)$ bits. If one can relabel the graph in advance, and if nodes know their neighbours, shortest path routing schemes are shown to occupy only $O(n \log^2 n)$ bits. Allowing stretch factors larger than one reduces the space requirements — even as low as $O(n)$ bits for stretch factors of $O(\log n)$.

Theorem 8.8 *For shortest path routing in $O(\log n)$-random graphs, where the port assignment may be changed or nodes know their neighbours ($IB \vee II$), it suffices to have local routing functions stored in $6n$ bits per node (hence the*

complete routing scheme is represented by $6n^2$ *bits*).

Proof We prove the theorem for the model where nodes know their neighbours (II), without resorting to relabelling (i.e., nodes are labelled 1 through n). If, instead, the port assignment may be chosen arbitrarily, we can represent knowledge of the neighbours and the edges over which they are reached as follows. Neighbours are coded using the standard interconnection vector (as in Def. 8.2) using $n-1$ bits. The port mapping is chosen such that the i-th neighbour is connected to the i-th port. This adds only $n-1$ bits per node to the local routing function to be constructed next. Hence the theorem holds for the model with arbitrary port mapping as well (IB).

Let G be an $O(\log n)$-random graph on n nodes. By Lemma 8.7 we know that from each node u we can shortest path route to each node v through the first $O(\log n)$ directly adjacent nodes of u. By Lemma 8.6, G has diameter 2. Once the message reached node v the destination is either node v or a direct neighbour of node v which is known in node v by assumption of our routing model. It follows readily from Lemma 8.7 that routing functions of size $O(n \log\log n)$ can be used to do shortest-path routing. As we will see we can do better than this. Let $A_0 \subseteq V$ be the set of nodes in G which are not directly connected to u.

Claim 8.9 *Let* $v_1, \ldots, v_m$ *be* $O(\log n)$ *directly adjacent nodes to* u *through which we can shortest path route to all nodes in* A_0 (*for example, the* $O(\log n)$ *least nodes directly adjacent to node* u, *Lemma 8.7*). *For* $t := 1, 2 \ldots, l$ *define* $A_t := \{w \in A_0 - \bigcup_{s=1}^{t-1} A_s : v_t w \in E\}$. *Let* $m_0 = |A_0|$ *and define* $m_{t+1} = m_t - |A_{t+1}|$. *Let* l *be the first* t *such that* $m_t < n/\log\log n$. *Then,* $|A_t| > 1/3 * m_{t-1}$ *for* $1 \leq t \leq l$. *This means that* v_t *is connected by an edge in* E *to at least* $1/3$ *of the nodes not connected by edges in* E *to nodes* $u, v_1, \ldots, v_{t-1}$.

Proof Suppose, by way of contradiction, that there exists a least $t \leq l$ such that $||A_t| - m_{t-1}/2| \geq (1/6)m_{t-1}$. Then we can describe G, given n, as follows.

- This discussion in $O(1)$ bits;
- Nodes u, v_t in $2 \log n$ bits;
- The presence or absence of edges incident with nodes $u, v_1, \ldots, v_{t-1}$ in $r = n - 1 + \cdots + n - (t-1)$ bits. This gives us the characteristic

sequences of $A_0, \ldots, A_{t-1}$ in V^7;

▷ A self-delimiting description of the characteristic sequence of A_t in $A_0 - \bigcup_{s=1}^{t-1} A_s$, using Chernoff's bound as cited in Eq. (8.2), in at most $m_{t-1} - (1/6)^2 m_{t-1} \log e + O(\log m_{t-1})$ bits;

▷ The description $E(G)$ with all bits corresponding to the presence or absence of edges between v_t and the nodes in $A_0 - \bigcup_{s=1}^{t-1} A_s$ deleted, saving m_{t-1} bits. Furthermore, we delete also all bits corresponding to presence or absence of edges incident with $u, v_1, \ldots, v_{t-1}$ saving a further r bits.

This description of G uses at most

$$n(n-1)/2 + O(\log n) + m_{t-1} - (1/6)^2 m_{t-1} \log e - m_{t-1}$$

bits, which contradicts the $O(\log n)$-randomness of G by Eq. (8.1), because $m_{t-1} > n/\log\log n$. ◁

Recall that l is the first time in the construction such that $m_l < n/\log\log n$. By Lemma 8.7, $l = O(\log n)$. We construct the local routing function $F(u)$ as follows.

▷ A table of intermediate routing node entries for all the nodes in A_0 in increasing order. For each node w in $\bigcup_{s=1}^{l} A_s$ we enter in the w-th position in the table the unary representation of the least intermediate node v, with $uv, vw \in E$, followed by a 0. For the nodes that are not in $\bigcup_{s=1}^{l} A_s$ we enter a 0 in their position in the table indicating that an entry for this node can be found in the second table. By Claim 8.9, the size of this table is bounded by:

$$n + \sum_{s=1}^{l} (1/3)(2/3)^{s-1} sn \leq n + \sum_{s=1}^{\infty} (1/3)(2/3)^{s-1} sn \leq 4n \ ;$$

▷ A table with explicitly binary coded intermediate nodes on a shortest path for the ordered set of the remaining destination nodes. Those nodes had a 0 entry in the first table and there are at most $m_l < n/\log\log n$ of them, namely the nodes in $A_0 - \bigcup_{s=1}^{l} A_s$. Each entry consists of the code of length $\log\log n + O(1)$ for the position in

[7] A characteristic sequence of A in V is a string of $|V|$ bits with for each $v \in V$ the v-th bit equals 1 if $v \in A$ and the v-th bit is 0 otherwise.

increasing order of a node out of $v_1, \ldots, v_m$ with $m = O(\log n)$ by Lemma 8.7. Hence this second table requires at most $2n$ bits.

The routing algorithm is as follows. The direct neighbours of u are known in node u and are routed without routing table. If we route from start node u to target node w which is not directly adjacent to u, then we do the following. If node w has an entry in the first table then route over the edge coded in unary, otherwise find an entry for node w in the second table.

Altogether, we have $|F(u)| \leq 6n$. Adding another $n - 1$ in case the port assignment may be chosen arbitrarily, this proves the theorem with $7n$ instead of $6n$. Slightly more precise counting and choosing l such that m_l is the first such quantity $< n/\log n$ shows $|F(u)| \leq 3n$. ◁

If we allow arbitrary labels for the nodes, then shortest path routing schemes of $O(n \log^2 n)$ bits suffice on Kolmogorov random graphs, as witnessed by the following theorem.

Theorem 8.10 *For shortest path routing on $c \log n$-random graphs, if nodes know their neighbours and nodes may be arbitrarily relabelled ($II \wedge \gamma$), using labels of size $(1 + (c + 3) \log n) \log n$ bits results in local routing functions stored in $O(1)$ bits per node (hence the complete routing scheme is represented by $(c + 3)n \log^2 n + n \log n + O(n)$ bits).*

Proof Let $G = (V, E)$ be a $c \log n$-random graph on n nodes. By Lemma 8.7 we know that from each node u we can shortest path route to each node w through the first $(c + 3) \log n$ directly adjacent nodes $f(u) = v_1, \ldots, v_m$ of u. By lemma 8.6, G has diameter 2. Relabel G such that the label of node u equals u followed by the original labels of the first $(c + 3) \log n$ directly adjacent nodes $f(u)$. This new label occupies $(1 + (c + 3) \log n) \log n$ bits. To route from source u to destination v do the following.

If v is directly adjacent to u we route to v in 1 step in our model (nodes know their neighbours). If v is not directly adjacent to u, we consider the immediate neighbours $f(v)$ contained in the name of v. By Lemma 8.7 at least one of the neighbours of u must have a label whose original label (stored in the first $\log n$ bits of its new label) corresponds to one of the labels in $f(v)$. Node u routes the message to any such neighbour. This routing function can be stored in $O(1)$ bits. ◁

Without relabelling routing using less than $O(n^2)$ bits is possible if we allow stretch factors larger than 1. The next three theorems clearly show a trade-off between the stretch factor and the size of the routing scheme.

