Distrib. Comput. (2003) 16: 121–163
Digital Object Identifier (DOI) 10.1007/s00446-003-0091-y

DISTRIBUTED COMPUTING

Hundreds of impossibility results for distributed computing

Faith Fich[1], **Eric Ruppert**[2]

[1] Department of Computer Science, University of Toronto, Toronto, Canada (e-mail: fich@cs.utoronto.ca)
[2] Department of Computer Science, York University, Toronto, Canada (e-mail: ruppert@cs.yorku.ca)

Received: September 2001 / Accepted: February 2003

Abstract. We survey results from distributed computing that show tasks to be impossible, either outright or within given resource bounds, in various models. The parameters of the models considered include synchrony, fault-tolerance, different communication media, and randomization. The resource bounds refer to time, space and message complexity. These results are useful in understanding the inherent difficulty of individual problems and in studying the power of different models of distributed computing. There is a strong emphasis in our presentation on explaining the wide variety of techniques that are used to obtain the results described.

Keywords: Distributed computing – Impossibility results – Complexity lower bounds

Contents

1 Introduction 121
2 Models 122
3 Distributed problems 124
3.1 Consensus 124
3.2 Approximate agreement 124
3.3 Other agreement problems 125
3.4 Resource allocation 125
4 Proving impossibility results 126
4.1 Proof techniques 127
5 Unsolvability of consensus and other problems 129
5.1 Asynchronous message-passing models 129
5.2 Asynchronous shared-memory models 131
5.3 Bounds on the number of faults in synchronous message-passing models 132
5.4 Anonymous systems 133
5.5 Clock synchronization 133
5.6 Other problems 134
6 Relationships between models 135
6.1 System size 135
6.2 Timing assumptions 136
6.3 Consistency conditions 137
6.4 Fault models 138
6.5 Other separations between models 139
7 Deciding when problems are solvable 139
7.1 Undecidability 139
7.2 Decidability of consensus numbers 140
7.3 Characterizing solvable tasks 140
8 Topology 141
8.1 Simplicial complexes 141
8.2 Set consensus results 142
8.3 The Asynchronous Computability Theorem 142
8.4 Solving tasks with other types 143
9 Robustness 143
9.1 Non-robustness results 144
9.2 Robustness results 144
10 Space complexity lower bounds 145
11 Time complexity lower bounds 146
11.1 Agreement problems 146
11.2 Approximate agreement 147
11.3 The complexity of universal constructions 148
11.4 Counting 148
11.5 Other time complexity lower bounds 148
12 Message and bit complexity lower bounds 149
12.1 Computation in a ring 149
12.2 Other network topologies 151
13 Randomized computation 153
13.1 Unsolvability results 153
13.2 Complexity lower bounds 154
14 Conclusions 155
References 155
Index 162

1 Introduction

What can be computed in a distributed system? This is a very broad question. There are many different models of distributed systems that reflect different system architectures and different levels of fault-tolerance. Unlike the situation in sequential models of computation, small changes in the model of a distributed system can radically alter the class of problems that can be solved. Another important goal in the theory of distributed computing is to understand how efficiently a distributed system can solve those problems that are solvable. There are a variety of resources to consider, including time, contention, and the number and sizes of messages and shared data structures.

This paper discusses unsolvability results, which show that problems are unsolvable in certain environments, and lower bound results, which show that problems cannot be solved when insufficient resources are available. We refer to these two types of theorems collectively as impossibility results. A comprehensive survey of impossibility results in distributed computing would require an entire book; consequently, we have focused on the models and problems that we feel are most central to distributed computing. Our aim is to give you the flavour of this research area and an outline of the most important techniques that have been used. There are, however, some important topics that we decided lie outside the scope of this survey, including self-stabilization [121], failure detectors [89], and uses of cryptography in distributed computing [150].

Why are impossibility results important for distributed computing? They help us understand the nature of distributed computing: what makes certain problems hard, what makes a model powerful, and how different models compare. They tell us when to stop looking for better solutions or, at least, which approaches will not work. If we have a problem that we need to solve, despite an impossibility result, the impossibility proof may indicate ways to adjust the problem specification or the modelling of the environment to allow reasonable solutions. Impossibility results have also influenced real systems, for example, the design of network file systems [84], the architecture of fault tolerant systems for safety-critical applications [275], the design of programming languages [59], the specifications of group membership services [97], and the definition and study of failure detectors and systems based on them [88]. Finally, trying to prove impossibility results can suggest new and different algorithms, especially when attempts to prove impossibility fail. As John Cage wrote, "If someone says 'can't', that shows you what to do" [82].

We begin in Sects. 2 and 3 with brief descriptions of the models, terminology, and problems that are discussed throughout the paper. Section 4 discusses how to approach impossibility results and gives an overview of the major proof techniques for impossibility results in distributed computing. The rest of the paper presents a wide variety of results. Section 5 describes unsolvability results for the consensus problem and some similar process-coordination tasks. The use of impossibility results to study relationships between different models is addressed in Sect. 6. A systematic approach to studying computability for distributed systems is to characterize the models that can solve a particular problem. Alternatively, one can characterize the set of problems solvable in a given model. Results of both these types are described in Sect. 7. Further characterizations of problems solvable in specific models appear in Sect. 8, which discusses applications of topology to proving impossibility results in distributed computing. Section 9 examines the question of whether weak shared object types can become more powerful when they are used in combination with other weak types. The next three sections consider complexity results. Each focuses on a different complexity measure: space, time, and the number of messages. Section 13 discusses impossibility results for randomized algorithms. An index of problems and proof techniques appears at the end of the paper.

This survey builds on Lynch's excellent survey paper, "A Hundred Impossibility Proofs for Distributed Computing" [233], which covers results up to 1989. A shorter, preliminary version of our survey, emphasizing results from 1990 onwards, appeared in [137].

2 Models

There are a number of very good descriptions of distributed models of computation, including motivation and formal definitions [47,223,234]. Here, we shall only briefly mention some aspects of these models which are necessary for the results we present.

A distributed system consists of a collection of n processes that run concurrently. Each process executes a sequential algorithm and can communicate with other processes.

There are different ways processes can communicate. In **message-passing** models, processes send messages to one another via communication channels. This is modelled by a graph, with processes represented by nodes, bidirectional channels represented by undirected edges, and unidirectional channels represented by directed edges. A correct channel behaves as a (FIFO) queue, with the sender enqueuing its messages and the receiver dequeueing them. If the queue is empty, the receiver gets a special empty queue message. If message delivery is not instantaneous, the messages in the queue will not be immediately available to the receiver.

In **shared-memory** models, processes communicate by performing operations on shared data structures, called **objects**, of various types. The `typewriter` font is used to denote object types. Each **type** describes the set of possible states of an object of that type, the set of operations that can be performed on the object, and the responses the object can return. At any time, an object has a **state** and, when a process performs an operation on the object, the object can change to a new state and return a response to the process. For example, a `stack` object stores a sequence of values in its state and supports the operations push and pop. A basic type of object is the `register`, which stores a value that can be read or written by all processes. A `single-writer register` is a restricted type of `register` to which only a single, fixed process can write. Similarly, only one fixed process can read from a `single-reader register`. A `snapshot` object stores an array of values. Processes can scan the entire array to learn the value of every element in a single, atomic operation and can update the value of individual elements. For each element in a `single-writer snapshot` object, there is only one fixed process that can update it. An important class of object types are the **read-modify-write (RMW) types** [213]. A RMW operation updates the state of the object by applying some function, and returns the old value of the state. For example, the test&set operation is a RMW operation that applies the function $f(x) = 1$, and fetch&add applies the function $f(x) = x + 1$. Other RMW operations include read and compare&swap. A RMW type is one where all permitted operations have this form.

Consistency conditions describe how objects behave when accessed by several concurrent operations. One example is **linearizability** [181], which requires that operations appear to happen instantaneously at distinct points in time, although they may actually run concurrently. Furthermore, the order in which the operations appear to happen must be consistent with real time: if an operation terminates before another operation begins, then it will occur earlier in the order. Many other consistency conditions have been studied [13,16,38,149,183, 186,217,255].

A linearizable object type is **deterministic** if the outcome (i.e. the response and new state) of each operation with specified input parameters is uniquely determined by the object's current state. **Non-deterministic** types may have more than one possible outcome for an operation in some states. Algorithms that use non-deterministic objects must work correctly for all possible outcomes.

In **randomized** algorithms, a process may have many choices for its next step, but the choice is made according to some probability distribution. Generally, for randomized algorithms, termination is required only with high probability and one considers worst-case expected time, rather than worst-case time. Non-determinism in the shared objects (or other parts of the computing environment) makes problems harder to solve, while allowing randomization in the algorithm can make problems easier to solve.

Ordinarily, it is assumed that each process has a unique name, called its **identifier**. **Comparison-based** algorithms only use identifiers by comparing their values. Other algorithms may use process identifiers to index shared arrays or control which pieces of code are executed. An **anonymous** system is one in which processes have identical code and do not have identifiers.

When a system is **synchronous**, all processes take steps at exactly the same speed. If the speed of each process may vary arbitrarily during an execution, the system is **asynchronous**. Synchronous computation proceeds in **synchronous rounds**. In each round, every process takes exactly one step. Synchronous shared-memory systems have been studied extensively using the parallel random-access machine model (see [269]) and mostly lie outside the scope of this survey. In synchronous message-passing models, messages sent in one round are available to be received in the next round. Typically, in one step, a process dequeues one message from each of its incoming channels and enqueues at most one message on each of its outgoing channels. Asynchronous computation is generally modelled by an adversarial scheduler that chooses the order in which processes take steps. In one step, a process can either send a single message, receive a single message, or access a single shared object. Asynchronous algorithms must work correctly for every legal schedule. In **partially synchronous** or **semi-synchronous** models, processes may also run at different speeds, but there are bounds on the relative speeds of processes (and on message-delivery times for message-passing systems).

In synchronous systems, time is measured by the number of rounds. In asynchronous and partially synchronous systems, there are several ways to measure time [47,234,257]. The **step complexity** counts the maximum number of steps taken by a single process. **Work** counts the total number of steps taken by all processes. Asynchronous computation can also be divided into **asynchronous rounds**, where a round ends as soon as every process has taken at least one step since the beginning of the round. The round complexity can be less than the step complexity, because some processes may take multiple steps per round.

In message-passing systems, the total number of messages transmitted during an execution of an algorithm is an important measure of the algorithm's complexity. This is called its **message complexity**. The **bit complexity** counts the total number of bits in these messages. Some algorithms with small message complexity are, in fact, very inefficient, because they send very long messages.

Many different kinds of faults are considered in distributed systems. Processes may fail and perhaps recover, their states can become corrupted, or they can behave arbitrarily. The latter kind of fault is called an **arbitrary process fault** or a **Byzantine fault**. An algorithm in a model with arbitrary process faults must work correctly no matter how faulty processes behave. This type of fault is useful for modelling malicious attacks or situations in which the faults that can occur are difficult to characterize. Arbitrary process faults are not usually considered for shared-memory systems, since a faulty process can corrupt the entire shared memory. A **crash failure** is when a process fails by halting permanently.

In an f**-faulty** system, there are at most f faulty processes, so an f-faulty system is f'-faulty, for all $f' \geq f$. An algorithm that works correctly in an f-faulty system, i.e. can tolerate up to f process faults, is called f**-resilient**. Thus an f-resilient algorithm is f'-resilient, for all $f' \leq f$. A **wait-free** algorithm ensures that every non-faulty process will correctly complete its task, taking only a finite number of steps, even if any number of other processes crash. Thus, in a system of n processes that are subject only to crash failures, wait-freedom is the same as $(n-1)$-resilience. For randomized algorithms, wait-freedom means that the expected number of steps needed by a process to complete its task is finite, regardless of the number of failures.

Communication channels can also fail in many different ways: They can crash or they can lose, delay, or duplicate messages, or deliver them out of order. One way to model a communication channel that can lose messages is to consider that the process at one endpoint fails to send or receive certain messages that it is supposed to. Such a process is said to have an **omission fault**. Another way to model message losses in synchronous message-passing systems is to allow at most a certain number messages to be lost each round, but the communication channels on which these losses occur may change from round to round. These are called **dynamic omission faults**. Finally, shared objects can fail to respond (i.e. crash), have their states corrupted, or behave contrary to their type specifications.

Throughout this paper, unless we state otherwise, we assume that all objects deterministic, linearizable, and non-faulty, all communication channels are non-faulty, all algorithms are deterministic, and processes only have crash failures.

Processes in a shared-memory distributed system are provided with some basic primitive objects that they can use to communicate with one another. Any other types of shared objects that a programmer wishes to use must be implemented from those primitives. Thus, one of the fundamental issues in the study of distributed computing is determining the circumstances under which such implementations are possible. An **implementation** provides, for each process, a programme that it can execute to perform each possible operation on an object of the type being implemented. Before an execution of this programme terminates, it should produce a response to the operation. In any legal execution where processes concurrently execute the programmes for various operations, the responses provided should satisfy the desired consistency conditions. For example, consider a linearizable implementation of an object. Then, for any execution, there exists a linear order of the simulated operations so that the correct responses for these operations are the same as in the execution. Furthermore, if the programme for one operation has finished executing before the execution of the programme for another operation begins, the latter operation comes later in the linear order. (See [234] for more formal definitions of implementations.) Examples of implementations can also be found in [47], including a series of wait-free implementations to construct `snapshot` objects from `single-writer, single-reader registers`. This means that the different types of `registers` and `snapshot` objects are equivalent in terms of the wait-free solvability of problems using these objects.

A related notion is that of a **simulation** of model A by model B. Intuitively, such a simulation describes how any algorithm designed for a collection of processes in model A can be adapted so that a collection of processes can execute it in model B. To distinguish the simulating processes from the simulated ones, we refer to the processes of model A as **threads** and the processes of model B as **simulators** whenever we describe simulations. The two models may be quite different. For example, they might use different communication media, have different synchrony assumptions or permit different numbers of faults. Usually, each simulator simulates the actions of one thread, but this is not always the case. The simulation should specify the programme a simulator must execute to simulate a step of a thread, and also how the simulator can determine the outcome of the simulated step. In cases where a simulator simulates several threads, the simulation should also describe how a simulator chooses which thread's step it should simulate next. Suppose the simulation is used to simulate an algorithm Π designed for model A. For any execution of the simulation that is legal in model B, there must exist a corresponding execution of Π that is legal in model A such that the response to each step of each thread is identical to the response computed by the simulation.

3 Distributed problems

In this section, we define a number of important and well-studied problems in the theory of distributed computing. Impossibility results concerning these problems will be presented in subsequent sections.

3.1 Consensus

The **consensus** problem is the most thoroughly investigated problem in distributed computing and it is used as a primitive building block for solutions to many distributed problems. Consensus is an example of a **decision task**, in which each process gets a private input value from some set and must eventually terminate after having produced an output value. The task specification describes which output values are legal for given input values. For consensus, there are two correctness properties that must be satisfied:

Agreement: the output values of all processes are identical, and

Validity: the output value of each process is the input value of some process.

In models where arbitrary faults may occur, these properties are weakened and apply only to correct processes, since one cannot guarantee anything about the behaviour of faulty processes. The definition of the consensus problem was carefully designed so that it is extremely simple to state, yet captures much of the difficulty of designing algorithms that allow processes to solve problems cooperatively. In the **binary consensus** problem, all input values come from the set $\{0, 1\}$.

Consensus is an excellent problem to use for a systematic study of solvability, since Herlihy [169] showed that it is universal: a shared-memory system equipped with `registers` and objects that can solve wait-free consensus can implement any other object type in a wait-free manner.

Object types can be classified according to the ability of an asynchronous shared-memory distributed system to solve consensus using them. Specifically, the **consensus number** $\text{cons}(\mathcal{T})$ **of a set of object types** $\mathcal{T}$ is the maximum number of processes for which wait-free consensus can be solved using any number of objects in $\mathcal{T}$ and `registers` [169,192]. The **consensus number** $\text{cons}(\mathtt{T})$ **of an object type** $\mathtt{T}$ is $\text{cons}(\{\mathtt{T}\})$. Suppose $\text{cons}(\mathtt{T}) < \text{cons}(\mathtt{T}')$. It follows from the definition of consensus numbers that $\mathtt{T}'$ cannot be implemented in a wait-free manner from objects of type $\mathtt{T}$ and `registers` (in a system of more than $\text{cons}(\mathtt{T})$ processes). On the other hand, $\mathtt{T}$ has a wait-free implementation from objects of type $\mathtt{T}'$ and registers, for up to $\text{cons}(\mathtt{T}')$ processes, by Herlihy's universality result. Thus, this classification, called the **consensus hierarchy**, gives a great deal of information about the relationships between different models of asynchronous, shared-memory systems. However, the consensus number of an object type does not completely describe the power of a shared-memory model that provides objects of that type and `registers`. For example, there are object types $\mathtt{T}$ and $\mathtt{T}'$ with consensus numbers 1 and n, respectively, such that 2-set consensus (defined in Sect. 3.3) has a wait-free solution for $2n+1$ processes using objects of type $\mathtt{T}$ and `registers`, but not using only objects of type $\mathtt{T}'$ and `registers` [267]. Thus the power of two types $\mathtt{T}$ and $\mathtt{T}'$ can be incomparable in a system with sufficiently many processes.

In **randomized consensus**, both the agreement and validity properties must be satisfied, but the termination condition is weaker: for any schedule, the expected number of steps taken by each non-faulty process must be finite. This version of consensus can be solved in a wait-free manner by a randomized algorithm using only `registers` in an asynchronous system [98,1]. Thus, for randomized computation, the consensus hierarchy collapses into a single level.

3.2 Approximate agreement

Allowing processes to disagree by a small amount results in a significantly easier problem. In the **approximate agreement** problem, input and output values are real numbers. There is a tolerance parameter, ϵ, known by all processes. The agreement and validity conditions of consensus are replaced by

ϵ-Agreement: the output values of all processes are within ϵ of one another, and

Validity: the output value of each process must lie between the minimum and maximum input values.

In the case of arbitrary faults, the conditions constrain only the non-faulty processes and the validity condition is strengthened to require all outputs to be within the range of inputs of correct processes, to ensure that malicious processes cannot cause arbitrary outputs. The **convergence ratio** ρ of an approximate agreement algorithm is the worst-case ratio of the size of the range of the output values (of correct processes) to the size of the range of the input values. If the size of the range of the input values is R, then $\rho \leq \epsilon/R$.

Approximate agreement arises in algorithms for **clock synchronization**, where processes are assumed to have separate physical clocks that can start at different times or can run at different rates. The object of clock synchronization is for processes to compute adjustments to their physical clocks so that the adjusted clocks of non-faulty processes remain close to one another and within the range of the physical clocks.

3.3 Other agreement problems

The **terminating reliable broadcast** problem is a version of consensus where only one process, the **sender**, has an input which it must communicate to all other processes in the system. The agreement condition is the same as for consensus, but the validity condition only requires that the output values of non-faulty processes must be the sender's input value, if the sender is non-faulty. This problem is also called **Byzantine agreement**. It is typically studied in synchronous systems when processes can have arbitrary faults, instead of just crash failures. Relationships between consensus and terminating reliable broadcast in various message-passing models are discussed by Hadzilacos and Toueg [163]. For example, in synchronous models, the terminating reliable broadcast problem can be reduced to consensus by having the sender send its input to all other processes in the first round.

Restricted versions of the terminating reliable broadcast and consensus problems in which all processes must produce their output values in the same round are called **simultaneous terminating reliable broadcast** and **simultaneous consensus**, respectively. Simultaneous consensus is also called **coordinated attack**. These problems are well-defined only for models in which processes run synchronously.

A common assumption when solving problems in synchronous systems is that all processes start at the same time. One problem that addresses this assumption is **wakeup** [142], where some number of processes must detect when sufficiently many processes have begun taking steps. Another is the **distributed firing squad** problem, which requires that all processes execute a special "fire" command in the same round, even though they may start at different rounds.

There are close connections between consensus and other problems. One such problem is **leader election**, where there are no inputs, exactly one process (called the **leader**) must output 1, and all other processes must output 0. If a system can solve consensus, then it can also solve leader election: each process uses its own unique identifier as an input to consensus, and processes agree on the identity of the leader. Conversely, if one requires that a process inform all others of its identity before proclaiming itself a leader, one can solve consensus by using the leader's input value as the common output value. Thus, impossibility results for the two problems are closely related.

The k**-set consensus** problem, introduced by Chaudhuri [92], is similar to the consensus problem, but relaxes the agreement property. Instead of requiring that all output values are identical, it requires that the set of output values produced has cardinality at most k. Thus, consensus is the special case of k-set consensus where $k = 1$.

Many variants of the consensus and set consensus problems, with slightly different agreement and validity properties, have been studied [109,138,140,163,218,234,254]. One example is the **commit** problem. It is a version of binary consensus, where the validity condition requires that, if any input value is 0, the output value of each process must be 0 and, if all input values are 1 and there are no faults, the output value of each process must be 1. This problem arises when maintaining consistency among several copies of a database as updates occur. In this case, the output value 1 denotes the *commit* to an update and the output value 0 denotes that the update is to be *aborted*. The specifications of this problem allow any process to abort an update unilaterally.

In the **choice coordination** problem [265], each process must choose the same shared option from among k alternatives. Each alternative has an associated shared object, but there are no global names for the alternatives (or objects): each process has its own local names for them. In the **group membership** problem [97], processes must maintain a consistent view of a set containing process identifiers as processes make requests to add or remove their own identifiers.

3.4 Resource allocation

An extremely well-studied distributed computing problem is **mutual exclusion**, introduced by Dijkstra [111]. It is an abstraction of the problem of sharing a resource, for example, a printer, to which processes need temporary exclusive access. A process which has this access is said to be in the **critical section** of its code. Processes may repeatedly compete for permission to access the resource. A correct algorithm ensures that two or more processes are never simultaneously in their critical sections. There is also a liveness property,

Deadlock Freedom: if some process wants the resource and no process has permission to access it, then, eventually, some process will be given permission.

Various fairness conditions have also been considered, for example,

Lockout Freedom: if some process wants the resource, then, eventually, it will be given permission.

The k**-assignment** problem [80] is a generalization of mutual exclusion in which there are $k < n$ identical, named resources that may be requested by the n processes. Requesters must determine the name of one of the resources in such a way that no two processes choose the same resource at the same time.

The **dining philosophers** problem [112] is another related resource allocation problem: the processes are arranged in a ring, each pair of adjacent processes share a resource, and each process sometimes requires simultaneous exclusive access to both the resources it shares. Variants of this problem where a resource can be shared by more than two processes and processes may have different sets of required resources have also been considered [55,91,203].

In the **renaming** problem, each participating process is initially given a unique identifier from a large name space. They must all select unique identifiers from a smaller name space. In **order-preserving renaming**, the two name spaces are ordered and the identifiers of the processes must have the same relative order in each. The renaming problem can be applied to improve the efficiency of algorithms: If an algorithm's complexity depends on the size of the name space, one can use renaming to reduce this size before the algorithm is executed.

Sometimes, the processes, themselves, are the resources that must be allocated. In the **task assignment** problem, each of the tasks in some set must be chosen by at least one process, with each process choosing at most one task. A related problem is **write-all**, where a set of `registers`, each initially 0, must each have value 1 written to it by one or more of the n processes. This is a representative instance of the more general problem of ensuring that a set of idempotent tasks are all performed.

4 Proving impossibility results

This section discusses issues that arise when proving impossibility results and a variety of proof techniques. In subsequent sections, we give many examples of results proved using these techniques and, in some cases, explain them more fully in the context of particular examples.

To prove that no algorithm can solve a particular problem or solve it efficiently, it is necessary to define the model of computation and the class of allowable algorithms. These definitions will be used repeatedly in proofs of impossibility. It will be apparent from many of the results in this survey that the difficulty of many problems depends on precisely which model is being used.

Without a clear and precise definition of the model, ambiguities and subtle errors can arise. The use of formal models forces people to make their assumptions explicit. This helps to expose subtle differences in assumptions, which often lead to many variations of models, with corresponding, different results. In turn, such results help us to understand our models better and to converge on good sets of assumptions [97]. In fact, some of the early papers containing impossibility results for distributed computing included the formulation of correctness conditions and directly led to the development of formal models for distributed computing [4,79,108,235]. There is another benefit to carefully stating the assumptions about the model that are necessary for the impossibility proof to work: once the assumptions are identified, one can look for algorithms that beat the impossibility result by operating in a model where one or more of the assumptions do not hold.

Models should be simple so as to be feasible to work with, interesting to work on, and applicable to a variety of real implementations. In choosing models, one should follow the dictum of Ludwig Mies van der Rohe: "Less is more" [244]. When trying to establish an impossibility result, it is often helpful to simplify the model as much as possible, while ensuring that the simplifications have not weakened the model, and then prove impossibility in the streamlined model. Showing that the simplified model can simulate the original one is a good way to show that the model has not been weakened. Furthermore, impossibility results proved for strong models are better than the same results proved for weak models. This is because an algorithm designed for one model automatically works in a stronger model, so an impossibility result for a stronger model automatically applies to a weaker model.

Impossibility results are always proved for a class of algorithms. A lower bound or unsolvability result for a class trivially applies to any subclass. Sometimes, the proofs are easier for a restricted class of algorithms, for example, comparison-based algorithms. Such proofs can help our understanding of the problem and provide insight for more general results. If algorithms in a restricted class can simulate more general algorithms, then impossibility results proved for this restricted class also imply impossibility results for the more general class. For example, it is sometimes helpful to assume that each process remembers its entire history and sends this information whenever it communicates with another process or writes to a `register`. Such algorithms are called **full-information algorithms**. A full-information algorithm can simulate an algorithm that is not of this form by having processes ignore some of the information they receive. Showing that a problem has no full-information algorithm also automatically implies that it has no algorithm that uses more realistic resources, such as bounded message lengths or bounded size `registers`, and a limited amount of local computation in each step.

Precise problem statements are just as important as precise descriptions of the model. Elegant, simple problem statements are much easier to use in impossibility proofs. Complex, specialized problems, even when they arise from real systems, are unlikely to be good choices. Instead, one needs to extract simple prototype problems, prove impossibility results about them, and then use (often simple) reductions from them to obtain corresponding impossibility results for the original complex problem. Results about well-chosen problems are more likely to be fundamental.

Arriving at a good statement of a problem can be an iterative process. It is easy to make the problem statement too strong, in which case impossibility results might hold for trivial reasons. (For example, consider a problem that requires that every process requesting exclusive access to a shared resource eventually gets it, but does not say that, whenever a process has a resource, it must eventually release the resource.) It is also easy to make the problem statement too weak, in which case, trivial counter-example algorithms can arise. (For example, requiring all processes to output the same value is easy if no other constraints are imposed: they can always simply output 0.) This iterative process may eventually lead to an interesting problem statement and a corresponding impossibility result or algorithm. Assumptions that are not needed can be eliminated, so that the proof is based on the weakest possible set of requirements.

Papers will often present algorithms for a difficult version of a problem using a weak model of computation, and then prove matching complexity lower bounds for an easier version of the problem in a stronger model of computation. Results stated in this way show that the complexity of the problem is insensitive to small changes in the model or problem statement. They can also point out aspects of the problems that are not important and features of the models that do not affect the solution.

When faced with the question of whether or not a problem is solvable or efficiently solvable in a particular model, one usually begins by trying to devise an algorithm. If this fails, one might start to look for an impossibility proof by trying to find a reduction from some other problem that is already known to be hard. This paper gives you many candidates. Another simple approach to proving impossibility is to show that the model of computation being considered is weaker than some model in which the problem is known to be hard. These approaches usually provide some intuition about what makes the problem difficult. One can then alternate between working on an algorithm and trying to prove impossibility. If there are difficulties that arise persistently, causing candidate algorithms to fail or perform inefficiently, it might be possible to produce an impossibility result by showing that these difficulties cannot be avoided by *any* algorithm. Similarly, if the same obstacle foils all attempts at an impossibility proof, it may suggest an algorithm that can exploit this loophole.

4.1 Proof techniques

There is one fundamental idea underlying all of the proofs of impossibility results for distributed computing: "the limitations imposed by local knowledge" [233]. In order to solve distributed computing problems, processes have to learn about the rest of the system. We get unsolvability results and lower bounds by showing this is impossible, either outright or with a limitation on resources.

A process may have incomplete knowledge of the system because it does not initially know the inputs of other processes or because of asynchrony or faults. The process may not be able to learn about other parts of the system quickly because of the distance information must travel or limitations on the communication medium, such as the size of the shared memory.