Theorem 8.11 *For routing with any stretch factor* > 1 *in* $c \log n$*-random graphs, where nodes know their neighbours (II), it suffices to have* $n - 1 - (c+3)\log n$ *nodes with local routing functions stored in at most* $\lceil \log(n+1) \rceil$ *bits per node, and* $1+(c+3)\log n$ *nodes with local routing functions stored in* $6n$ *bits per node (hence the complete routing scheme is represented by less than* $(6c+20)n \log n$ *bits).*

Proof Let $G = (V, E)$ be a $c \log n$-random graph on n nodes. By Lemma 8.7 we know that from each node u we can shortest path route to each node w through the first $(c+3)\log n$ directly adjacent nodes $v_1, \ldots, v_m$ of u. By Lemma 8.6, G has diameter 2. Consequently, each node in V is directly adjacent to some node in $B = \{u, v_1, \ldots, v_m\}$. Hence, it suffices to select the nodes of B as routing centers and store, in each node $w \in B$, a shortest path routing function $F(w)$ to all other nodes, occupying $6n$ bits (the same routing function as constructed in the proof of Theorem 8.8 if the neighbours are known). Nodes $v \in V - B$ route any destination unequal to their own label to some fixed directly adjacent node $w \in B$. Then $|F(v)| \leq \lceil \log(n+1) \rceil + O(1)$, and this gives the bit count in the theorem

To route from a originating node v to a target node w the following steps are taken. If w is directly adjacent to v we route to w in 1 step in our model. If w is not directly adjacent to v then we first route in 1 step from v to its directly connected node in B, and then via a shortest path to w. Altogether, this takes either 2 or 3 steps whereas the shortest path has length 2. Hence the stretch factor is at most 1.5 which for graphs of diameter 2 (i.e., all $c \log n$-random graphs by Lemma 8.6) is the only possibility between stretch factors 1 and 2. This proves the theorem. ◁

Theorem 8.12 *For routing with stretch factor 2 in* $c \log n$*-random graphs, if nodes know their neighbours (II), it suffices to have* $n - 1$ *nodes with local routing functions stored in at most* $\log \log n$ *bits per node and* 1 *node with its local routing function stored in* $6n$ *bits (hence the complete routing scheme is represented by* $n \log \log n + 6n$ *bits).*

Proof Let G be a $c \log n$-random graph on n nodes. By Lemma 8.6, G has diameter 2. Therefore the following routing scheme has stretch factor 2. Let

node 1 store a shortest path routing function. All other nodes only store a shortest path to node 1. To route from a originating node v to a target node w the following steps are taken. If w is an immediate neighbour of v, we route to w in 1 step in our model. If not, we first route the message to node 1 in at most 2 steps, and then from node 1 through a node v to node w in again 2 steps. Because node 1 stores a shortest path routing function, either $v = w$ or w is a direct neighbour of v.

Node 1 can store a shortest path routing function in at most $6n$ bits using the same construction as used in the proof of Theorem 8.8 (if the neighbours are known). The immediate neighbours of 1 either route to 1 or directly to the destination of the message. For these nodes, the routing function occupies $O(1)$ bits. For nodes v at distance 2 of node 1 we use Lemma 8.7, which tells us that we can shortest path route to node 1 through the first $(c + 3)\log n$ directly adjacent nodes of v. Hence, to represent this edge takes $\log\log n + \log(c + 3)$ bits and hence the local routing function $F(v)$ occupies at most $\log\log n + O(1)$ bits. ◁

Theorem 8.13 *For routing with stretch factor $(c + 3)\log n$ in $c\log n$-random graphs, where nodes know their neighbours (II), it suffices to have local routing functions stored in $O(1)$ bits per node (hence the complete routing scheme is represented by $O(n)$ bits).*

Proof From Lemma 8.7 we know that from each node u we can shortest path route to each node v through the first $(c + 3)\log n$ directly adjacent nodes of u. By Lemma 8.6, G has diameter 2. So the local routing function — representable in $O(1)$ bits — is to route directly to the target node if it is a directly adjacent node, otherwise to simply traverse the first $(c + 3)\log n$ incident edges of the starting node and look in each of the visited nodes whether the target node is a directly adjacent node. If so, the message is forwarded to that node, otherwise it is returned to the starting node for trying the next node. Hence each message for a destination at distance 2 traverses at most $2(c + 3)\log n$ edges. ◁

8.5 Lower bounds

The first two theorems of this section together show that indeed $\Omega(n^2)$ bits are necessary to route on Kolmogorov random graphs in all models we consider, except for the models where nodes know their neighbours and relabelling is allowed (II $\wedge$ β and II $\wedge$ γ). Hence the upper bound in Theorem 8.8

is tight.

Theorem 8.14 *For shortest path routing in $o(n)$-random graphs where relabelling is not allowed and nodes know their neighbours ($II \wedge \alpha$), each local routing function must be stored in at least $n/2 - o(n)$ bits per node (hence the complete routing scheme requires at least $n^2/2 - o(n^2)$ bits to be stored).*

Proof Let G be a $o(n)$-random graph. Let $F(u)$ be the local routing function of node u of G, and let $|F(u)|$ be the number of bits used to store $F(u)$. Let $E(G)$ be the standard encoding of G in $n(n-1)/2$ bits as in Def. 8.2. We now give another way to describe G using some local routing function $F(u)$.

- ▷ A description of this discussion in $O(1)$ bits;
- ▷ A description of u in $\log n$ bits (if it is less pad the description with 0's);
- ▷ A description of the presence or absence of edges between u and the other nodes in V in $n-1$ bits;
- ▷ A description of $F(u)$ in $|F(u)| + O(\log |F(u)|)$ bits (the logarithmic term to make the description self-delimiting);
- ▷ The code $E(G)$ with all bits deleted corresponding to edges $vw \in E$ for each w and v such that $F(u)$ routes messages to w through the least[8] intermediary node v (saving at least $n/2-o(n)$ bits since there are at least $n/2 - o(n)$ nodes w such that $uw \notin E$ by Lemma 8.5 and since the diameter of G is 2 by Lemma 8.6 there is a shortest path $uv, vw \in E^2$ for some v).[9] Furthermore, also all bits deleted corresponding to the presence or absence of edges between u and the other nodes in V, saving another $n-1$ bits.

From this description we can reconstruct G, given n, by reconstructing the bits corresponding to the deleted edges from u and $F(u)$ and subsequently

[8] $F(u)$ may route from node u to node w by many different paths of length 2 using more than one intermediary node v. Allowing this may have as consequence that there is a more compact way of encoding $F(u)$, and makes our lower bound results stronger.

[9] Note that we cannot save the bits for the presence or absence of edges uv and uw. Namely, the routing function $F(u)$ may use the particular connection pattern of u saying things like 'route to node w using the 30th least node directly connected to u'. Thus, $F(u)$ may route *using* the connection pattern of node u, while that pattern cannot be reconstructed from $F(u)$.

inserting them in the appropriate positions to reconstruct $E(G)$. We can do so because these positions can be simply reconstructed in increasing order. In total this new description has

$$n(n-1)/2 + O(1) + O(\log n) + |F(u)| - n/2 + o(n)$$

bits which must be at least $n(n-1)/2 - o(n)$ by Eq. (8.1). We conclude that $|F(u)| = n/2 - o(n)$, which proves the theorem. ◁

Theorem 8.15 *For shortest path routing in $o(n)$-random graphs, if the neighbours are not known ($IA \vee IB$), the complete routing scheme requires at least $n^2/32 - o(n^2)$ bits to be stored.*

Proof In the proof of this theorem we need the following combinatorial result.