Indistinguishability is one way of formalizing this lack of knowledge. An important observation is that, if processes see the same thing in two executions, they will behave the same way in both. A **configuration** describes a distributed system at some point during the execution of an algorithm: it consists of the states of all processes and the state of the environment (i.e. the messages in transit for a message-passing system, or the states of all shared objects for a shared-memory system). Two configurations are said to be **indistinguishable** to a process if its local state is the same both configurations and the information that it can access from the communication medium is the same in both configurations. When a sequence of steps can be performed by a set S of processes starting from a particular configuration, the sequence can also be performed starting from any other configuration that is indistinguishable to each process in S. Moreover, the two resulting configurations are indistinguishable to every process in S. If we can use indistinguishability to show that, for any algorithm, some process cannot distinguish two executions for which it must produce different outputs, then we can conclude that no correct algorithm exists.

One way to construct two indistinguishable executions in an asynchronous message-passing system is by **stretching**. Starting with a carefully chosen execution, the idea is to speed up some processes, slow down others, and adjust message delivery times so that each process performs the same steps in the same order. In message-passing systems where all processes run at the same rate, **shifting** can be used instead: The operations of some processes are moved earlier, the operations of other processes are moved later, and message delivery times are adjusted appropriately.

In a distributed system, there are often many executions that can arise and one algorithm must work correctly for all of them. Thus, for an unsolvability result, it suffices to construct one incorrect execution. Similarly, for worst-case lower bounds, it suffices to construct one execution which uses a lot of resources. We think of these executions as being constructed by an **adversary**. For example, in an asynchronous system, we use an adversarial scheduler to choose the order in which processes take steps. An adversary can also choose input values and decide when and where faults occur. The power of the adversary may be limited by the definition of the model. For example, in an asynchronous system, a fairness condition might be imposed on the adversary requiring it to allocate an infinite number of steps to every non-faulty process in every non-terminating execution. In a partially synchronous system, the adversary must adhere to the bounds on the speeds of the processes.

An impossibility result obtained using a restricted adversary (i.e. one that can construct only a limited set of executions) automatically implies the same result for more powerful adversaries. For this reason, it is better to prove a result using a restricted adversary. Furthermore, an appropriately chosen weak adversary can clarify which aspects of the problem or model make the problem difficult. The lower bound proofs for simultaneous consensus discussed in Sect. 11.1 are good examples. Impossibility results with restricted adversaries may also be easier to understand and have more elegant proofs (because there are fewer cases to consider). The key to such proofs is coming up with the right adversary. One must discard any unnecessary complications while ensuring that the adversary is still strong enough to prove the desired result. For example, see the unified unsolvability results for consensus in different models in Sect. 5.2.

In situations where there is a bound on the number of possible states of the shared memory, a bad execution can sometimes be found by considering a large number of different reachable configurations. The **pigeonhole principle** can be used to show that two of these configurations are indistinguishable to some group of processes. Then, an adversary can construct a sequence of steps of these processes starting at one of these configurations, but which violates a correctness condition when started at the other. More general **combinatorial arguments** can also be used to prove the existence of bad executions. For example, one might count the number of possible executions to show that two of them are indistinguishable if the space or time used is too small.

Sometimes an adversary can construct executions in which a sequence of steps by a set of processes can be hidden from the other processes, i.e. removing these steps from the execution yields a final configuration that is indistinguishable to the other processes. For example, consider an execution in an asynchronous system where processes communicate through shared `registers`, such that, from some configuration C, all steps are performed by processes from a set P. If, immediately after

C, the execution performs a write to each `register`, then steps performed immediately before C by processes not in P will be hidden from the processes in P. More generally, a process **covers** an object in a configuration if the process will write to the object (or perform an operation that will obliterate any information previously stored in the object) whenever it is next allocated a step by the scheduler. In **covering arguments**, introduced by Burns and Lynch [77], an adversary carefully constructs an execution ending in a configuration where all the shared objects are covered. The adversary extends this execution with steps by certain processes. Then one process covering each object performs its next step. These operations hide the extension of the execution from other processes. Specifically, the resulting configuration and the configuration obtained by performing these operations immediately after the execution (without the extension) are indistinguishable to the other processes. The adversary can then use this fact to construct a bad execution from one of these configurations.

In anonymous shared-memory systems where processes communicate using `registers`, an adversary can treat a group of processes with the same input values as **clones**, running them together in lock step. These processes always read the same values from the same `registers` and write the same values to the same `registers` so they remain in the same state as one another. No process in such a group can detect the presence of any of its clones, so it has no knowledge of the size of the group and will behave as if the group consisted of it alone. Clones are useful in covering arguments, since an adversary can delay one process of the group just before it is about to write and use it to cover a `register`.

Another way to find a bad execution for an algorithm is to paste together information from several executions, started from different, carefully selected initial configurations. An adversary chooses these executions so that each process finds certain pairs of the executions indistinguishable. Such proofs are called **scenario arguments**. Sometimes, for reasons of clarity, these executions are described implicitly, by giving a simple way of generating them in a (possibly different) system.

A **chain argument** is a particularly useful approach for agreement problems such as consensus. Consider a chain (i.e. a sequence) of executions such that, for any two adjacent executions in the chain, the resulting configurations are indistinguishable to some processes. If processes output different values in the first and last executions, then there must also be two adjacent executions in this chain where processes output different values. This leads to a contradiction, since processes that cannot distinguish between these two executions must produce the same output values in both. Sometimes these chains are constructed inductively, and may be quite long and complicated.

Formal notions of **knowledge** can be used to show that, in a precise sense, common knowledge cannot be gained in some asynchronous systems [166]. The structure of the proofs is similar to some chain arguments. Knowledge-based arguments are also used to show lower bounds on how fast common knowledge of certain facts can be achieved in synchronous systems.

The **valency argument** has become the most widely-used technique for proving that consensus and related problems are impossible in various models of distributed computing. It was introduced by Fischer, Lynch and Paterson [141] to prove that consensus is unsolvable in an asynchronous message-passing system, even when message delivery is reliable, if there is the possibility of even a single process failing. (See Sect. 5.1.) Chor, Israeli and Li [98], Loui and Abu-Amara [230] and Herlihy [169] adapted the valency argument to show impossibility results for several asynchronous shared-memory models. (See Sect. 5.2.) The proofs classify each configuration of the system as either **univalent**, if all executions starting from the configuration produce the same output, or **multivalent**, if there are at least two executions starting from the configuration that produce different outputs. There are two parts to a valency argument. The first part is to show that every algorithm has a multivalent initial configuration. This typically follows from the problem specifications, often via a chain argument. The second part is to show that from every multivalent configuration there is a non-empty execution that results in a multivalent configuration. This is usually proved by contradiction. Attention is focused on the point in an execution where the outcome of the algorithm is determined. One assumes the existence of a **critical configuration**, a multivalent configuration such that all configurations that can be reached from it are univalent. Then one argues, usually case by case, that any possible steps by processes after the critical configuration will result in a contradiction of the definition of the critical configuration. These two parts imply the existence of an infinite execution containing only multivalent configurations, which contradicts termination conditions of the problem. One must also ensure the infinite execution satisfies all fairness constraints of the model. Valency arguments have been adapted to prove unsolvability results for shared-memory systems with non-deterministic objects and lower bounds in synchronous systems and for randomized algorithms.

Symmetry arguments prove impossibility results by showing that several processes must perform similar actions. Lewis Carroll gave a succinct example in 1872: " 'But if everybody obeyed that rule,' said Alice, who was always ready for a little argument, 'and if you only spoke when you were spoken to, and the other person always waited for YOU to begin, you see nobody would ever say anything" [83]. A century later, Rosenstiehl, Fiksel and Holliger [272] used symmetry arguments in the context of distributed computing. These arguments are particularly useful for models with anonymous processes or for comparison-based algorithms. They can often be applied even when there are no faults in the system and when processes behave synchronously. The proofs focus attention on processes that are in the same state (or, more generally, in states in which they behave the same way). The processes retain this property, provided they continue to receive the same (or sufficiently similar) information. For example, in a ring of anonymous processes, deterministic leader election is impossible, because there is no way to break initial symmetry [26]. Similar arguments can be used to show the impossibility of solving resource-sharing problems, such as the dining philosophers problem [266]. Lower bounds on time complexity can be obtained by starting with a highly symmetric configuration and bounding the rate at which the symmetry decreases. Each time one process must send a message, an adversary can force all similar processes to send messages too, yielding good lower bounds for message complexity. Some symmetry arguments designed for comparison-based algorithms can be extended to more general algorithms using **Ramsey theory**

techniques, provided process identifiers or input values can be chosen from a very large set.

In networks, some lower bounds on message complexity rely on the observation that it takes many messages to get information from one process to distant processes. These are called **distance arguments**. Similarly, it takes a long time to collect information from many different processes in both message-passing and shared-memory systems. An **information-theoretic** approach begins by carefully defining a measure of information (e.g., the number of input values that influence the state of the process at a given point in time). Then a recurrence is used to describe how much the information can increase in a process or shared object as a result of a single step. In synchronous models, information-theoretic arguments are complicated by the unexpected ways such information can be acquired. For example, information can be conveyed from one process to another by the fact that a message was not sent in a particular round. Cook, Dwork and Reischuk [106] dealt with similar issues in the context of lower bounds for synchronous parallel computation.

A systematic approach to understanding the computational power of different models is to obtain **simulations** of some models by others. This allows results derived in one model to be extended to other models. For example, suppose that one system A can simulate another system A'. This means that any algorithm designed for system A' can be converted into an algorithm that solves the same problem in A. Thus, a problem that has been proved unsolvable in A, is also unsolvable in A'. Similarly, lower bounds in A imply lower bounds in A', although the bounds obtained for A' may be smaller, depending on the efficiency of the simulation. As is the case for sequential computation, **reductions** from problems that are known to be hard or unsolvable are useful for obtaining additional impossibility results.

The **topology** of geometric objects called simplicial complexes has been applied very effectively to obtain impossibility results for distributed tasks. One can represent the computation of an algorithm by a complex called the **protocol complex**. It can be shown that all protocol complexes for a particular model of computation have certain topological properties. One can also represent a distributed task by a map from a complex representing the possible inputs to a complex representing the possible outputs. When there exists an algorithm to solve a given task in a given model, there is a **decision map** from the corresponding protocol complex to the output complex, satisfying certain properties. Topological arguments can then be used to show that such decision maps cannot exist. This technique is described in greater detail in Sect. 8.

Giving a **characterization** of the set of solvable problems for a particular model is a good way to show unsolvability for many problems in a systematic way. Some important characterizations have been given using topological arguments. Others describe algebraic properties of functions that can be computed or objects that can be implemented. Some characterizations describe how the solvability of a problem depends on the set of allowable inputs: if the inputs are sufficiently restricted, an unsolvable problem may become solvable. One can also characterize the types of shared objects that can be used to solve a particular problem.

When proving one model is more powerful than another, **constructive arguments** are often effective. The idea is to construct a carefully tailored problem that is easy to solve in the first, but difficult or impossible to solve in the second. To prove impossibility results about a class of objects, it suffices to construct an object in the class that demonstrates the difficulty. Some lower bounds show there is no algorithm solving a problem that works well on all networks. One way to do this is to construct a network, specially designed to have bad properties, which can be exploited by an adversary to force algorithms working on the network to be inefficient.

A final important technique for the study of impossibility results is the construction of **counter-example algorithms** [233]. These are algorithms that are not necessarily practical, but they point out the limitations of existing impossibility results, by finding ways of circumventing them. They can also be counter-examples to impossibility conjectures. In the same way that lower bounds tell us we should stop looking for better algorithms, counter-example algorithms tell us we should stop looking for better lower bounds.

5 Unsolvability of consensus and other problems

One of the earliest impossibility results in distributed computing concerns the **two generals problem**: a version of consensus for two processes in a message-passing system where messages can be lost. Ordinary consensus is clearly unsolvable if all messages can be lost. So, in the two generals problem, the problem specification is weakened so that validity must hold for fault-free executions, but need not hold for executions where messages are lost. (Agreement and termination must still hold for all executions.) In 1978, Gray proved that the two generals problem is unsolvable [154,233]. The proof can be formalized as a chain argument that establishes the result even for synchronous systems. Suppose there is an algorithm for solving the two generals problem. Consider an execution with reliable message delivery in which both processes have initial value 0 and, hence, output 0. A chain of executions is constructed: each execution is obtained from the previous one by removing the last message-receipt event (i.e. the message is lost). Each execution cannot be distinguished from the previous one by the sender of the lost message. For the last execution in the chain, both processes have value 0, but no messages are received. Then add one additional execution to the chain: the first process has initial value 0, the second process has initial value 1 and no messages are received. In each execution, the two processes must output the same value. Since each pair of consecutive executions in this chain is indistinguishable to one of the two processes, it follows that both processes must output 0 in all executions in this chain. Similarly, there is a chain of executions that starts with an execution with reliable message delivery in which both processes have initial value 1 and ends with an execution in which the first process has initial value 0, the second process has initial value 1, and no messages are received. Both processes must output 1 in all executions in this chain. This is a contradiction, since the last execution in both chains is the same.

5.1 Asynchronous message-passing models

There is no 1-resilient solution to terminating reliable broadcast in asynchronous message-passing systems, even if only crash failures may occur: If the sender is very slow, the other processes must output their values before any of them have received a message from the sender. An adversary can then ensure that the sender's input is a different value. A similar adversary argument is used to prove that the commit problem is unsolvable in an asynchronous system, even if process crashes may only occur before the first step of any execution [291].

Halpern and Moses [166] introduce formal notions of knowledge for the study of distributed systems. Using essentially the same chain argument as for the two generals problem, they show that, in a precise sense, common knowledge cannot be attained when message delivery is unreliable or there is no upper bound on the time for messages to be delivered. Then they prove that common knowledge is necessary for simultaneous consensus. Hence, this problem is unsolvable unless message delivery is guaranteed within a bounded amount of time.

A similar chain argument is used by Koo and Toueg [209] for studying asynchronous systems with transient message losses (i.e. there is no infinite execution where the same message is sent repeatedly along a communication channel, but is never received). They show that, in this model, common knowledge can be achieved, but any algorithm that satisfies even a weak form of termination cannot guarantee any transfer of knowledge. In particular, consensus is not solvable in this model.

Fischer, Lynch and Paterson [141] developed the valency argument to give the first proof that 1-resilient (and, hence, wait-free) binary consensus is impossible in an asynchronous message-passing system with reliable message delivery. They considered a strong model where, in one atomic step, a process can receive a message (or find out that no messages are available for delivery), perform local computation, and send an arbitrary finite set of messages to other processes. A lower bound in this model automatically applies to the standard message-passing model. Here, we demonstrate the technique by proving the unsolvability of wait-free binary consensus in a standard message-passing model with send steps and receive steps.

Suppose a consensus algorithm has a critical configuration C. (See Sect. 4.1 for the definitions of terms used in valency arguments.) Then, from C, there is a step s_0 that leads to a univalent configuration C_0 from which all executions output only 0 and a step s_1 that leads to a univalent configuration C_1 from which all executions output only 1. Two different possible cases for s_0 and s_1 are illustrated in Fig. 1. The outputs labelled by ? cause contradictions no matter what value they have.

If these steps are by different processes, then the same configuration C' can result if s_1 is performed from configuration C_0 or s_0 is performed from C_1. (For example, suppose s_0 is a send step and s_1 is a receive event, for the same channel, in which no message is available to be received. The step s_1 can follow s_0, provided the adversary delays delivery of the message sent by s_0.) By termination, there is an execution from configuration C' that outputs a value. This is a contradiction, since this value cannot be both 0 and 1.

Now suppose steps s_0 and s_1 are by the same process P. The only instruction P can perform whose outcome is not uniquely determined is to check an incoming channel for messages: In one step, P receives a message from the channel, but,

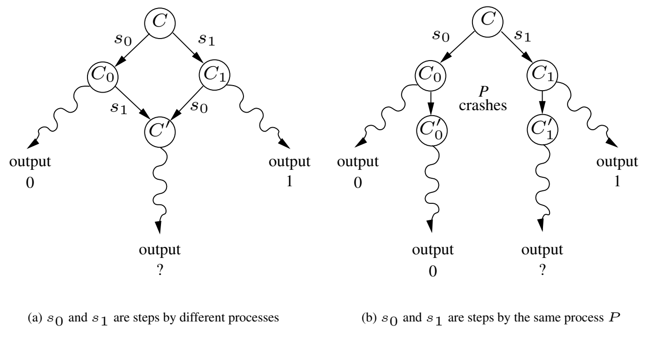

Fig. 1. Two possibilities for a (multivalent) critical configuration C (where C_0 and C_1 are univalent)

in the other step, this message is not yet available. Let C_0' and C_1' be the configurations obtained from C_0 and C_1 when P crashes. By termination, there is an execution starting from C_0' in which 0 is the output value. However, C_0' and C_1' are indistinguishable to all processes except P, so the same sequence of steps can be performed from C_1'. But this sequence of steps outputs 0, contradicting the fact that all executions from C_1 must output 1.

The existence of a multivalent initial configuration is established by a chain argument that considers executions from initial configurations with successively more processes having input value 1. Since the algorithm has no critical configuration, there must be a non-terminating execution and, hence, the algorithm does not solve consensus. The proof for the impossibility of 1-resilient consensus is slightly more difficult, because the adversary must ensure that this non-terminating execution has at most one failure and all other processes take an infinite number of steps. By examining a well-chosen problem, Fischer, Lynch and Paterson obtained an elegant impossibility result which was later adapted to prove results in many other models of distributed systems.

In contrast to their impossibility result, Fischer, Lynch and Paterson also present an f-resilient consensus algorithm, for $f < n/2$, provided faulty processes take no steps (i.e. crashes occur only before the first step of any execution). This shows that it is not the occurrence of faults that makes consensus unsolvable in an asynchronous message-passing system, but, rather, it is the uncertainty of when these faults might occur.

Dolev, Dwork and Stockmeyer [114] extended the work of Fischer, Lynch and Paterson by considering how the solvability of consensus is affected by five properties of a message-passing system: whether there is a bound on relative process speed, whether there is an upper bound on message delivery time, whether messages to each process are received in the order in which they were sent, whether a process can send more than one message in a single step, and whether a process can receive and send in a single step. Using valency arguments, they proved that certain combinations of these properties make 1-resilient or 2-resilient consensus unsolvable. They also provided wait-free consensus algorithms in models with the remaining combinations of properties, and 1-resilient consensus algorithms for those models where they had proved the impossibility only of 2-resilient consensus. For example, wait-free consensus is solvable is an asynchronous system in which a process can send and receive multiple messages in a single step and there is an upper bound on the message delivery time known to all processes. However, if this model is changed to allow arbitrary process faults, then Attiya, Dolev, and Gil [33] showed, using a valency argument, that consensus becomes unsolvable, even if the validity condition is only required to hold when all processes are non-faulty and have the same input value. Interestingly, if they replace validity with a non-triviality condition (i.e. two different output values can occur), they show that any number of arbitrary process faults can be tolerated.

Welch [296] shows how an asynchronous message-passing system can simulate a message-passing system with synchronous processes, but no bounds on message delivery time. Combined with the fact that 1-resilient consensus is unsolvable in the former model, this gives a different proof that 1-resilient consensus is also unsolvable in the latter model.

5.2 Asynchronous shared-memory models

Chor, Israeli and Li [98], Loui and Abu-Amara [230], Abrahamson [1], and Herlihy [169] adapted valency arguments to show that wait-free consensus is unsolvable for two processes that communicate using `registers`. Using slightly more complicated valency arguments, Loui and Abu-Amara extended this result to show that, for $n > 2$ processes, 1-resilient consensus is unsolvable when processes communicate using `registers` and that 2-resilient consensus is unsolvable using shared-memory systems in which objects have only two states, for example, `test&set` objects. Herlihy proved that `queues` have consensus number 2. This line of research was carried further to give characterizations of the types of shared-memory objects that are capable of solving wait-free consensus. See Sect. 7.2 for results of this type. Herlihy also showed that an array of `registers` has increased power if processes can access several elements of the array in a single atomic action: if a process can atomically assign values to $m > 1$ different elements, the consensus number becomes $2m-2$. This was the first example of an object type whose consensus number was greater than two but still finite. Merritt and Taubenfeld [242] generalized this result to show that consensus can be solved in the presence of up to f crash failures using such an array if and only if $f \leq \max(2m - 3, 0)$. They also gave a lower bound on the number of elements needed in the array to do so. Interestingly, if the m `registers` accessed by an atomic operation must be adjacent in the array, Kleinberg and Mullainathan [208] have shown that the consensus number is 3, rather than $2m - 2$, for all $m \geq 3$.

In most cases, the part of a valency argument that shows no critical configuration can exist amounts to showing that some process cannot always learn enough about which step was taken immediately after the critical configuration to decide on an output value. For example, in Fig. 1(b), the information about which step was performed first is destroyed by having process P crash. Sometimes, the adversarial scheduler must construct much more complicated executions to destroy this evidence. For example, consider Herlihy's proof that three-process consensus cannot be solved using queues [169]. If two processes enqueue different values onto a `queue` just after the critical configuration, processes must be made to dequeue those values (as well as all values that precede them in the queue) to ensure that the order of the enqueue operations cannot be determined. Other types of faults can also be used to conceal information about the first operation that is performed after the critical configuration. This approach is used by Jayanti, Chandra and Toueg [196] to prove impossibility results in models where objects, instead of processes, fail.

Afek, Greenberg, Merritt and Taubenfeld [10] also studied models where objects can fail. They showed that consensus is impossible when processes communicate using any types of objects, if half the processes can crash and the states of half the objects can become corrupted. Assuming an algorithm exists, the proof constructs two reachable configurations that are indistinguishable to half the processes. Both are obtained from an initial configuration where half the processes have input 0 and half the processes have input 1. The first configuration is obtained by running the processes with input 0 until they all produce output values, and then corrupting half the objects so that their states are reset to their initial states. The processes with input 1 do not take any steps. The resulting configuration is univalent with output 0, since the values output by the processes with input 0 must be 0. To construct the second configuration, the processes with input 0 crash before taking any steps and the other half of the objects are corrupted so that their values are the same as in the first configuration. The second configuration is univalent with output 1. But this is impossible, since the two configurations are indistinguishable to the processes with input 1.

Moses and Rajsbaum [250] gave a unified framework for proving impossibility results, based on the valency argument, that applies to both synchronous and asynchronous systems using either message passing or shared memory. Their approach is to restrict the adversarial scheduler to a nicely structured subset of the possible executions. For example, they showed that consensus is unsolvable if processes communicate using only `single-writer registers`, even when processes are guaranteed to be scheduled in slightly asynchronous rounds where, in each round, at least $n - 1$ processes each write to a `register` and then read the values written in that round by at least $n - 2$ other processes. Lubitch and Moran [231] also used a restricted set of runs to obtain a unified way of proving the impossibility of f-resilient consensus in a variety of models. Recall that a valency argument works by showing that every multivalent configuration must have a multivalent successor, in order to construct an infinite execution where no process ever produces an output. However, when studying f-resilient consensus, one must also show that the infinite execution produced has at most f failures. Lubitch and Moran's restricted scheduler ensures that this is automatically true. Other attempts to unify impossibility results for different models include the work of Herlihy, Rajsbaum and Tuttle [177] (see Sect. 8.4) and Gafni [146].

Taubenfeld and Moran [292] used a valency argument to provide a general impossibility result for a large class of problems in asynchronous systems with crash failures, where processes communicate using `registers`. In particular, they show that there is no f-resilient algorithm for the consensus problem restricted to input vectors in which the number of 0's and the number of 1's differ by at least f. This version of consensus can easily be solved in an $(f - 1)$-faulty system: each process waits until it has learned the input values of $n - f + 1$ processes and then outputs the majority value.

Mostefaoui, Rajsbaum, and Raynal [252] recently gave an efficiently decidable characterization of the sets of input vectors for which f-resilient consensus algorithms exist. They also extended their results to message-passing systems.

Malkhi, Merritt, Reiter and Taubenfeld studied shared-memory systems with arbitrary process faults using access control lists, specified by the programmer, specifying which processes can access each shared object [236]. These lists limit the extent to which arbitrary process faults can corrupt the shared memory, making it possible to solve some problems. They used a simple indistinguishability argument to show that, in this model, f-resilient consensus cannot be solved if $n \leq 3f$ (using any type of object). However, if $n \geq 3f+1$, the problem is solvable using `sticky bits` and `registers` [241].

5.3 Bounds on the number of faults in synchronous message-passing models

In synchronous models, consensus and related problems are solvable even when arbitrary process faults can occur, provided there are not too many faults. The earliest papers proved that, in complete networks where arbitrary process faults can occur, terminating reliable broadcast is solvable if and only if less than $n/3$ processes can fail [113,225,260]. Scenario arguments are used to prove the lower bounds.

The following version of the argument, by Fischer, Lynch and Merritt [140], shows that consensus is impossible for three processes, P, Q and R, if one arbitrary process fault may occur. Suppose there is a consensus algorithm for this system. Consider a system with six processes composed of two copies each of P, Q and R, joined in a ring in the order $P_0, Q_0, R_0, P_1, Q_1, R_1$. (See Fig. 2.) Let α be a scenario (i.e. an execution of this system) where P_0, Q_0 and R_0 have input 0 and P_1, Q_1 and R_1 have input 1. The output value of each process in α can be determined by considering another scenario that is indistinguishable to that process.

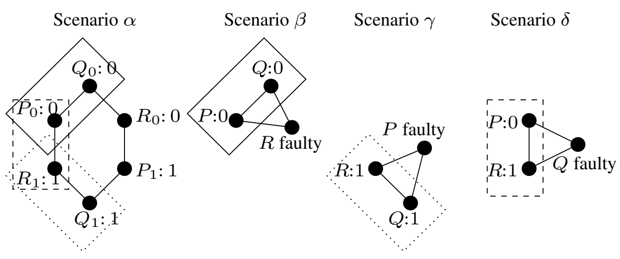

Fig. 2. Scenarios for impossibility of consensus for three processes tolerating one arbitrary process fault

Let β be an execution of the algorithm in a three-process system, consisting of P, Q and R, where each process starts with input 0. Process R is faulty in β. It behaves in the same way towards P as R_1 behaves towards P_0 in α, and it behaves in the same way towards Q as R_0 behaves towards Q_0 in α. Then, the steps performed by P and Q in β are identical to the steps performed by P_0 and Q_0 in α. Since P and Q must output 0 in β, P_0 and Q_0 must also output 0 in α. Similarly, by considering an execution γ in a 3-process system where Q and R have input value 1 and P is faulty, one can show that Q_1 and R_1 must output 1 in α. Finally, consider another scenario δ where P has input 0, R has input 1 and Q is faulty, sending the same messages to P as Q_0 sends to P_0 and sending the same messages to R as Q_1 sends to R_1. Process P must output 0 in δ, since P_0 outputs 0 in α, and R must output 1 in δ, since R_1 outputs 1 in α. This contradicts the agreement property for the execution δ. The elegance of this argument lies in its ability to draw conclusions about the behaviour of processes without requiring any detailed analysis of exactly what those processes are doing.

This unsolvability result can be extended to systems with $n \leq 3f$ processes, of which at most f can have arbitrary faults, via a reduction from the three-process case [225]. Each of the processes P, Q and R can simulate disjoint sets of processes of size at most f. If one of P,Q, or R fails, then at most f of the simulated processes fail.

The **(vertex) connectivity** of a network is the minimum number of nodes that must be removed so that the resulting network is disconnected. We consider the connectivity of the complete network of n nodes to be n. With only crash failures, f-resilient consensus is solvable if and only if the connectivity of the network is greater than f [225]. To see why the impossibility result holds, suppose that f processes crash disconnecting the network and the input values of processes within each component are identical, but different from those in some other component. For each non-faulty process P, any execution from this configuration cannot be distinguished by P from an execution that starts in the configuration in which all processes have the same input value as P. This is because processes in different components cannot communicate with one another. Hence, each process must decide its own input value, violating the agreement condition.

With arbitrary process faults, a higher degree of connectivity is required to solve consensus: Dolev [113] proved, using a scenario argument, that there is an f-resilient algorithm for terminating reliable broadcast only if the connectivity of the network is more than $2f$. Fischer, Lynch and Merritt [140] gave a similar proof to show that consensus is solvable in the presence of f arbitrary faults only if the connectivity is greater than $2f$. Dolev [113] gave an f-resilient algorithm for terminating reliable broadcast provided the network has connectivity more than $2f$ and more than $3f$ processes. Hence, these lower bounds are tight.

These scenario arguments for arbitrary process faults have been extended to weak forms of terminating reliable broadcast [218] and consensus [140], where validity applies only when all processes are non-faulty.

Garay and Perry [148] observed that these results imply that, in f-faulty synchronous message-passing systems where there are at most a arbitrary process faults and at most $f - a$ additional processes that can crash (without taking any steps), consensus is solvable if and only if the number of processes is greater than $f + 2a$ and the connectivity of the network is greater than $f + a$.