Claim 8.16 *Let k and n be arbitrary natural numbers such that $1 \leq k \leq n$. Let x_i, for $1 \leq i \leq k$, be natural numbers such that $x_i \geq 1$. If $\sum_{i=1}^{k} x_i = n$, then*

$$\sum_{i=1}^{k} \lceil \log x_i \rceil \leq n - k \ .$$

Proof By induction on k. If $k = 1$, then $x_1 = n$ and clearly $\lceil \log n \rceil \leq n - 1$ if $n \geq 1$. Supposing the claim holds for k and arbitrary n and x_i, we now prove it for $k' = k + 1$, n and arbitrary x_i. Let $\sum_{i=1}^{k'} x_i = n$. Then $\sum_{i=1}^{k} x_i = n - x_{k'}$. Now

$$\sum_{i=1}^{k'} \lceil \log x_i \rceil = \sum_{i=1}^{k} \lceil \log x_i \rceil + \lceil \log x_{k'} \rceil \ .$$

By the induction hypothesis the first term on the right-hand side is less than or equal to $\leq n - x_{k'} - k$, so

$$\sum_{i=1}^{k'} \lceil \log x_i \rceil \leq n - x_{k'} - k + \lceil \log x_{k'} \rceil = n - k' + \lceil \log x_{k'} \rceil + 1 - x_{k'} \ .$$

Clearly $\lceil \log x_{k'} \rceil + 1 \leq x_{k'}$ if $x_{k'} \geq 1$, which proves the claim. ◁

If we cannot enumerate the labels of all nodes in less than $n^2/32$ we are done. So assume we can (this includes models α and β where the labels are not charged for, but can be described using $\log n$ bits). Let G be an $o(n)$-random graph.

Claim 8.17 *Given the labels of all nodes, we can describe the interconnection pattern of a node u using the local routing function of node u plus an additional $n/2 + o(n)$ bits.*

Proof Apply the local routing function to each of the labels of the nodes in turn (these are given by assumption). This will return for each edge a list of destinations reached over that edge. To describe the interconnection pattern it remains to encode, for each edge, which of the destinations reached is actually its immediate neighbour. If edge i routes x_i destinations, this will cost $\lceil \log x_i \rceil$ bits. By Lemma 8.5 the degree of a node in G is at least $n/2 - o(n)$. Then in total, $\sum_{i=1}^{n/2-o(n)} \lceil \log x_i \rceil$ bits will be necessary; separations need not be encoded because they can be determined using the knowledge of all x_i's. Using Claim 8.16 finishes the proof. ◁

Now we show that there are $n/2$ nodes in G whose local routing function requires at least $n/8 - 3\log n$ bits to describe (which implies the theorem).

Assume, by way of contradiction, that there are $n/2$ nodes in G whose local routing function requires at most $n/8 - 3\log n$ bits to describe. Then we can describe G as follows:

- ▷ A description of this discussion in $O(1)$ bits;
- ▷ The enumeration of all labels in at most $n^2/32$ (by assumption);
- ▷ A description of the $n/2$ nodes in this enumeration in at most n bits;
- ▷ The interconnection patterns of these $n/2$ nodes in $n/8-3\log n$ plus $n/2 + o(n)$ bits each (by assumption, and using Claim 8.17). This amounts to $n/2(5n/8-3\log n)+o(n^2)$ bits in total, with separations encoded in another $n\log n$ bits;
- ▷ The interconnection patterns of the remaining $n/2$ nodes *only among themselves* using the standard encoding, in $1/2(n/2)^2$ bits.

This description altogether uses

$$\begin{aligned} &O(1) + n^2/32 + n + n/2(5n/8 - 3\log n) + \\ &\quad + o(n^2) + n\log n + 1/2(n/2)^2 = \\ &= n^2/2 - n^2/32 + n + o(n^2) - n/2\log n \end{aligned}$$

bits, contradicting the $o(n)$-randomness of G by Eq. (8.1). We conclude that on at least $n/2$ nodes a total of $n^2/16 - o(n^2)$ bits are used to store the routing scheme. ◁

If neither relabelling nor changing the port assignment is allowed, the next theorem implies that for shortest path routing on such 'static' graphs one cannot do better than storing the routing tables literally, in $O(n^2 \log n)$ bits.

Theorem 8.18 *For shortest path routing in $o(n)$-random graphs where relabelling and changing the port assignment is not allowed ($IA \wedge \alpha$), each local routing function must be stored in at least $n/2 \log n/2 - O(n)$ bits per node (hence the complete routing scheme requires at least $n^2/2 \log n/2 - O(n^2)$ bits to be stored).*

Proof If the graph cannot be relabelled and the port assignment cannot be changed, the adversary can set the port assignment of each node to correspond to a permutation of the neighbours. As each node has $n/2 - o(n)$ neighbours by Lemma 8.5, such a permutation can have Kolmogorov complexity as high as $n/2 \log n/2 - O(n)$ [LV93]. Because the neighbours are not known, the local routing function must for each neighbour determine the port to route messages for that neighbour over. Hence the local routing function completely describes the permutation and thus it must also occupy at least $n/2 \log n/2 - O(n)$ bits. This proves the theorem. ◁

Even if stretch factors between 1 and 2 are allowed, the next theorem shows that $\Omega(n^2 \log n)$ bits are necessary to represent the routing scheme in the worst case.

Theorem 8.19 *For routing with stretch factor < 2 in graphs where relabelling is not allowed (α), there exist graphs on n nodes (almost $(n/3)!$ such graphs) where the local routing function must be stored in at least $(n/3) \log n - O(n)$ bits per node at $n/3$ nodes (hence the complete routing scheme requires at least $(n^2/9) \log n - O(n^2)$ bits to be stored).*

Proof Consider the graph G_k with $n = 3k$ nodes depicted in Figure 8.1. Each node v_i in $v_{k+1}, \ldots, v_{2k}$ is connected to v_{i+k} and to each of the nodes $v_1, \ldots, v_k$. Fix a labelling of the nodes $v_1, \ldots, v_{2k}$ with labels from $\{1, \ldots, 2k\}$. Then any labelling of the nodes $v_{2k+1}, \ldots, v_{3k}$ with labels from $\{2k+1, \ldots, 3k\}$ corresponds to a permutation of $\{2k+1, \ldots, 3k\}$ and vice versa.

Clearly, for any two nodes v_i and v_j with $1 \leq i \leq k$ and $2k+1 \leq j \leq 3k$, the shortest path from v_i to v_j passes through node v_{j-k} and has length 2, whereas any other path from v_i to v_j has length at least 4. Hence any routing function on G_k with stretch factor < 2 routes such v_j from v_i

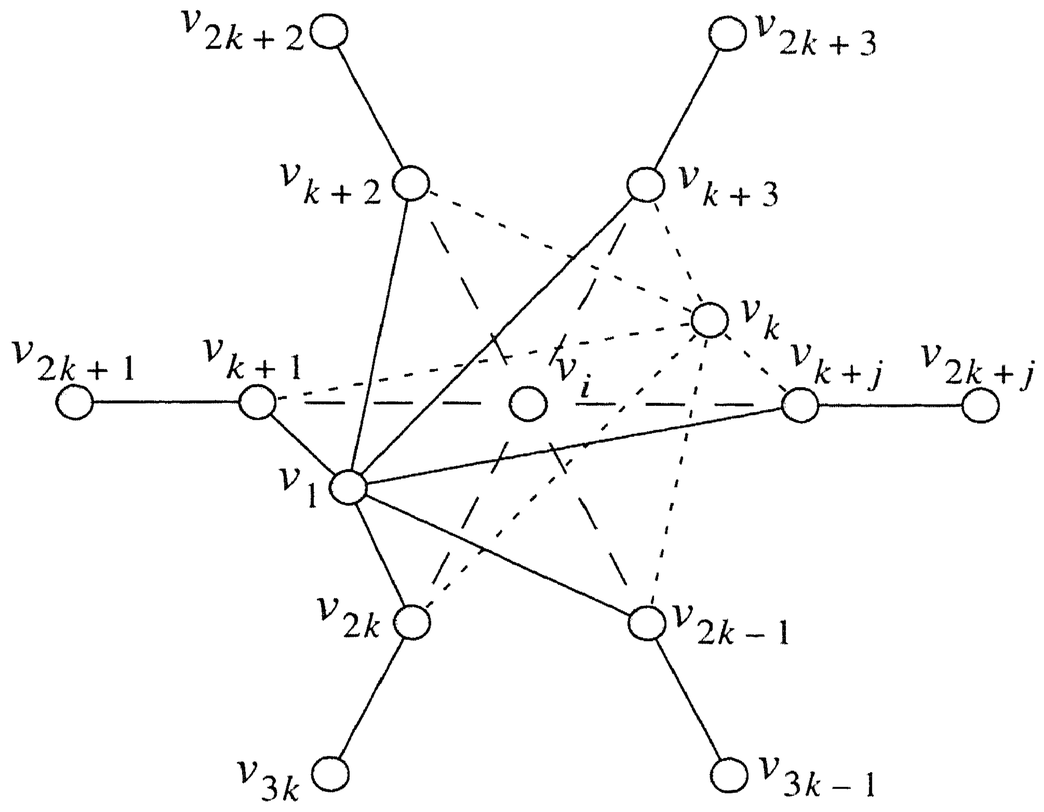

Figure 8.1 *Graph* G_k.