Hadzilacos [160] studied the terminating reliable broadcast problem in synchronous message-passing models, where processes can crash and communication channels can fail by ceasing to deliver messages. He proved that this problem is unsolvable, if there is a set of processes and communication channels which can all fail in the same execution and whose removal disconnects the underlying graph. This is done using a reduction from 1-resilient terminating reliable broadcast in a line of three nodes: the middle node simulates the processes in the set, another node simulates the processes in one of the connected components that results from removing this set of processes and channels from the graph, and the last node simulates the remaining processes. Communication channels not in the set are simulated either within a process or by one of the two undirected edges in the line. A simple adversary argument is used to show that 1-resilient terminating reliable broadcast is unsolvable in a line of three nodes. Hadzilacos also gives a matching upper bound, showing that if no such set exists, then terminating reliable broadcast can be solved even if processes and communication channels may fail to send some of their messages.

With partially synchronous communication, where there is an unknown upper bound on message delivery time, and synchronous processes, Dwork, Lynch, and Stockmeyer [125] used a scenario argument to prove that binary consensus is unsolvable if f processes can crash and $1 < n \leq 2f$: They consider an execution where half the processes have input value 0 and half have input value 1, communication between processes with the same input value takes 1 unit of time, and communication between processes with different input values does not occur until both have produced their outputs. For $n > 2f$, they gave a consensus algorithm, for partially synchronous communication and processes, that tolerates f or fewer omission faults. Thus, if communication is partially synchronous, the number of faults that can be tolerated does not depend on whether the faults are crashes or omissions, whether the processes are synchronous or partially synchronous, or whether the input domain is restricted in size. When communication is synchronous and processes are partially synchronous, they show that consensus is solvable for any number of crash failures, but, using a somewhat more complicated scenario argument, prove that $n \geq 2f$ is necessary for omission faults.

Santoro and Widmayer [280] considered a synchronous message-passing model with dynamic omission faults. Specifically, at each round, the message broadcast by one (possibly different) process might not be delivered to some or all of the other processes. Using a valency argument, they prove that a variant of consensus is unsolvable in this model.

The scenario proof illustrated in Fig. 2 can also be used to show that the approximate agreement problem is unsolvable in the message-passing model if there are f Byzantine faults and $n \leq 3f$ [140]. Dolev, Lynch, Pinter, Stark and Weihl [117] gave algorithms for the synchronous case when $n > 3f$ and for the asynchronous case when $n > 5f$. The latter result contrasts with the unsolvability of consensus for $f = 1$ (Sect. 5.1).

5.4 Anonymous systems

Symmetry arguments can often be used to show that tasks are unsolvable in anonymous systems, even when processes and communication are assumed to be completely reliable. For example, leader election is impossible in an anonymous synchronous ring, where processes are arranged in a circle and each process can communicate only with its two neighbours [47,234]. An easy induction shows that, at the end of each round, every process must be in the same state. Thus, if any process declares itself the leader, all other processes do so too, which would violate the definition of leader election. Similarly, in an asynchronous message-passing system, a round-robin schedule maintains symmetry among all the processes, if every message sent during a round is delivered at the very end of that round. Johnson and Schneider [202] and Jayanti and Toueg [199] proved that the same is true for anonymous systems where processes communicate via `registers`. A version of this symmetry argument was also used by Angluin [26] to show that leader election is unsolvable for yet another anonymous model. Rabin and Lehmann [266] gave a nearly identical proof showing that processes in an anonymous ring cannot solve the dining philosophers problem. Bougé [73] used similar arguments to show the impossibility of leader election in various anonymous networks that have underlying graphs which are sufficiently symmetric.

Randomization can sometimes be used to break symmetry. Thus, impossibility results that are proved using symmetry arguments often break down when algorithms are allowed to be randomized. For example, to elect a leader in an anonymous ring of known size, all processes can choose random identifiers, and elect the process with the smallest unique identifier, if there is one [26]. (If there is no unique identifier, the algorithm is repeated.) This algorithm does have infinite executions, but it will terminate with probability 1.

Some distributed tasks can be viewed as computing a function of n inputs, where the inputs are initially distributed, one to each process. Cidon and Shavitt [102] showed that many important functions cannot be computed by randomized algorithms in a synchronous anonymous ring of unknown size. (See Sect. 13.1.) Attiya, Snir and Warmuth [46] gave characterizations of the functions that can be computed on rings of anonymous processes for both synchronous and asynchronous models. Yamashita and Kameda [298] gave a characterization of the functions that can be computed in an asynchronous, reliable, anonymous network, if processes know the network topology. For example, if a function is computable, then for any automorphism of the network graph, the corresponding permutation of input values cannot change the value of the function. Boldi and Vigna [66,67] gave a characterization when only partial knowledge about the network topology is known: they assume that processes know only that the network graph comes from a known set of possible graphs. Attiya, Gorbach, and Moran [39] gave similar characterizations for functions and agreement tasks (i.e. decision tasks satisfying agreement) that can be computed in a reliable, anonymous shared-memory system where processes communicate via `registers`. All of these results rely heavily on symmetry arguments.

5.5 Clock synchronization

An important problem is to determine how closely clocks (of non-faulty asynchronous processes) can be synchronized when there is uncertainty in message delivery time. Each process knows the times (on its own physical clock) at which its incoming messages arrive. All clocks are assumed to run at the same rate. Lundelius and Lynch [232] proved, using a shifting argument that, when the maximum and minimum times for message delivery on each edge can differ by D, then there are executions in a complete network of n processes in which clocks of two different processes differ by at least $2D(1 - 1/n)$. To obtain these results, they first construct an execution with message delivery times within the allowable ranges. Then, for each process P_i, its physical clock and the times at which its steps occur are shifted by some amount Δ_i, with message delivery times still within the allowable ranges. These two executions are indistinguishable to all processes, so processes will adjust their clocks the same ways in both. However, the time between steps of different processes will differ in the two executions. This implies the lower bound on the guaranteed closeness of the synchronization of the clocks.

Halpern, Megiddo, and Munshi [165] and Biaz and Welch [60] extended this work to arbitrary networks, with different uncertainties in message delivery time on different edges. Specifically, if the uncertainties are treated as edge weights, then half the weighted diameter of the graph is a lower bound on the quality of the approximation that can be achieved. For some graphs, they gave a matching upper bound; for the rest, they showed this bound is tight to within a factor of 2. Halpern, Megiddo, and Munshi also proved that randomization does not help to solve this problem.

Attiya, Herzberg, and Rajsbaum [40] consider the situation where upper bounds on message delivery time might not exist, but other information is available, such as bounds on the difference in message delivery times for opposite directions of a link. They gave lower bounds on how closely clocks can be synchronized in terms of how far the steps performed by each pair of processes can be shifted in time relative to one another in an execution.

Scenario arguments have been used to obtain unsolvability results, similar to those in Sect. 5.3, for the clock synchronization problem when the physical clocks of the processes do not run at the same rate, but are bounded above and below by linear functions of real time. An algorithm must ensure that the adjusted clocks remain close to one another. In addition, adjusted clocks must also be bounded above and below by linear functions of real time. Otherwise, the problem is trivial: For example, when the adjusted clock of each process is proportional to the logarithm of its physical clock, the adjusted clocks will eventually become close to one another, so clock synchronization can be achieved without any communication [115]. Dolev, Halpern, and Strong [115] proved that this version of clock synchronization is impossible in a network with uncertainty in message delivery times, if at least f of the processes can have arbitrary faults and $n \leq 3f$. If message delivery is always instantaneous, clock synchronization is also unsolvable for $n \leq 3f$, using essentially the same proof. Algorithms for complete networks exist when $n > 3f$ [297,224]. Fischer, Lynch, and Merritt [140,233] gave similar, simpler proofs of these unsolvability results and extended them to networks with connectivity at most $2f$. When the message delivery time on each link is a known, non-zero value, Fischer, Lynch, and Merritt [115] showed that, once clock synchronization has been achieved, it can be maintained with up to f arbitrary process faults if and only if the degree of every node in the network graph is at least $2f + 1$. If a process sends a message to a correct neighbour and waits for an acknowledgement, it can determine how much real time has passed and increment its clock accordingly. The fact that a majority of each process's neighbours are correct can be used to ensure that faulty processes do not cause correct processes to update their clocks incorrectly.

If only crash failures can occur, Simons, Welch, and Lynch [285] observed that network connectivity of at least $f + 1$ is required; otherwise the network can become disconnected. For $(f + 1)$-connected graphs, f-resilient clock synchronization algorithms exist, even when messages can be lost [115].

Srikanth and Toueg [288] studied the accuracy of clocks with respect to real time in fault-free asynchronous systems, where there is a known upper bound on message delivery time and the physical clocks are bounded above and below by known linear functions of real time. They used a stretching argument to prove that it is impossible to ensure that the adjusted clocks more closely approximate real time. Even if there are up to f processes with arbitrary faults, they show this accuracy with respect to real time can be achieved, provided that $n \geq 2f+1$. They also prove that $n \geq 2f+1$ is necessary, even for a much weaker fault model in which processes can only fail by having physical clocks that violate the assumed rate bounds.

Patt-Shamir and Rajsbaum [259] and Fetzer and Cristian [135] obtain lower bounds on how closely clocks can be synchronized, if processes can only send synchronization information to one another by piggybacking on messages that are being used by the system for other purposes.

5.6 Other problems

There are many other distributed computing problems for which unsolvability results are known. For example, mutual exclusion is unsolvable in systems with unreliable processes for a very simple reason: a process that crashes in its critical section prevents any further progress. In this section, we shall briefly mention some other results. Section 7.3 discusses results that characterize solvable tasks in various models, which can be used to obtain unsolvability results for additional problems.

Coan, Dolev, Dwork, and Stockmeyer [103] considered the distributed firing squad problem. Using scenario arguments, they proved bounds on the number of process faults that can be tolerated when processes can exhibit various types of timing faults.

Topological arguments have been used to prove that wait-free set consensus is unsolvable in asynchronous systems, except in the trivial case where every process can have a different output value. This is discussed in more detail in Sect. 8.2. De Prisco, Malkhi and Reiter [109] studied the solvability of set consensus with a number of different versions of validity. Using reductions and simple adversary arguments, they prove bounds on the number of crash and arbitrary process faults that can be tolerated. They also develop algorithms that show these bounds are tight for most cases.

Bridgland and Watro [76] prove that there is no f-resilient solution to task assignment in message-passing systems with fewer than $\sum_{i=1}^{f} \lceil (f+1)/i \rceil$ processes. They do this by explicitly constructing a bad execution with the aid of the pigeonhole principle.

Attiya, Bar-Noy, Dolev, Koller, Peleg and Reischuk [35] used a valency argument to show that, for any asynchronous message-passing algorithm for renaming that tolerates one crash failure, the smaller name space must contain at least $n+1$ identifiers. Herlihy and Shavit [179] later generalized this result using their Asynchronous Computability Theorem (see Sect. 8.3). They said that an asynchronous renaming algorithm that tolerates f crash failures in a shared-memory model using only `registers` (and, hence, in the message-passing model) must have a name space of at least $n+f$ identifiers to choose from. Attiya, Bar-Noy, Dolev, Koller, Peleg and Reischuk [35] proved a lower bound for order-preserving renaming: the name space must contain at least $2^f(n-f+1)$ identifiers. Their proof considered executions where f of the processes run sequentially after the other $n-f$ have halted. They showed that in such runs, the processes that finish early must leave large gaps between the identifiers they choose, so that the processes that start running later can choose names from within the gaps. They also gave matching upper bounds on the size of the name space for renaming and order-preserving renaming.

Once processes have determined their output values in an asynchronous, message-passing algorithm for renaming, they must continue to take steps. This is because other processes that begin executing later need to communicate with them to find out which output values have already been taken. Using a game-theoretic characterization of solvable tasks (see Sect. 7.3), Taubenfeld, Katz, and Moran [291] proved that there is no f-resilient order-preserving renaming algorithm in an asynchronous message-passing system, for $f \geq 2$, if the processes are required to terminate. The impossibility result holds even if the faulty processes can only crash at the beginning of an execution, without taking any steps.

The k-assignment problem was studied by Burns and Peterson [80], who used a valency argument to show that, for $f \geq k/2$, there is no f-resilient solution in an asynchronous system where processes communicate using `registers`.

Chandra, Hadzilacos, Toueg, and Charron-Bost [87] extended the valency argument of Fischer, Lynch and Paterson [141] to show that the group membership problem is unsolvable in an asynchronous message-passing system with one crash failure, even if the algorithm must satisfy only rather weak correctness properties. There have been many systems that solve versions of the group membership problem using stronger models; see the survey by Chockler, Keidar and Vitenberg [97].

Jayaram and Varghese [201] consider a message-passing model where message channels may drop packets and each process may experience faults that reset its local memory to its initial state. Building on earlier work by Fekete, Lynch, Mansour, and Spinelli [131], they showed that an algorithm in this model can be driven into a global state that contains any combination of reachable local states. It follows easily that problems such as leader election and mutual exclusion are impossible in this model.

Greenberg, Taubenfeld and Wang [155] studied the choice coordination problem in asynchronous systems where processes communicate using read-modify-write objects. Processes have identifiers, although more than one process may have the same identifier. They showed that a wait-free solution exists if and only if the number of processes that share the same identifier is less than the least non-trivial divisor of k, the number of alternatives from which processes must choose. The unsolvability result is proved using a simple symmetry argument.

6 Relationships between models

Researchers in distributed computing have proposed a large variety of mathematical models of distributed systems. One of the central goals of the theory of distributed computing is to understand the relationships between them. Which ones can solve more problems? Which ones can solve problems more efficiently? In this section, we examine how impossibility results can address these questions and how the answers to these questions can give rise to new impossibility results.

The most direct way to compare two different models is to show that one can simulate the other. For example, a shared-memory system can simulate a message-passing system with the same number of processes by using a `single-writer single-reader register` that can be written to by P and read by P' to represent a communication channel from P to P' [47,292]. Then, as mentioned in Sect. 4.1, impossibility results for this shared-memory model immediately imply impossibility results for the message-passing model. This gives an alternate proof of the unsolvability of 1-resilient consensus for $n > 1$ processes in asynchronous message-passing systems (Sect. 5.1), since this problem is unsolvable in asynchronous systems when processes communicate using `registers` (Sect. 5.2). Simulations of shared-memory by message-passing models will be discussed in Sect. 6.3.

6.1 System size

Here we describe some simulations between systems with different numbers of processes that are used in a crucial way to establish impossibility results. For clarity, throughout this section, we call the simulated processes **threads** and the simulating processes **simulators**.

An easy observation is that an f-faulty system of $n+\Delta$ simulators, where $n > f$ and $\Delta \geq 0$, can simulate an f-faulty system of n threads, by employing only n of the simulators, each of which performs the actions of a different thread. Similarly, an f-faulty system of $n > f$ simulators can simulate an $(f+\Delta)$-faulty system of $n+\Delta$ threads. Each simulator performs the steps of a different thread. The unsimulated threads are considered to be crashed processes.

A more interesting simulation of shared-memory systems, by Chandra, Hadzilacos, Jayanti, and Toueg [86], shows how, for any $n, n' > f$, an f-faulty system of n' simulators can simulate an f-faulty asynchronous system of n threads. The simulators each try to perform steps of each of the threads, using a total of n `registers` and n `test&set` objects in addition to the objects used by the threads. One `test&set` object is associated with each thread. It is used as a lock to ensure that only one simulator at a time performs a step of that thread. The state of the thread is recorded in one of the `registers`. To perform a step of the thread, a simulator obtains its lock, reads the thread's state from the `register`, executes the next step on behalf of the thread, and updates the state stored in the `register`. Then it releases the lock and proceeds to the next thread. If a simulator tries to perform a step of a thread, but does not get access to the lock, it continues with the next thread. The key observation is that each simulator crash causes the crash of at most one simulated thread. This is illustrated in Fig. 3, where a $\surd$ indicates that the specified simulator has simulated the step, and an $\times$ indicates that the simulator crashed while simulating the step. Because steps of the threads may be performed at different rates, the simulated system must be asynchronous.

		steps					
		1	2	3	4	5	$\cdots$
	T_1	$P_1\surd$	$P_2\surd$	$P_1\surd$	$P_1\surd$	$P_1\surd$	
	T_2	$P_1\surd$	$P_1\surd$	$P_3\times$			
threads	$\vdots$			$\vdots$			
	$T_{n'}$	$P_3\surd$	$P_1\surd$	$P_2\surd$	$P_1\surd$		

Fig. 3. An illustration of an execution of Chandra, Hadzilacos, Jayanti and Toueg's simulation

Any set of object types that has consensus number at least 2 can be used, together with `registers`, to implement `test&set` objects [10] and, hence, perform Chandra, Hadzilacos, Jayanti and Toueg's simulation. This implies that if a set of object types can be used to solve f-resilient consensus among n processes, for some $n > f \geq 2$, then it can be used to solve wait-free consensus among $f + 1$ processes, i.e. the consensus number of the set is at least $f + 1$. Note that, for $f = 1$, this result is false: Lo and Hadzilacos [226, 228] construct object types that can be used to solve 1-resilient consensus among n processes, for each $n > 2$, but cannot be used to solve wait-free consensus for two processes. The latter is proved using valency arguments, which had to be adapted to handle the non-determinism of some of their objects. A special case of this result is that there are object types that can be used to solve 1-resilient consensus for $n = 3$ processes, but not for $n - 1 = 2$ processes. However, if a set of object types can be used to solve to solve 1-resilient consensus for $n \geq 4$ processes, those object types can also be used to solve 1-resilient consensus for $n - 1$ processes. Lo and Hadzilacos [228] proved this by using a non-wait-free implementation of `test&set` objects by `registers` to adapt the simulation.

The BG simulation [69,72] describes how an f-faulty system of n' simulators (for any $n' > f$) can simulate an f-faulty asynchronous system of n threads, where both simulators and threads communicate using only `snapshot` objects or, equivalently, `registers` [6,22,30]. A key element of the simulation is the **safe agreement** subroutine. It satisfies the agreement and validity properties of the consensus problem, but might not terminate. Every simulator simulates the steps of every thread, including steps where a thread receives its input. This is done in parallel for all simulators and threads. The simulators use safe agreement to ensure that a simulated step has the same result in the simulations carried out by different simulators. Each simulator visits the threads in round-robin order and tries to execute the next step of each thread it visits. The safe agreement routine is designed so that if simulators are running several copies of the safe agreement routine in parallel, a simulator failure will block the simulation of at most one thread. This ensures that, if the original algorithm was f-resilient, then the resulting algorithm will also be f-resilient.

For a wide class of problems such as consensus, set consensus, and approximate agreement, a simulator can terminate if it observes a thread that has terminated, using the output of the thread as its own output. This allows impossibility results for such problems to be extended from one model to another using the BG simulation. For example, as described in Sect. 8.2, there is no wait-free (i.e. k-resilient) k-set consensus algorithm for $k + 1$ processes that communicate using `registers`. Then, the BG simulation implies that there is no k-resilient k-set consensus algorithm for n processes, for any $n > k$. Similarly, from the fact that wait-free consensus is unsolvable for two processes, it follows that f-resilient consensus is unsolvable when $n > f \geq 1$.

An (n, k)-`set consensus` object is designed to allow n processes to solve k-set consensus. Specifically, each of the n processes can perform a propose operation, which takes one input as an argument and returns a value that satisfies validity (the value returned is an input of some propose operation to that object) and k-agreement (at most k different values are ever returned). Suppose $k' \geq k$ and we wish to implement an (n', k')-`set consensus` object from (n, k)-`set consensus` objects and `registers`. One very simple way to do this is to divide the n' processes into $\lfloor n'/n \rfloor$ groups of size n and one group of size n' mod n. Then each group can use a different (n, k)-`set consensus` object to produce at most k distinct outputs, except for the last group, which will produce at most $\min(k, n' \bmod n)$ distinct outputs. This simple approach works, provided $k' \geq k \lfloor n'/n \rfloor + \min(k, n' \bmod n)$. It has been shown that no implementation is possible for smaller values of k' [68,95]. This was done using an extension of the BG simulation that can be applied to algorithms that use (n, k)-`set consensus` objects as well as `registers`: the results of propose operations are determined using safe agreement.

6.2 Timing assumptions

An important reason to study impossibility results is to establish that one model of computation is more powerful than another. This is done by showing that a particular problem that can be solved in the first model either cannot be solved as efficiently or cannot be solved at all in the second model. Here, we describe some impossibility results which were designed specifically to establish such separations between models having different assumptions on process speeds.

Synchronous systems are known to be more powerful than asynchronous systems: It is possible to solve wait-free consensus in synchronous message-passing systems on complete networks (Sect. 5.3), but not in asynchronous shared-memory systems where processes communicate via `registers` (Sect. 5.2) or in asynchronous message-passing systems (Sect. 5.1). This separation between synchronous and asynchronous models was further refined by Dolev, Dwork and Stockmeyer [114], who examined which properties of the model are needed for consensus to be solvable in a fault-tolerant way. (See Sect. 5.1.)

Dwork, Lynch and Stockmeyer [125] showed that, when solving consensus in a message-passing model, some partially synchronous systems lie between synchronous and asynchronous ones: consensus can be solved in the partially synchronous system, but the number of failures that can be tolerated is lower than in the fully synchronous model. (See Sect. 5.3.)

One of the first separations between synchronous and asynchronous shared-memory systems was proved by Arjomandi, Fischer and Lynch [27], who considered the time required by algorithms in systems where no faults can occur. They used a simple synchronization task called the **session problem**. A **session** consists of a section of an execution in which a particular set V of n objects are each accessed at least once. The **(s,n) session problem** is to design an n-process algorithm such that any execution can be partitioned into s sessions. It is assumed that each object can be accessed by at most b processes. There is a trivial synchronous algorithm that solves the problem in s rounds: every process accesses a different object and each process performs s accesses. However, the same approach does not work in an asynchronous setting, because one process could perform its last access before some other process performs its first, resulting in an execution with a single session. They prove that, in an asynchronous system, $\Omega(s \log_b n)$ rounds are required. (Recall that, in an asynchronous system a round is a segment of the execution where every process takes at least one step.) This implies that a straightforward step-by-step and process-by-process simulation of an n-process synchronous system by an n-process asynchronous system necessarily increases the round complexity by a factor of $\Omega(\log_b n)$, assuming each object can be accessed by at most b different processes.

We sketch the proof of the lower bound. Consider a round-robin execution of any solution to the problem. Partition the execution into segments of $\lfloor \log_b n \rfloor$ rounds each. We say that one step directly depends on an earlier step if they involve the same object or are performed by the same process. One step depends on an earlier one if there is a chain of such direct dependencies linking them. This notion of dependency is useful for determining whether information could have propagated from one object to another. If two steps do not depend on one another, they can be reordered without any process noticing the change. For any particular step, the number of objects that have been accessed by steps that depend on that step increases by a factor of at most b (roughly) in each round. It follows from the fact that the kth segment contains only $\lfloor \log_b n \rfloor$ rounds that information cannot be propagated from object $v_k \in V$ to some other object $v_{k+1} \in V$. Therefore, the steps within the segment can be reordered so that all accesses to v_{k+1} take place before all accesses to v_k and no process will notice the difference. For the reordered run, one can prove by induction that the kth session ends no earlier than the first access to v_k in the kth segment. Suppose this is true for the kth session. Then there is no access to v_{k+1} in segment k after the end of session k, so session $k+1$ cannot end before the first access to v_{k+1} in the next segment. It follows that at most one session ends in each segment. Since the problem requires that at least s segments occur in the reordered run, the original run must have contained at least s segments and hence $\Omega(s \log_b n)$ rounds.

Similar techniques were used by Rhee and Welch [271] and by Attiya and Mavronicolas [43] to analyse the complexity of the session problem in other shared-memory and message-passing settings, including partially synchronous models. These results again established separations between the various models.

Awerbuch [50,51] proved a tradeoff between the time and the amount of communication necessary for asynchronous message-passing systems to simulate a round of a synchronous algorithm. This implies lower bounds on the ability of an asynchronous system to simulate a synchronous system in a round-by-round manner. To prove the tradeoff, he considered specially constructed networks with many edges, but no small cycles. During the simulation of a round, a node must receive information from each of its neighbours, either directly or via a chain of messages, before it knows that the round has been completed. If information travels across every edge in the graph, the number of messages used will be high. However, if no information travels across some edge, that information must instead travel along a lengthy path, since there are no short cycles. In this case, a large amount of time will be used.

6.3 Consistency conditions

Another way in which models may differ is in the way correctness is defined for objects when they are accessed concurrently by different processes. Linearizability (defined in Sect. 2) is a strong condition. **Sequential consistency** [217] is a weaker condition. Like linearizability, operations must appear to happen instantaneously at distinct point in time and, for each process, the order in which its operations appear to happen must be consistent with the order in which that process performed them. However, the ordering need not respect the real-time order of pairs of operations performed by different processes. **Hybrid consistency** [38] requires that some operations satisfy stronger consistency conditions than other operations. We now look at lower bound results that were proved to show there exist implementations satisfying weaker consistency conditions which are more efficient than any implementation satisfying stronger conditions.

Attiya and Welch [48] considered the problem of implementing `registers`, `stacks` and `queues` in a fault-free message-passing model. They gave complexity separations between linearizable implementations and sequentially consistent implementations. For example, if the time required for a message to travel from one process to another is between d_1 and d_2, they used a shifting argument to show that, for any linearizable implementation of `registers`, read and write operations have worst-case time bounds of at least $(d_2 - d_1)/4$ and $(d_2 - d_1)/2$, respectively. Sequentially consistent `registers`, on the other hand, can be implemented so that one of the two operations (read or write) requires no communication at all and hence incurs no delay, while the other operation runs in time $2d_2$.

The basic idea for their lower bound proofs is similar to earlier work on clock synchronization lower bounds [232]: If there is sufficient uncertainty about the length of time it takes for messages to be delivered, one can shift the execution of one process in time without any process noticing the difference. For example, suppose there is an execution where all message deliveries take time $(d_1 + d_2)/2$. Consider the execution that results if process P's steps are shifted, so that a step that took place at time t in the original run now takes place at time $t + (d_2 - d_1)/2$. The new run is still legal: the delivery time for each message sent to P is d_2 and the delivery time for each message sent from P is d_1. Furthermore, the two runs are indistinguishable to all processes. Attiya and Welch show that, if the `register` operations are performed very quickly, it is possible to shift processes enough so that the linearization order (and hence the responses returned by processes) must be different, and yet the executions are indistinguishable to the processes because the shift is "hidden" by the uncertainty in message delays.

Several papers used similar shifting arguments to give lower bounds on the implementations of various objects in fault-free partially synchronous message-passing systems. Mavronicolas and Roth [240] proved lower bounds for implementations of linearizable and sequentially consistent `registers` when process clocks are imperfectly synchronized. In their model, each process's clock runs at the same rate, but they may differ from one another by a bounded amount. They also assumed that operations simulated on one `register` do not affect the way operations on other `registers` are simulated, and that each `register` is implemented in the same way. Kosa [212] compared the costs of satisfying the different consistency conditions for implementing objects that have operations satisfying certain algebraic properties, for example, operations that do not commute with one another. Lower bounds are given for sequentially consistent implementations with perfectly synchronized process clocks, for linearizable implementations with clocks that run at the same rate but are not synchronized, and for implementations that satisfy hybrid consistency using perfect clocks. Friedman [144] gave lower bounds for implementations of RMW objects and `queues` satisfying **weak consistency**, where, for each process P, there must be a linear order of all operations that gives the same responses and agrees with the actual order of P's operations. Lower bounds for the other consistency models mentioned above follow as corollaries.

James and Singh [190] studied differences in the resilience of implementations of `registers` in an asynchronous message-passing system, where the implementations must satisfy different consistency conditions. For example, in a linearizable implementation, one cannot guarantee that either read or write operations will terminate if half the processes may fail. However, for sequentially consistent implementations, one can guarantee that either the read operations or the write operations (but not both) will be completed correctly in the presence of any number of failures.

Higham and Kawash [182] studied the mutual exclusion problem using `registers` that satisfy Goodman's version of **processor consistency** [15]. This means that, given an execution, it is possible to find, for each process P, a linear ordering of P's reads and the writes of every process that preserves the order of any two operations done by the same process and gives the same results for each of P's reads. Furthermore, the order of the writes to each `register` must be the same in all of the linear orderings. Higham and Kawash proved that any mutual exclusion algorithm for $n > 1$ processes must use at least one (multi-writer) `register` and n `single-writer registers`. A corollary is that (multi-writer) `registers` cannot be implemented from `single-writer registers` in a model that only supports this consistency condition.