over the edge $v_i v_{j-k}$. Then at each of the k nodes $v_1, \ldots, v_k$ the local routing functions corresponding to any two labellings of the nodes $v_{2k+1}, \ldots, v_{3k}$ are different. Hence each representation of a local routing function at the k nodes v_i, $1 \leq i \leq k$, corresponds one-one to a permutation of $\{2k+1, \ldots, 3k\}$. So given such a local routing function we can reconstruct the permutation (by collecting the response of the local routing function for each of the nodes $k+1, \ldots, 3k$ and grouping all pairs reached over the same edge). The number of such permutations is $k!$. A fraction at least $1 - 1/2^k$ of such permutations π has Kolmogorov complexity $C(\pi) = k \log k - O(k)$ [LV93]. Because π can be reconstructed given any of the k local routing functions, these k local routing functions each must have Kolmogorov complexity $k \log k - O(k)$ too. This proves the theorem for n a multiple of 3. For $n = 3k-1$ or $n = 3k-2$ we can use G_k, dropping v_k and v_{k-1}. ◁

Our last theorem shows that for full information shortest path routing schemes on Kolmogorov random graphs one cannot do better than the trivial upper bound.

Theorem 8.20 *For full-information shortest path routing on $o(n)$-random graphs where relabelling is not allowed (α), the local routing function occupies*

at least $n^2/4 - o(n^2)$ *bits for every node (hence the complete routing scheme requires at least* $n^3/4 - o(n^3)$ *bits to be stored).*

Proof Let G be a graph on nodes $\{1, 2, \ldots, n\}$ satisfying Eq. (8.1) with $\delta(n) = o(n)$. Then we know that G satisfies Lemmas 8.5, 8.6. Let $F(u)$ be the local routing function of node u of G, and let $|F(u)|$ be the number of bits used to encode $F(u)$. Let $E(G)$ be the standard encoding of G in $n(n-1)/2$ bits as in Def. 8.2. We now give another way to describe G using some local routing function $F(u)$.

- ▷ A description of this discussion in $O(1)$ bits;
- ▷ A description of u in $\log n$ bits (if it is less pad the description with 0's);
- ▷ A description of the presence or absence of edges between u and the other nodes in V in $n-1$ bits;
- ▷ A description of $F(u)$ in $|F(u)| + O(\log |F(u)|)$ bits (the logarithmic term to make the description self-delimiting);
- ▷ The code $E(G)$ with all bits deleted corresponding to the presence or absence of edges between each w and v such that v is a neighbour of u and w is not a neighbour of u. Since there are at least $n/2 - o(n)$ nodes w such that $uw \notin E$ and at least $n/2 - o(n)$ nodes v such that $uv \in E$, by Lemma 8.5, this saves at least $(n/2 - o(n))^2$ bits.

From this description we can reconstruct G, given n, by reconstructing the bits corresponding to the deleted edges from u and $F(u)$ and subsequently inserting them in the appropriate positions to reconstruct $E(G)$. We can do so because $F(u)$ represents a full information routing scheme implying that $vw \in E$ iff uv is among the edges used to route from u to w. In total this new description has

$$n(n-1)/2 + O(\log n) + |F(u)| - n^2/4 + o(n^2)$$

bits which must be at least $n(n-1)/2 - o(n)$ by Eq. (8.1). We conclude that $|F(u)| = n^2/4 - o(n^2)$, which proves the theorem. ◁

8.6 Average routing

We now extend our results to the average cost, taken over all labelled graphs of n nodes, of representing a routing scheme for graphs over n nodes.

Definition 8.21 *For a graph G, let $T(G)$ be the number of bits used to store its routing scheme. The* average *total number of bits to store the routing scheme for routing over graphs on n nodes is $\sum T(G)/2^{n(n-1)/2}$ with the sum taken over all graphs G on nodes $\{1, 2, \ldots, n\}$. That is, the uniform average over all the labelled graphs on n nodes.*

The results on Kolmogorov random graphs above have the following corollaries. Consider the subset of $(3 \log n)$-random graphs within the class of $O(\log n)$-random graphs on n nodes. They constitute a fraction of at least $(1 - 1/n^3)$ of the class of all graphs on n nodes. The trivial upper bound on the minimal total number of bits for all routing functions together is $O(n^2 \log n)$ for shortest path routing on all graphs on n nodes (or $O(n^3)$ for full-information shortest path routing). Simple computation of the average of the total number of bits used to store the routing scheme over all graphs on n nodes shows the following.

Corollary 8.22 *The average total number of bits to store the routing scheme for graphs of n nodes is:*

1. $O(n^2)$ *for shortest path routing in model IB $\vee$ II (Theorem 8.8),*
2. $O(n \log^2 n)$ *for shortest path routing in model II $\wedge$ γ (Theorem 8.10),*
3. $O(n \log n)$ *for routing with any stretch factor s for $1 < s < 2$ in model II (Theorem 8.11),*
4. $O(n \log \log n)$ *for routing with stretch factor 2 in model II (Theorem 8.12),*
5. $O(n)$ *for routing with stretch factor $6 \log n$ in model II (Theorem 8.13 with $c = 3$),*
6. $\Omega(n^2)$ *for shortest path routing in model α, IA and IB (Theorem 8.14 and Theorem 8.15),*
7. $\Omega(n^2 \log n)$ *for shortest path routing in model IA $\wedge$ α (Theorem 8.18),*
8. $\Theta(n^3)$ *for full information shortest path routing in model α (Theorem 8.20).*

Acknowledgements

We thank Jan van Leeuwen, Evangelos Kranakis and Danny Krizanc for helpful discussions.

Cogito ergo sum
is de uitdrukking van een intellectueel
die kiespijn onderschat

MILAN KUNDERA
Onsterfelijkheid

A Sammenvatting

Communicatie netwerken als het Internet verbinden computers onderling. Dit maakt het mogelijk om met meerdere computers tegelijk een taak te verrichten. Daarbij vormen de computers en het netwerk een zogenaamd gedistribueerd systeem. Communicatie, synchronizatie en fout-tolerantie vormen een belangrijk bestanddeel van zo'n systeem. Ze worden echter zelden volledig door de hardware ondersteund, en zeker niet tot op het noodzakelijke nivo.

Onderzoek naar gedistribueerde algoritmen heeft tot doel efficiente algoritmen voor allerlei nivo's van communicatie, synchronizatie en fout-tolerantie te ontwikkelen en te analyseren. Dit proefschrift analyseert een zevental problemen op dit gebied, en bevat verschillende algoritmen als oplossing daarvan.

Routering is een bekend communicatie probleem. Voor het effectief versturen van boodschappen over wereldwijde netwerken als het Internet moet de bestemming binnen afzienbare tijd bereikt worden. Dit proefschrift bewijst dat dit mogelijk is mits men bereid is om op elk van de computers op de route gemiddeld veel geheugen te spenderen aan de routerings-tabellen.

Computer fouten komen veelvuldig voor. Computers kunnen vastlopen; kortstondige storingen kunnen waarden opgeslagen in het geheugen veranderen. Fout-tolerante algoritmen verbergen of verhelpen de effecten van dergelijke fouten.