6.4 Fault models

One can consider models where the communication mechanisms have varying degrees of reliability. Although it is easier to build systems that have weaker reliability guarantees, they are much harder to programme. This tension between ease of implementation and ease of use can be resolved if reliable versions of the communication media can be implemented (efficiently) in software, in a system where the unreliable versions are provided. This approach is not possible in some cases, for example, when a certain problem is solvable using the reliable model but not using the unreliable one.

Unreliable communication channels cannot be used to implement reliable message delivery in many cases. This can be formalized by considering the **sequence transmission problem**, also called the **end-to-end communication problem**, where a sender has a sequence of messages that it wants to transmit to a receiver. In the simplest case, the sender and receiver are connected by an unreliable channel, along which both can send packets. If the channel crashes, no information can be transmitted. Packet loss, duplication and reordering can all be tolerated using sequence numbers [289]. Unfortunately, sequence numbers cause the length of packet headers to grow without bound. However, Wang and Zuck [294] showed that any protocol tolerating both packet reordering and duplication requires unbounded sequence numbers. Afek, Attiya, Fekete, Fischer, Lynch, Mansour, Wang and Zuck [7] proved that any protocol tolerating both packet reordering and loss either requires unbounded sequence numbers or can cause the receiver to receive an unbounded number of packets per message. When there is a fixed bound on the size of packet headers and there is a fixed probability that any packet is delayed, Mansour and Schieber [237] proved lower bounds on the number of packets that have to be sent as a function of the number of messages, matching known upper bounds [7]. The alternating bit protocol [57,215] uses a single-bit packet header: the least significant bit of the sequence number. It tolerates both packet loss and duplication. Fekete and Lynch [130] proved that there are no headerless protocols that tolerate packet losses, so the alternating bit protocol uses the shortest possible headers. When the sender and the receiver can fail by losing the information stored in their local memories, Fekete, Lynch, Mansour and Spinelli [131] proved that no protocol can tolerate packet losses, even for an easier version of the problem where the sequence of messages can be output by the receiver in any order. For more general networks where packet losses can occur, Adler and Fich [5] proved lower bounds on the size of packet headers necessary for sequence transmission when intermediate nodes do not store information. Their lower bound is logarithmic in the number of simple paths between the sender and receiver for many networks, including the complete graph, series-parallel graphs and fixed degree meshes, matching the upper bounds of an existing algorithm [122]. All of these lower bounds are proved using adversary arguments. More specifically, an adversary constructs two executions that are indistinguishable to the receiver, but in which the sequence of messages to be transmitted is different.

It would be useful to automatically translate algorithms that tolerate crash failures into algorithms that tolerate the same number of more serious faults. For synchronous message-passing systems, the natural approach is to simulate the original algorithm in a round-by-round manner. For fully connected networks, Neiger and Toueg [256] gave a round-by-round simulation of algorithms tolerating crash failures for models in which faulty processes can fail to send some of their messages. The resulting algorithms use twice as many rounds as the original algorithms. A similar simulation is possible for models in which f faulty processes can fail to send or receive messages, provided $n > 2f$. They also proved that, for this model, a round-by-round simulation is impossible if $n \leq 2f$ and the simulation must simulate the faulty processes as well as the non-faulty ones. Bazzi and Neiger [58] showed the simulation is possible if it is not necessary to simulate the internal states of the faulty processes accurately, but the simulation must increase the time complexity by a factor of approximately $2f/(n-f)$. They did this by giving a lower bound on the complexity of solving a broadcast problem in a way that tolerates omission faults. Finally, one can consider the problem of translating to a model where arbitrary process faults can occur. Such a translation is impossible if $n \leq 3f$, since consensus can be solved in the presence of f crash failures, but not when f arbitrary process faults can occur. (See Section 5.3.) For $n > 3f$, Bazzi and Neiger [58] gave upper and lower bounds showing that the translation increases the running time by a factor of two, three or four depending on the exact value of f.

Jayanti, Chandra, and Toueg [196] showed how a 1-faulty system of $m + 2$ simulators that communicate using `registers` can simulate a 1-faulty system of two threads and m objects, in which one *object* may fail by delaying its responses forever. The actions of each thread and object are simulated by a different simulator. To simulate an operation on an object by a thread, the simulator of the thread writes the operation to a `single-writer single-reader register`, which is read by the simulator of the object. The simulator of the object simulates the operation in its local memory and returns its response to the simulator of the thread via another `single-writer single-reader register`. Using this simulation, they prove that wait-free consensus for two processes is unsolvable in an asynchronous shared-memory system in which one object may fail. Specifically, if such an algorithm does exist and uses m objects, then $m + 2$ simulators can solve 1-resilient consensus using the simulation, by having the output value from any completed thread written into a `register`, enabling all simulators to return that value. As discussed in Sect. 5.2, this is known to be impossible.

Stomp and Taubenfeld [290] gave a space lower bound for implementing reliable `test&set` objects from unreliable ones.

6.5 Other separations between models

Chaudhuri and Welch [96] gave time and space lower bounds on the complexity of implementing multi-bit `registers` from single-bit `registers`. Jayanti, Sethi and Lloyd [197] established lower bounds on the space complexity of implementing regular N-bit `single-writer single-reader registers` from linearizable single-bit `single-writer single-reader registers`. (A **regular** `register` is weaker than a linearizable one: the value returned by a read operation can be any one of the values that is stored in the `register` at some time between the beginning and end of the read [220].) Valois [293] studied lower bounds on the number of read-modify-write objects needed to implement `load-linked/store-conditional registers`, in which a process's write operation succeeds in updating the state of the `register` only if the state has not changed since that process's last read operation. Schenk [281] showed that approximate agreement can be solved faster using (multi-writer) `registers` than using `single-writer registers`. This result is described in Sect. 11.2.

Merritt and Taubenfeld [243] studied the differences in the power of systems equipped with objects that satisfy different fairness guarantees. Sakamoto [277] studied how the power of a system depends on the knowledge that processes have, initially, about the network to which they belong, for example, its size.

7 Deciding when problems are solvable

Because there are so many different models of distributed systems, it would be useful to have a general technique to determine whether a given model can solve a given problem, or implement a given data structure. Unfortunately, this is not possible in general.

7.1 Undecidability

Jayanti and Toueg [200] proved that there is no algorithm that, given the description of an object type and an initial state, determines whether it has a wait-free implementation from `registers`. They use a reduction from the halting problem. Given a deterministic Turing machine M, they construct a type $\mathtt{T}(M)$ whose state stores a configuration of M and a Boolean flag. The object is initially in a state corresponding to M's initial configuration on a blank input tape and the Boolean flag is false. The type $\mathtt{T}(M)$ is equipped with a single operation. The operation updates the configuration stored in the state by simulating one step of M and returns 0 as long as M has not halted. The first operation applied to $\mathtt{T}(M)$ after the simulated machine M has halted sets the flag to true and returns 1. Any operation on $\mathtt{T}(M)$ after the flag is set returns 2.

If M halts on a blank tape, then $\mathtt{T}(M)$ can be used to solve wait-free leader election (and hence consensus) for two processes: each process repeatedly accesses the object until it returns a non-zero value and the process that receives the value 1 becomes the leader. This means `registers` cannot implement $\mathtt{T}(M)$. However, if M never halts on a blank input tape, then $\mathtt{T}(M)$ can be implemented using a `register` initialized to 0 and having each operation applied to $\mathtt{T}(M)$ replaced by a read of this `register`. It follows that one cannot decide whether the type $\mathtt{T}(M)$ can be implemented from `registers`. A similar construction can be used to show that determining the consensus number of a given finitely-specified type is undecidable.

Naor and Stockmeyer [253] showed that it is undecidable whether a task can be solved in a message-passing system in constant time in bounded-degree networks, although it is decidable whether a problem can be solved within a given number of steps.

Further undecidability results are described in Sect. 8.3.

7.2 Decidability of consensus numbers

For some natural classes of types, decision procedures for consensus numbers do exist. They follow from theorems that characterize types in the class in terms of their consensus number.

One such class consists of the read-modify-write (RMW) object types [213], defined in Sect. 2. Ruppert [274] gave a characterization of the RMW types that can solve wait-free consensus among n processes. The characterization uses a restricted form of the consensus problem, called **team consensus**, where processes are divided into two teams and all processes on the same team receive the same input. A RMW type `T` has consensus number at least n if and only if there is an algorithm for solving team consensus among n processes in which every process performs exactly one step on an object of type `T`. A valency argument was used to show the necessity of this condition: by examining the behaviour of processes as they each take their first step after the critical configuration of a consensus algorithm, one can obtain the required one-step algorithm for team consensus. For finite types, this condition is decidable.

A similar characterization was also given for **readable** types [274], which allow processes to read the state of the object without changing it. Together, these two classes of objects contain many of the most common shared-memory primitives. These characterizations were used to prove unsolvability results for consensus in **multi-object** models [273] (where processes can access more than one shared object in a single atomic action), following the work of Afek, Merritt and Taubenfeld [11].

Recently, Herlihy and Ruppert [178] gave a characterization of deterministic one-shot types that can solve wait-free consensus among n processes. A **one-shot** type is one that can be accessed only once by each process. This characterization, too, is decidable for finite types. A partial characterization is also given for non-deterministic types. (See Sect. 9.2 for more on this result.)

A natural open question is to obtain an algorithm that decides the consensus number of any type with finite state set. An interesting special case would be to consider non-deterministic RMW and readable types. One might also be able to gain a better understanding of the relative power of different types by characterizing the types that can solve other problems such as set consensus or implementing an object that can be used repeatedly to solve different instances of consensus.

7.3 Characterizing solvable tasks

The preceding section describes attempts to characterize the models that can solve a particular important problem. Another way of systematically studying solvability is to characterize the problems that can be solved in a particular model.

In the asynchronous message-passing model, Biran, Moran and Zaks [61], building on earlier work by Moran and Wolfstahl [249], gave a combinatorial characterization of the decision tasks that can be solved 1-resiliently in an asynchronous message-passing system with crash failures. This characterization, described below, is in terms of the task's vectors of input and output values (with one coordinate of the vectors corresponding to each process).

Suppose there is a 1-resilient message-passing algorithm to solve a given task for n processes. Let $G(\boldsymbol{x})$ denote the set consisting of all output vectors produced by the algorithm with input vector $\boldsymbol{x}$. First, for each input vector $\boldsymbol{x}$, Biran, Moran and Zaks consider the **similarity graph** with vertex set $G(\boldsymbol{x})$ and edges between any two vectors that differ in exactly one coordinate. They prove this similarity graph is connected using a valency argument with slightly different definitions: a configuration C is univalent if all executions from C lead to an output vector in the same connected component and multivalent otherwise. Secondly, they show that, if I is any set of input vectors that differ only in coordinate j, then there is a set of output vectors, one from $G(\boldsymbol{x})$ for each $\boldsymbol{x} \in I$, that differ only in coordinate j. This follows from consideration of those executions in which process P_j is non-faulty, but takes no steps until all other processes have produced an output.

Conversely, suppose there is a task for n processes such that there is a set $G(\boldsymbol{x})$ of allowable output vectors for each input vector $\boldsymbol{x}$, which has the following two properties: the similarity graph with vertex set $G(\boldsymbol{x})$ is connected, and if I is a set of input vectors that differ only in coordinate j, then there is a set of output vectors, one from $G(\boldsymbol{x})$ for each $\boldsymbol{x} \in I$, that differ only in coordinate j. Then Biran, Moran and Zaks proved that there is a 1-resilient message-passing algorithm to solve the task. In later papers, they also showed that determining whether a task has these properties is NP-hard for more than two processes [62], and gave very precise bounds on the round complexity of solving any task that satisfies them [63].

Taubenfeld, Katz and Moran [291] gave a game-theoretic characterization of the tasks that can be solved in a message-passing asynchronous system with $f < n/2$ crash failures that can only occur before processes take any steps. They consider tasks with and without termination requirements. They define a two-player game, where one player represents the algorithm and the other represents an adversarial scheduler. In each round of the game, the adversary chooses the input values for some additional processes and the algorithm must choose output values for those processes consistent with the output values chosen in previous rounds. The algorithm loses if no such choice is possible. The task is solvable if and only if the player representing the algorithm has a winning strategy in this game. The characterization is constructive: a winning strategy for the game yields an algorithm for the task.

Chor and Moscovici [100] gave characterizations of tasks that can be solved in asynchronous models by f-resilient randomized algorithms that never produce an incorrect output. Their characterizations apply to message-passing systems for $f < n/2$ and to the shared-memory model where processes communicate using `registers`. Although their results are not phrased in terms of game theory, the characterizations are similar to those described in the previous paragraph. In fact, it follows from the characterizations in these two papers that a task is solvable by a deterministic message-passing algorithm tolerating f crash failures which occur before processes take any steps if and only if it is solvable by a randomized algorithm tolerating f crash failures [291].

Attiya, Gorbach and Moran [39] gave a simple characterization of the tasks that are solvable in systems where asynchronous processes have no names, run identical programmes, do not know how many processes are in the system, and communicate using `registers`. The characterization (and the proof of its necessity) is similar in flavour to the results by Biran, Moran and Zaks, described above.

In an **interactive task**, each process receives a sequence of input values and must produce the output value corresponding to its current input value before being given its next input value. A task is specified by giving the legal sequences of input and output vectors. Chor and Nelson [101] considered asynchronous systems in which consensus is solvable and characterized the interactive tasks that can be solved in these systems. Their conditions ensure, among other things, that the set of allowable outputs does not depend on the input values which have not yet been received. Herlihy's universality result [169] does not imply that all interactive tasks can be solved, since the definition of an interactive task is quite general, and there are some interactive tasks that cannot be viewed as a problem of implementing a linearizable object.

8 Topology

Perhaps the most interesting development in the theory of distributed computing during the past decade has been the use of topological ideas to prove results about computability in fault-tolerant distributed systems. The results described in this section include some powerful applications of topology in distributed computing, particularly for proving impossibility results. Other connections between topology and distributed computing have been discussed in the literature (see [152,157, 204,268]).

8.1 Simplicial complexes

We begin with some brief definitions of ideas from the topology of simplicial complexes. Several papers contain good introductions to the connections between distributed computing and simplicial complexes [145,173,176].