Zelf-stabiliserende algoritmen herstellen automatisch van geheugenfouten. In dit proefschrift worden twee van zulke algoritmen beschreven: voor wederzijdse uitsluiting en voor ring-orientatie.

Zogenaamde wait-free algoritmen laten een computer zinvol voortgang maken zelfs als alle andere computers zijn vastgelopen. In dit proef-

schrift wordt een wait-free implementatie van snapshot objecten uit binaire snapshot objecten beschreven, en wordt het eerste snelle algoritme voor long-lived renaming afgeleid.

Een belangrijk probleem is het bereiken van overeenstemming tussen computers over, bijvoorbeeld, een gemeten waarde, terwijl een fractie van de computers (eventueel willens en wetens) foutief functioneerd. Alle correcte computers moeten eenmalig allemaal dezelfde meetwaarde kiezen. De gekozen meetwaarde moet ook echt door één van de computers gemeten zijn. Dit probleem is oplosbaar als ten alle tijde slechts 1/3-de van de computers geïnfecteerd is. Zelfs als de computer fouten zich, als virussen, over het netwerk verspreiden, tenminste als ook nog één van de computers altijd gezond is.

Tenslotte wordt in dit proefschrift een aanzet gegeven voor onderzoek naar extreem fout-tolerante algoritmen die zowel geheugen-fouten als computer crashes kunnen overleven. Eerst wordt een algemene definitie van wait-free zelf-stabilizerende geheugen objecten gegeven; daarna komen enkele implementaties aan bod.

Neamsto dit sé hwer is it wetter?
Is it krekt fuort of komt it letter?
Ik ha myn nocht
Ik wol nei hus ta al jierrenlang

DEINUM
Swarte Hoanne

B Curriculum Vitæ

Born: 12 November 1966, Groningen.

1978 – 1884 : Grammar school (Gymnasium β), at the Stedelijk Gymnasium, Leeuwarden. Eight subjects, including philosophy.

1984 – 1989 : Doctoral in Computer Science, Computer Science Dept., University of Groningen (RuG). Graduated on the following three subjects:

- ▷ Computer science — *Informatica*,
- ▷ Pure and applied mathematics — *Zuivere en toegepaste wiskunde*, and
- ▷ Selected topics from psychology — *Hoofdstukken uit de Psychologie*.

The final year of my studies, I took part in the following two graduation seminars.

- ▷ *Logics for deductive databases*, and
- ▷ *Parallel and distributed information processing*.

1990 – 1991 : As a conscientious objector, I fulfilled my social service as a research fellow at the Law and Informatics Dept., University of Groningen (RuG).

Tasks: programming an expert system to support the application and enforcement of environmental law. Exploring the non-monotonic logics behind such rule-based systems, especially in the context of reasoning based on law and legislation.

1992 - 1996 : PhD student (OiO), Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

Research interests: Design and analysis of distributed algorithms; understanding the structure and complexity of synchronization and communication in distributed systems; fault-tolerance (especially self-stabilization).

In the near future I would like to focus on mobile computing. How does 'mobility' change the way we think about distributed algorithms? Are there any new fundamental problems involved? What are the main applications for mobile computing, and how must we design systems supporting those applications? Answers to these fundamental questions can directly be applied in the actual design of new distributed systems and applications for mobile computing.

Email: `jhh@cwi.nl` .

C — Big blue wobbly thing that mermaids live in.

BLACKADDER THE THIRD
Ink and Incapability

C Publications

All research reported in this thesis has been published in the proceedings of several conferences. In particular,

Chapter 2: JAAP-HENK HOEPMAN AND JOHN TROMP. Binary Snapshots. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, September 27–29, 1993), André Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 18–25.

Chapter 3: HARRY BUHRMAN, JUAN A. GARAY, JAAP-HENK HOEPMAN AND MARK MOIR. Long-Lived Renaming Made Fast. In *14th Ann. Symp. on Principles of Distributed Computing* (Ottawa, Ont., Canada, August 20–23, 1995), ACM Press, pp. 194–203.

Chapter 4: DICK ALSTEIN, JAAP-HENK HOEPMAN, BRYAN E. OLIVIER AND PASCALE I. A. VAN DER PUT. Self-Stabilizing Mutual Exclusion on Directed Graphs (extended abstract). In *Computer Science in the Netherlands* (Utrecht, The Netherlands, November 21–22, 1994), E. Backer (Ed.), Stichting Mathematisch Centrum, Amsterdam, pp. 42–53.

Chapter 5: JAAP-HENK HOEPMAN. Uniform Deterministic Self-Stabilizing Ring-Orientation on Odd-Length Rings. In *8th Int. Workshop on Distributed Algorithms* (Terschelling, The Netherlands, September 29 – October 1, 1994), Gerard Tel and Paul M. B. Vitányi (Eds.), Lect. Not. Comp. Sci. 857, Springer-Verlag, pp. 265–279.

Chapter 6: JAAP-HENK HOEPMAN, MARINA PAPATRIANTAFILOU AND PHILIPPAS TSIGAS. Self-Stabilization of Wait-Free Shared Memory Objects. In *9th Int. Workshop on Distributed Algorithms* (Le Mont-Saint-Michel, France, September 12–15, 1995), Jean-Michel Hélary and Michel Ray-

nal (Eds.), Lect. Not. Comp. Sci. 972, Springer, pp. 273–287.

Chapter 7: HARRY BUHRMAN, JUAN A. GARAY AND JAAP-HENK HOEPMAN. Optimal Resiliency Against Mobile Faults. In *25th Int. Symp. on Fault-Tolerant Computing* (Pasadena, CA, USA, June 27–30, 1995), IEEE Comp. Soc. Press, pp. 83–88.

Chapter 8: HARRY BUHRMAN, JAAP-HENK HOEPMAN AND PAUL M. B. VITÁNYI. Optimal Routing Tables. In *15th Ann. Symp. on Principles of Distributed Computing* (Philadelphia, PA, USA, May 23–26, 1996), ACM Press (to appear).

I would again like to take this opportunity to thank all my co-authors. Had I written all these papers on my own, they would have been much less interesting; they might not even have been written at all. It was a real pleasure to work with all of you. The same goes for all those people I met during these years and with whom I discussed many of the topics covered in this thesis.

How do you hide from
Something you have found

NOMEANSNO
It's Catching Up

D Bibliography

AAD+90 Y. AFEK, H. ATTIYA, D. DOLEV, E. GAFNI, M. MERRITT, AND N. SHAVIT. Atomic snapshots of shared memory. In *9th Ann. Symp. on Principles of Distributed Computing* (Quebec City, Qu., Canada, 1990), ACM Press, pp. 1–13.

AAD+93 Y. AFEK, H. ATTIYA, D. DOLEV, E. GAFNI, M. MERRITT, AND N. SHAVIT. Atomic snapshots of shared memory. *Journal of the ACM* **40**, 4 (1993), 873–890.

AGT+92 Y. AFEK, E. GAFNI, J. TROMP, AND P. M. B. VITÁNYI. Wait-free test-and-set. In *6th Int. Workshop on Distributed Algorithms* (Haifa, Israel, 1992), A. Segall and S. Zaks (Eds.), Lect. Not. Comp. Sci. 647, Springer-Verlag, pp. 85–94.

AGM+92 Y. AFEK, D. S. GREENBERG, M. MERRITT, AND G. TAUBENFELD. Computing with faulty shared memory. In *11th Ann. Symp. on Principles of Distributed Computing* (Vancouver, B.C., Canada, 1992), ACM Press, pp. 47–58.

AKY90 Y. AFEK, S. KUTTEN, AND M. YUNG. Memory-efficient self stabilizing protocols for general networks. In *4th Int. Workshop on Distributed Algorithms* (Bari, Italy, 1990), J. van Leeuwen and N. Santoro (Eds.), Lect. Not. Comp. Sci. 486, Springer-Verlag, pp. 15–28.

AMT93 Y. AFEK, M. MERRITT, AND G. TAUBENFELD. Benign failure models for shared memory. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, 1993), A. Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 69–83.

AHO'94a D. ALSTEIN, J.-H. HOEPMAN, B. E. OLIVIER, AND P. I. A. VAN DER PUT. Self-stabilizing mutual exclusion on directed graphs. Tech. Rep. CS-R9513, Stichting Mathematisch Centrum (CWI), Amsterdam, 1994.

AHO+94b D. ALSTEIN, J.-H. HOEPMAN, B. E. OLIVIER, AND P. I. A. VAN DER PUT. Self-stabilizing mutual exclusion on directed graphs. In *Computer Science in the Netherlands* (Utrecht, The Netherlands, 1994), E. Backer (Ed.), Stichting Mathematisch Centrum, Amsterdam, pp. 42–53.

AH93 E. ANAGNOSTOU AND V. HADZILACOS. Tolerating transient and permanent failures. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, 1993), A. Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 174–188.

And90 J. H. ANDERSON. Composite registers. In *9th Ann. Symp. on Principles of Distributed Computing* (Quebec City, Qu., Canada, 1990), ACM Press, pp. 15–29.

And93 J. H. ANDERSON. Composite registers. *Distributed Computing* **6**, 3 (1993), 141–154.

And94 J. H. ANDERSON. Multi-writer composite registers. *Distributed Computing* **7**, 4 (1994), 175–195.

AM94 J. H. ANDERSON AND M. MOIR. Using k-exclusion to implement resilient, scalable shared objects. In *13th Ann. Symp. on Principles of Distributed Computing* (Los Angeles, CA, USA, 1994), ACM Press, pp. 141–150.

AH90 J. ASPNES AND M. P. HERLIHY. Wait-free data structures in the asynchronous PRAM model. In *2nd Ann. Symp. on Parallel Algorithms and Architectures* (Crete, Greece, 1990), ACM Press, pp. 340–349.

ABND+90 H. ATTIYA, A. BAR-NOY, D. DOLEV, D. PELEG, AND R. REISCHUK. Renaming in an asynchronous environment. *Journal of the ACM* **37**, 3 (1990), 524–548.

AHR92 H. ATTIYA, M. P. HERLIHY, AND O. RACHMAN. Efficient atomic snapshots using lattice agreement. In *6th Int. Workshop on Distributed Algorithms* (Haifa, Israel, 1992), A. Segall and S. Zaks (Eds.), Lect. Not. Comp. Sci. 647, Springer-Verlag, pp. 35–53.

AR93 H. ATTIYA AND O. RACHMAN. Atomic snapshots in $O(n \log n)$ operations. In *12th Ann. Symp. on Principles of Distributed Computing* (Ithaca, NY, USA, 1993), ACM Press, pp. 29–40.

ASW88 H. ATTIYA, M. SNIR, AND M. K. WARMUTH. Computing on an anonymous ring. *Journal of the ACM* **35**, 4 (1988), 845–875.

AKK+88 B. AWERBUCH, L. M. KIROUSIS, E. KRANAKIS, AND P. M. B. VITÁNYI. A proof technique for register atomicity. In *8th Conf. on Foundations of Software Technology and Theoretical Computer Science* (Pune, India, 1988), K. V. Nori and S. Kumar (Eds.), Lect. Not. Comp. Sci. 338, Springer Verlag, pp. 286–303.

AKM+93 B. AWERBUCH, S. KUTTEN, Y. MANSOUR, B. PATT-SHAMIR, AND G. VARGHESE. Time optimal self-stabilizing synchronization. In *25th Ann. Symp. on Theory of Computing* (San Diego, CA, USA, 1993), ACM Press, pp. 652–661.

B

BND89 A. BAR-NOY AND D. DOLEV. Shared-memory vs. message-passing in an asynchronous distributed environment. In *8th Ann. Symp. on Principles of Distributed Computing* (Edmonton, Alb., Canada, 1989), ACM Press, pp. 307–318.

BNDD+87 A. BAR-NOY, D. DOLEV, C. DWORK, AND H. R. STRONG. Shifting gears: changing algorithms on the fly to expedite byzantine agreement. In *6th Ann. Symp. on Principles of Distributed Computing* (Vancouver, B.C., Canada, 1987), ACM Press, pp. 42–51.

BNDK+91 A. BAR-NOY, D. DOLEV, D. KOLLER, AND D. PELEG. Fault-tolerant critical section management in asynchronous environments. *Information and Computation* **95**, 1 (1991), 1–20.

BG89 P. BERMAN AND J. A. GARAY. Asymptotically optimal distributed consensus. In *16th Int. Conf. on Automata, Languages and Programming* (Stresa, Italy, 1989), G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi della Rocca (Eds.), Lect. Not. Comp. Sci. 372, Springer-Verlag, pp. 80–94.

BG93 E. BOROWSKY AND E. GAFNI. Immediate atomic snapshots and fast renaming. In *12th Ann. Symp. on Principles of Distributed Computing* (Ithaca, NY, USA, 1993), ACM Press, pp. 41–51.

BGW89 G. M. BROWN, M. G. GOUDA, AND C.-L. WU. Token systems that self-stabilize. *IEEE Transactions on Computers* **38**, 6 (1989), 845–852.

BGH95a H. BUHRMAN, J. A. GARAY, AND J.-H. HOEPMAN. Optimal resiliency against mobile faults. In *25th Int. Symp. on Fault-Tolerant Computing* (Pasadena, CA, USA, 1995), IEEE Comp. Soc. Press, pp. 83–88.

BGH+95b H. BUHRMAN, J. A. GARAY, J.-H. HOEPMAN, AND M. MOIR. Long-lived renaming made fast. In *14th Ann. Symp. on Principles of Distributed Computing* (Ottawa, Ont., Canada, 1995), ACM Press, pp. 194–203.

BHL95 H. BUHRMAN, E. HEMASPAANDRA, AND L. LONGPRÉ. SPARSE reduces conjunctively to TALLY. *SIAM Journal on Computing* **24**, 4 (1995), 673–681.

BHV96 H. BUHRMAN, J.-H. HOEPMAN, AND P. M. B. VITÁNYI. Optimal routing tables. In *15th Ann. Symp. on Principles of Distributed Computing* (Philadelphia, PA, USA, 1996), ACM Press.

BLS93 H. BUHRMAN, L. LONGPRÉ, AND E. SPAAN. SPARSE reduces conjunctively to TALLY. In *Proc. Structure in Complexity Theory 8th annual conference* (San Diego, CA, USA, 1993), IEEE Computer Society Press, pp. 208–214.

Bur87 J. E. BURNS. Self-stabilizing rings without demons. Tech. Rep. GIT-ICS-87/36, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332, 1987.

BGM93 J. E. BURNS, M. G. GOUDA, AND R. E. MILLER. Stabilization and pseudo-stabilization. *Distributed Computing* **7**, 1 (1993), 35–42.

BP89a J. E. BURNS AND J. PACHL. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems* **11**, 2 (1989), 330–344.

BP89b J. E. BURNS AND G. L. PETERSON. The ambiguity of choosing. In *8th Ann. Symp. on Principles of Distributed Computing* (Edmonton, Alb., Canada, 1989), ACM Press, pp. 145–157.

C

CH94 R. CANETTI AND A. HERZBERG. Maintaining security in the presence of transient faults. In *14th Int. Ann. Cryptology Conference* (Santa Barbara, CA, USA, 1994), Y. G. Desmedt (Ed.), Lect. Not. Comp. Sci. 839, Springer, pp. 425–438.

CT91 T. D. CHANDRA AND S. TOUEG. Unreliable failure detectors for asynchronous systems. In *10th Ann. Symp. on Principles of Distributed Computing* (Montreal, Qu., Canada, 1991), ACM Press, pp. 325–340.

Che52 L. CHEBYSHEV. Mémoire sur les nombres premiers. *Journal de Math.* **17** (1852), 366–390.

Coh74 P. M. COHN. *Algebra Volume 1.* John Wiley & Sons, 1974.

D

Dij65 E. W. DIJKSTRA. Solution of a problem in concurrent programming control. *Communications of the ACM* **8**, 9 (1965), 569.

Dij74 E. W. DIJKSTRA. Self-stabilizing systems in spite of distributed control. *Communications of the ACM* **17**, 11 (1974), 643–644.

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

DS82 D. DOLEV AND H. R. STRONG. Polynomial algorithms for multiple processor agreement. In *14th Ann. Symp. on Theory of Computing* (San Fransisco, CA, USA, 1982), ACM Press, pp. 401–407.

DIM93 S. DOLEV, A. ISRAELI, AND S. MORAN. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing* **7**, 1 (1993), 3–16.

DW93 S. DOLEV AND J. L. WELCH. Wait-free clock synchronization. In *12th Ann. Symp. on Principles of Distributed Computing* (Ithaca, NY, USA, 1993), ACM Press, pp. 97–108.

DHP+92 C. DWORK, M. P. HERLIHY, S. A. PLOTKIN, AND O. WAARTS. Time-lapse snapshots. In *Israel Symposium Theory of Computing and Systems* (Haifa, Israel, 1992), D. Dolev, Z. Galil, and M. Rodeh (Eds.), Lect. Not. Comp. Sci. 601, Springer Verlag, pp. 154–170.

DM90 C. DWORK AND Y. MOSES. Knowledge and common knowledge in a byzantine environment: Crash failures. *Information and Computation* **88**, 2 (1990), 156–186.

E

Ein16 A. EINSTEIN. *Über die spezielle und algemeine Relativitätstheorie.* Friedr. Vieweg & Sohn, Braunschweig, 1916.

EFF85 P. ERDÖS, P. FRANKL, AND Z. FÜREDI. Families of finite sets in which no set is covered by the union of r others. *Israel Journal of Mathematics* **51**, 1–2 (1985), 79–89.

F

FM88 P. FELDMAN AND S. MICALI. Optimal algorithms for byzantine agreement. In *20th Ann. Symp. on Theory of Computing* (Chicago, IL, USA, 1988), ACM Press, pp. 148–161.

FLMS95 M. FLAMMINI, J. VAN LEEUWEN, AND A. MARCHETTI-SPACCAMELA. The complexity of interval routing on random graphs. In *20th Int. Symp. on Mathematical Foundations of Computer Science* (Prague, Czech Republic, 1995), J. Wiedermann and P. Hájek (Eds.), Lect. Not. Comp. Sci. 969, Springer, pp. 37–49.

FG95 P. FRAIGNIAUD AND C. GAVOILLE. Memory requirement for universal routing schemes. In *14th Ann. Symp. on Principles of Distributed Computing* (Ottawa, Ont., Canada, 1995), ACM Press, pp. 223–230.

FGY93 M. FRANKLIN, Z. GALIL, AND M. YUNG. Eavesdropping games: A graph-theoretic approach to privacy in distributed systems. In *34th Symp. on Foundations of Computer Science* (Palo Alto, CA, USA, 1993), IEEE Comp. Soc. Press, pp. 670–679.

G

Gar94 J. A. GARAY. Reaching (and maintaining) agreement in the presence of mobile faults. In *8th Int. Workshop on Distributed Algorithms* (Terschelling, The Netherlands, 1994), G. Tel and P. M. B. Vitányi (Eds.), Lect. Not. Comp. Sci. 857, Springer-Verlag, pp. 253–264.

GP96 C. GAVOILLE AND S. PÉRENNÈS. Memory requirement for routing in distributed networks. In *15th Ann. Symp. on Principles of Distributed Computing* (Philadelphia, PA, USA, 1996), ACM Press. (to appear).

GP93 A. S. GOPAL AND K. J. PERRY. Unifying self-stabilization and fault-tolerance. In *12th Ann. Symp. on Principles of Distributed Computing* (Ithaca, NY, USA, 1993), ACM Press, pp. 195–206.

GHR90 M. G. GOUDA, R. R. HOWELL, AND L. E. ROSIER. The instability of self-stabilization. *Acta Informatica* **27**, 8 (1990), 697–724.

H

HV92 S. HALDAR AND K. VIDYASANKAR. Elegant constructions of atomic snapshot variables. Unpublished manuscript, 1992.

Her91 M. P. HERLIHY. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (1991), 124–149.

HS93 M. P. HERLIHY AND N. SHAVIT. The asynchronous computability theorem for t-resilient tasks. In *25th Ann. Symp. on Theory of Computing* (San Diego, CA, USA, 1993), ACM Press, pp. 111–120.

HW90 M. P. HERLIHY AND J. M. WING. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* **12**, 3 (1990), 463–492.

Hoe94a J.-H. HOEPMAN. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In *8th Int. Workshop on Distributed Algorithms* (Terschelling, The Netherlands, 1994), G. Tel and P. M. B. Vitányi (Eds.), Lect. Not. Comp. Sci. 857, Springer-Verlag, pp. 265–279.

Hoe94b J.-H. HOEPMAN. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. Tech. Rep. CS-R9423, Stichting Mathematisch Centrum (CWI), Amsterdam, 1994.

HPT95a J.-H. HOEPMAN, M. PAPATRIANTAFILOU, AND P. TSIGAS. Self-stabilization of wait-free shared memory objects. In *9th Int. Workshop on Distributed Algorithms* (Le Mont-Saint-Michel, France, 1995), J.-M. Hélary and M. Raynal (Eds.), Lect. Not. Comp. Sci. 972, Springer, pp. 273–287.

HPT95b J.-H. HOEPMAN, M. PAPATRIANTAFILOU, AND P. TSIGAS. Towards self-stabilizing wait-free shared memory objects. Tech. Rep. CS-R9514, Stichting Mathematisch Centrum (CWI), Amsterdam, 1995.

HT93a J.-H. HOEPMAN AND J. TROMP. Binary snapshots. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, 1993), A. Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 18–25.

HT93b J.-H. HOEPMAN AND J. TROMP. Binary snapshots. Tech. Rep. CS-R9319, Stichting Mathematisch Centrum (CWI), Amsterdam, 1993.

I

ICM+94 M. INOUE, W. CHEN, T. MASUZAWA, AND N. TOKURA. Linear-time snapshot using multi-writer multi-reader registers. In *8th Int. Workshop on Distributed Algorithms* (Terschelling, The Netherlands, 1994), G. Tel and P. M. B. Vitányi (Eds.), Lect. Not. Comp. Sci. 857, Springer-Verlag, pp. 130–140.

IJ93 A. ISRAELI AND M. JALFON. Uniform self-stabilizing ring orientation. *Information and Computation* **104**, 2 (1993), 175–196.

IS92 A. ISRAELI AND A. SHAHAM. Optimal multi-writer multi-reader atomic register. In *11th Ann. Symp. on Principles of Distributed Computing* (Vancouver, B.C., Canada, 1992), ACM Press, pp. 71–82.

ISS93 A. ISRAELI, A. SHAHAM, AND A. SHIRAZI. Linear-time snapshot protocols for unbalanced systems. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, 1993), A. Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 26–38.

ISS95 A. ISRAELI, A. SHAHAM, AND A. SHIRAZI. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory* **28** (1995), 469–486.

ILS95 G. ITKIS, C. LIN, AND J. SIMON. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *9th Int. Workshop on Distributed Algorithms* (Le Mont-Saint-Michel, France, 1995), J.-M. Hélary and M. Raynal (Eds.), Lect. Not. Comp. Sci. 972, Springer, pp. 288–302.

J

JCT92 P. JAYANTI, T. D. CHANDRA, AND S. TOUEG. Fault-tolerant wait-free shared objects. In *33rd Symp. on Foundations of Computer Science* (Pittsburgh, PA, USA, 1992), IEEE Comp. Soc. Press, pp. 157–166.

K

KP90 S. KATZ AND K. J. PERRY. Self-stabilizing extensions for message-passing systems. In *9th Ann. Symp. on Principles of Distributed Computing* (Quebec City, Qu., Canada, 1990), ACM Press, pp. 91–101.

KW91 J. KEPHART AND S. WHITE. Directed-graph epidemiological models of computer viruses. In *Proc. 1991 IEEE Computer Society Symp. on Research in Security and Privacy* (Oakland, CA, 1991), pp. 343–359.

KST91 L. M. KIROUSIS, P. SPIRAKIS, AND P. TSIGAS. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. In *5th Int. Workshop on Distributed Algorithms* (Delphi, Greece, 1991), S. Toueg, P. G. Spirakis, and L. Kirousis (Eds.), Lect. Not. Comp. Sci. 579, Springer-Verlag, pp. 229–241.

Kol65 A. N. KOLMOGOROV. Three approaches to the quantitative definition of information. *Problems Inform. Transmission* **1**, 1 (1965), 1–7.

KK96 E. KRANAKIS AND D. KRIZANC. Lower bounds for compact routing. In *13th Ann. Symp. on Theoretical Aspects of Computer Science* (Grenoble, France, 1996), C. Puech and R. Reischuk (Eds.), Lect. Not. Comp. Sci. 1046, Springer, pp. 529–540.

KKU95a E. KRANAKIS, D. KRIZANC, AND J. URRUTIA. Compact routing and shortest path information. In *Proc. 2nd International Colloquium on Structural Information and Communication Complexity* (1995), Carleton University Press.

KKU95b E. KRANAKIS, D. KRIZANC, AND J. URRUTIA. Lower bounds for compact shortest path routing schemes. Manuscript, 1995.

L

Lam86 L. LAMPORT. On interprocess communication. Part I: Basic formalism, part II: Algorithms. *Distributed Computing* **1**, 2 (1986), 77–101.

LSP82 L. LAMPORT, R. SHOSTAK, AND M. PEASE. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems* **4**, 3 (1982), 382–401.

LTV89 M. LI, J. TROMP, AND P. M. B. VITÁNYI. How to share concurrent wait-free variables. Tech. Rep. CS-R8916, Stichting Mathematisch Centrum (CWI), Amsterdam, 1989.

LV91 M. LI AND P. M. B. VITÁNYI. Optimality of wait-free atomic multiwriter variables. Tech. Rep. CS-R9128, Stichting Mathematisch Centrum (CWI), Amsterdam, 1991.

LV93 M. LI AND P. M. B. VITÁNYI. *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, New-York, 1993.

LAA87 M. C. LOUI AND H. H. ABU-AMARA. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research*, F. P. Preparata (Ed.), vol. 4. JAI Press, 1987, pp. 163–183.

LV93b N. A. LYNCH AND F. W. VAANDRAGER. Forward and backward simulations. Part I: Untimed systems. Tech. Rep. CS-R9313, Stichting Mathematisch Centrum (CWI), Amsterdam, 1993.

M

MA94 M. MOIR AND J. H. ANDERSON. Fast, long-lived renaming. In *8th Int. Workshop on Distributed Algorithms* (Terschelling, The Netherlands, 1994), G. Tel and P. M. B. Vitányi (Eds.), Lect. Not. Comp. Sci. 857, Springer-Verlag, pp. 141–155. To appear in Science of Computer Programming.

MG96 M. MOIR AND J. A. GARAY. Fast, long-lived renaming improved and simplified. In *15th Ann. Symp. on Principles of Distributed Computing* (Philadelphia, PA, USA, 1996), ACM Press. (to appear).

Mul89 S. MULLENDER. *Distributed Systems*. ACM Press, New York, 1989.

Mul93 S. MULLENDER. *Distributed Systems (2nd edition)*. ACM Press, New York, 1993.

OY91 R. OSTROVSKY AND M. YUNG. How to withstand mobile virus attacks. In *10th Ann. Symp. on Principles of Distributed Computing* (Montreal, Qu., Canada, 1991), ACM Press, pp. 51–59.

P

PPT+94 A. PANCONESI, M. PAPATRIANTAFILOU, P. TSIGAS, AND P. M. B. VITÁNYI. Randomized wait-free naming. In *5th Int. Symp. on Algorithms and Computation* (Beijing, P. R. China, 1994), D.-Z. Du and X.-S. Zhang (Eds.), Lect. Not. Comp. Sci. 834, pp. 83–91.

PT94 M. PAPATRIANTAFILOU AND P. TSIGAS. On self-stabilizing wait-free clock synchronization. In *4th Scandinavian Workshop on Algorithm Theory* (Århus, Denmark, 1994), Lect. Not. Comp. Sci. 824, Springer Verlag, pp. 267–277.

PSL80 M. PEASE, R. SHOSTAK, AND L. LAMPORT. Reaching agreement in the presence of faults. *Journal of the ACM* **27**, 2 (1980), 228–234.

PU89 D. PELEG AND E. UPFAL. A trade-off between space and efficiency for routing tables. *Journal of the ACM* **36**, 3 (1989), 510–530.

PB87 G. L. PETERSON AND J. E. BURNS. Concurrent reading while writing II: The multi-writer case. In *28th Symp. on Foundations of Computer Science* (Los Angeles, CA, USA, 1987), IEEE Comp. Soc. Press, pp. 383–392.

PF77 G. L. PETERSON AND M. J. FISCHER. Economical solutions for the critical section problem in a distributed system. In *9th Ann. Symp. on the Theory of Computing* (Boulder, CO, USA, 1977), ACM Press, pp. 91–97.

R

Rei85 R. REISCHUK. A new solution for the byzantine generals problem. *Information and Computation* **64** (1985), 23–42.

S

San84 N. SANTORO. Sense of direction, topological awareness and communication complexity. *ACM SIGACT News* **16**, 2 (1984), 50–56.

Sch93 M. SCHNEIDER. Self-stabilization. *ACM Computing Surveys* **25**, 1 (1993), 45–67.

Sha79 A. SHAMIR. How to share a secret. *Communications of the ACM* **22**, 11 (1979), 612–613.

SL95 D. E. SHASHA AND C. A. LAZERE. *Out Of Their Minds.* Copernicus (Springer-Verlag), New York, 1995.

SP87 V. SYROTIUK AND J. PACHL. A distributed ring orientation algorithm. In *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, The Netherlands, 1987), J. van Leeuwen (Ed.), Lect. Not. Comp. Sci. 312, Springer-Verlag, pp. 332–336.

T

Tch81 M. TCHUENTE. Sur l'auto-stabilisation dans un réseau d'ordinateurs. *RAIRO Informatique Théorique* **15**, 1 (1981), 47–66.

Tel94 G. TEL. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

V

VA86 P. M. B. VITÁNYI AND B. AWERBUCH. Atomic shared register access by asynchronous hardware. In *27th Symp. on Foundations of Computer Science* (Toronto, Ont., Canada, 1986), IEEE Comp. Soc. Press, pp. 233–243.

She's full of disorders, depends what you're used to
She's talking of trances of truncheons in battle
Of bruises from bottles that never get better
Bad baby bitching she screams at the door
Hammer in hand her head to the floor
Marlilyn Moore

SONIC YOUTH
Marilyn Moore

E Index

A

T

U

V

W

Colofon

Vormgeving en omslag: Jaap Henk Hoepman

Opmaak en druk: Ponsen & Looijen B.V., Wageningen

Lettertypen: Lucida, Futura, en Helvetica

Produced using LaTeX 2ε. Style files available on request from the author (jhh@cwi.nl).

Hör mich nur atmen
Doch das beweist nichts

Inmitten meiner Kreise
Doch deren Mitte bin ich nicht

Regungslos
Wartend
Wartend

Wenn du kommst, kommst du mit Licht
Du kommst strahlend

Zehrst meinen Schatten auf
Zählst meine Kerben
Und schlägst mich auf
Öffnest mein versteck
Und liest mich laut
Damit auch ich
Mich
Hören kann

Wenn du gehst, fragst du:
Wer von uns beiden glaubst du?
Ist der Geliebte? Wer von uns
Ist der Geliebte?

EINSTÜRZENDE NEUBAUTEN
Fiat Lux