A d-dimensional **simplex** (or d-simplex) is a set of $d+1$ independent **vertices**. Geometrically, the vertices can be thought of as (affinely) independent points in Euclidean space. A 0-simplex is a single point, a 1-simplex is represented by a line segment, a 2-simplex is represented by a filled-in triangle, and so on. A (simplicial) **complex** is a finite set of simplexes closed under inclusion and intersection. The **dimension** of a complex is the maximum dimension of any simplex that appears in it. Examples of simplicial complexes appear in Fig. 4.

A vertex can be used to represent the internal state (or part of the internal state) of a single process. A d-simplex whose vertices correspond to different processes represents compatible states of $d+1$ processes. As an example, consider the binary consensus problem for three processes, P, Q and R. The possible starting configurations of an algorithm for this problem are shown in Fig. 4(a). Each vertex is labelled by a process and the binary input value for that process. The complex consists of

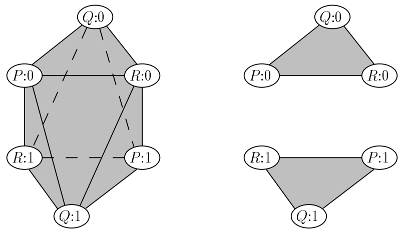

Fig. 4. (a) Input complex and (b) output complex for three-process binary consensus

eight 2-simplexes arranged to form a hollow octahedron. Each 2-simplex represents one of the eight possible sets of inputs to the three processes. The corresponding output complex in Fig. 4(b) shows the possible outputs for the binary consensus problem. In the upper 2-simplex, all processes output value 0, while in the lower 2-simplex, all processes output value 1. Not all output simplexes are legal for every input simplex: by the validity condition of consensus, if all processes start with the input value 0, then only the upper output simplex is legal.

More generally, any decision task for n processes can be modelled in a similar way. The input complex I contains one $(n-1)$-simplex for each possible input vector. The output complex O contains one simplex for each possible output vector. A map Δ that takes each simplex S of I to a set of simplexes in O (labelled by the same processes) defines which output vectors are legal for each input vector.

Simplicial complexes are used as a means of describing whether processes can distinguish different configurations from one another. In that sense, they are similar to, though more general than, the similarity graphs of Biran, Moran and Zaks [61] discussed in Sect. 7.3. Nodes in those graphs correspond to $(n-1)$-simplexes. The situation where two output vectors differ in only one coordinate, which is modelled by an edge in a similarity graph, is represented in the complex by having the two simplexes share $n-1$ common vertices. Complexes can capture more information about the degree to which two configurations are similar: two simplexes that have d common vertices are indistinguishable to exactly d processes. This fact makes complexes useful for studying f-resilient algorithms for any f, whereas similarity graphs are useful primarily when $f = 1$.

Consider a wait-free algorithm for n processes that solves some task. One can define a corresponding $(n-1)$-dimensional **protocol complex**. Each vertex is labelled by a process and the state of that process when it terminates in some execution. Given any input vector and any schedule for the processes (as well as a description of the results of any coin tosses or non-deterministic choices), the final state of every process is determined. This final configuration is represented by a simplex in the protocol complex.

Each process must decide on an output value for its task based solely on its internal state information at the end of the algorithm. This defines a decision map δ that takes each vertex of the protocol complex to a vertex of the output complex (labelled by the same process). Let S be a simplex of the protocol complex. Since S represents a configuration of compatible final states for some set of processes, $\delta(S)$ must be a simplex of the output complex, representing a compatible set of outputs for those processes. Furthermore, δ must "respect" the task specification: If S represents a configuration reached by some execution whose inputs come from the simplex I of the input complex, then $\delta(S)$ must be in $\Delta(I)$.

The basic method of proving impossibility results using the topological approach can now be summarized. One uses information about the model to prove that any protocol complex has some topological property which is preserved by the map δ. The specification of the task is used to show that the image of δ cannot have the property, implying that no such map δ can exist.

For example, it can be shown that, in the asynchronous model where processes use `registers` to communicate, any protocol complex (that begins from a connected input complex) is connected [179]. The connectivity property is preserved by any map δ, since δ maps simplexes to simplexes. As shown in Fig. 4, the input complex for three-process binary consensus is connected. The image of δ must include vertices in both triangles of the output complex, since the task specification requires that, for any run where all processes get the same input value v, all processes output v. Thus the image of δ is disconnected, and hence wait-free three-process binary consensus is impossible in this model.

8.2 Set consensus results

Much of the inspiration for the early topological impossibility results came from Chaudhuri [92], who defined the k-set consensus problem. She observed that Sperner's Lemma [287], a tool often used in topology, could be applied to study the task. In papers that first appeared at STOC in 1993, Borowsky and Gafni [69], Herlihy and Shavit [179] and Saks and Zaharoglou [279] independently proved, using versions of Sperner's Lemma, that $k+1$ processes cannot solve wait-free k-set consensus in an asynchronous model using `registers`. In addition to proving the unsolvability of set consensus, these papers developed interesting techniques that led to proofs of other results showing connections between distributed computing and topology. This is a great example of the important role that lower bounds for a well-chosen problem can play in opening up new areas of research. Similar tools have also been used to provide lower bounds for the set consensus problem in a synchronous message-passing model [93]. Attiya reproved the impossibility of set consensus using more elementary tools [34].

Borowsky and Gafni's impossibility proof [69] used the protocol complex for $k+1$ processes, in the case where each process uses its process name as its input to the set consensus problem. They introduced the **immediate snapshot model**. In this model, processes communicate using a `single-writer snapshot` object. Although processes run asynchronously, there are restrictions placed on the adversarial scheduler. They showed that this model can be simulated by the asynchronous model where processes communicate via `registers`; the opposite simulation is trivial. They restrict attention to full-information algorithms, where each process repeatedly scans the `snapshot` object and updates its element by appending the result of the scan. With these simplifications of the model, Borowsky and Gafni showed that protocol complexes have a very regular form. This allowed them to apply a variant of Sperner's Lemma to prove that, for some simplex of any protocol complex, each of the $k+1$ processes outputs a different value. Using the BG simulation technique, described in Sect. 6.1, they extended the unsolvability result for wait-free k-set consensus to the f-resilient setting, for $f \geq k$.

The impossibility proof by Saks and Zaharoglou [279], which uses point-set topology, has a different flavour from the other results described in this section. They use a simplified model similar to that of Borowsky and Gafni, and consider the space of all (finite and infinite) schedules. They defined a set of schedules to be open if they can be recognized by some algorithm (i.e. there is an algorithm where at least one process eventually writes "accept" if and only if the execution is following a schedule in the set). They proved that this collection of open sets defines a topology on the space of all schedules. Now, suppose a k-set consensus algorithm exists for $k+1$ processes (where each process has its name as its input). Then the set D_i of schedules in which some process outputs the value i is an open set, and it does not contain any schedule where i does not take any steps. These facts can be used, together with a version of Sperner's Lemma, to show that there is a schedule contained in $\cap_i D_i$. In this schedule, the processes output $k+1$ different values, which contradicts the correctness of the algorithm. An interesting direction for future research is to investigate the structure of this topological space of schedules. Perhaps theorems from point-set topology could then be applied to prove other results in distributed computing.

8.3 The Asynchronous Computability Theorem

The third paper that proved impossibility of k-set consensus has since been developed into a more general result that characterizes the tasks that can be solved in a wait-free manner using `registers`. Herlihy and Shavit [179] proved that a task is solvable if and only if it is possible to subdivide the simplexes of the input complex into smaller simplexes (with any newly created vertices being appropriately labelled by processes) that can then be mapped to the output complex. This mapping μ must satisfy properties similar to those of a decision map δ. It must preserve the process labels on the vertices, map simplexes to simplexes, and it must respect the task specification: if a simplex I of the input complex is subdivided into smaller simplexes, the smaller ones must all be mapped to simplexes in $\Delta(I)$. This characterization is called the Asynchronous Computability Theorem. It reduces the question of whether a task is solvable to a question about properties of the complexes defined by the task specification. A key step in proving the necessity of the condition is a valency argument that shows the protocol complexes in this model contain no holes. Although the condition used in the characterization is not decidable (see below), the theorem provides insight into the set of solvable tasks and can be used to study particular tasks. For example, to prove the impossibility of set consensus, Herlihy and Shavit show (using Sperner's Lemma) that no mapping μ can exist for the set consensus task. The paper also gives results on the impossibility of renaming.

Gafni and Koutsoupias [147] used the Asynchronous Computability Theorem to show that it is undecidable whether a given task has a wait-free solution using `registers`, even for finite three-process tasks. They use a reduction from a problem known to be undecidable: **loop contractibility**, where one must decide whether or not a loop on a 2-dimensional simplicial complex can be contracted, like an elastic band, to a point while staying on the surface of the complex. Suppose the input complex (for three processes) is simply a 2-simplex. Given a loop on a 2-dimensional output complex, they define a task that requires the boundary of the input simplex to be mapped to the loop by the function μ of the Asynchronous Computability Theorem. This map μ can be extended to the whole (subdivided) input simplex if and only if the loop can be contracted. This undecidability result contrasts with Biran, Moran and Zaks's characterization of tasks that can be solved 1-resiliently, which is decidable for finite tasks [61] (see Sect. 7.3). Herlihy and Rajsbaum [174] extended the undecidability result to other models.

Havlicek [167] used the Asynchronous Computability Theorem to identify a condition that is necessary for a task to have a wait-free solution using `registers`. This condition is computable for finite tasks.

Several researchers have presented characterizations similar to the Asynchronous Computability Theorem. These alternative views give further insight into the model, and the proof techniques are quite different in some cases. Herlihy and Rajsbaum [172] showed how to prove impossibility results in distributed computing using powerful ideas developed in the homology theory of simplicial complexes. They discussed models where the shared memory consists of `registers`, or of `registers` and `set consensus` objects. They reproved impossibility results for the set consensus problem, and gave some new results for the renaming problem. Attiya and Rajsbaum [44] used purely combinatorial arguments to develop a characterization of tasks solvable using `registers`, similar to the Asynchronous Computability Theorem. In particular, they showed that the protocol complexes for a simplified model have a very regular form.

Borowsky and Gafni [71] gave an elegant proof of a version of the Asynchronous Computability Theorem without using topological arguments. They introduced the **iterated immediate snapshot model** and prove that it is capable of solving the same set of tasks as the ordinary `register` model. They proved the equivalence of the models by giving algorithms to simulate one model in the other [70,71]. The protocol complex of a (full-information) algorithm in their simplified model is a well-understood subdivision of the input complex. Thus, a problem is solvable in either model if and only if there is a decision map from a subdivision of this form to the output complex that respects the task specification.

Hoest and Shavit [185] used topological techniques to determine the time complexity of approximate agreement in a generalization of Borowsky and Gafni's iterated immediate snapshot model. Essentially, they related the time complexity of the task to the degree to which the input complex must be subdivided before one can map it to the output complex. For this problem, the number of subdivisions required can be computed very precisely. Although, in terms of computability, their model is equivalent to the standard asynchronous model containing only `registers`, their complexity results do not carry over. Much work remains to find additional ways of applying topology to prove complexity lower bounds.

8.4 Solving tasks with other types

Herlihy and Rajsbaum [171] undertook a detailed investigation of the topology of set consensus. They gave conditions about the connectivity of protocol complexes that are necessary for the solution of set consensus. They also described connectivity properties of the protocol complexes in a model where the primitive objects are `set consensus` objects and `registers`. They later used this work to give a computable characterization of tasks that can be solved f-resiliently in various models that allow processes to access `consensus` or `set consensus` objects [174]. The characterization uses topological tools but also builds on the characterization given by Biran, Moran and Zaks (see Sect. 7.3) for systems using only `registers`.

Herlihy and Rajsbaum [175] also considered an interesting class of decision tasks, called **loop agreement** tasks [174]. Using topological properties of the output complexes, they describe when one loop agreement task can be solved using `registers` and a single copy of an object that solves a different loop agreement task.

Herlihy, Rajsbaum and Tuttle [177] used the topological approach to give unified proofs of impossibility results for set consensus in several message-passing models with varying degrees of synchrony. The key idea in this paper is that protocol complexes for several of the commonly studied models can be represented as unions of one type of simple building block.

A desirable goal is a better understanding of the structure of protocol complexes for different models. The protocol complexes tend to be quite complicated, but to obtain impossibility results, it is often sufficient to prove that they have certain properties, without fully describing their form. Restrictions on the adversarial scheduler can also simplify the structure of protocol complexes, making them easier to study while simultaneously strengthening any lower bounds obtained. Most of the research has focused on one-shot objects or tasks; extensions of these techniques to long-lived objects is a subject of current research.

9 Robustness

The consensus number of an object type provides information about the power of a system that has objects of that type and `registers`. However, the classification of individual types into the consensus hierarchy does not necessarily provide complete information about the power of a system that contains several different types of objects: it is possible that a collection of weak types can become strong when used in combination. The hierarchy is **robust** (with respect to a class of object types) if it is impossible to obtain a wait-free n-process implementation of a type at level n of the hierarchy from a finite set of types that are each at lower levels. Robustness is a desirable property since it allows one to study the synchronization power of a system equipped with several types by reasoning about the power of each of the types individually.

This issue of robustness was first addressed by Jayanti [192,194]. His impossibility results, discussed in Sect. 9.1, were instrumental in clarifying the definition of consensus number. Jayanti [191] provides a good description of early work on the robustness question. Work on this topic has produced some very interesting proofs, bringing together ideas discussed in Sect. 5, 6, 7 and 8.

9.1 Non-robustness results

A variety of non-robustness results have been proved during the past decade. Typically, one defines a pair of objects that are tailor-made to work together to solve consensus easily. To complete the proof, one must show that each of the types, when used by themselves, cannot solve consensus.

Recall that the definition of the consensus number of a type `T` is the maximum number of processes that can solve wait-free consensus using any number of objects of type `T` and `registers`. Some of the early results on the robustness question showed that the hierarchy is not robust under slightly different definitions of consensus number. Cori and Moran [107] and Kleinberg and Mullainathan [208] showed the hierarchy is not robust if only one object of type `T` (and no `registers`) can be used. Jayanti proved non-robustness results for the case where one object of type `T` and any number of `registers` can be used [192], and for the case where any number of objects of type `T` (but no `registers`) can be used [194]. These results are, in part, responsible for the choice of the now-standard definition of consensus number.

One of Jayanti's proofs [192] used an interesting simulation technique. He defined a simple object type, called `weak-sticky`. It is possible to solve wait-free consensus among n processes using `registers` and $n-1$ `weak-sticky` objects, but it is not possible using fewer `weak-sticky` objects. He proved this by giving an implementation of `weak-sticky` from `registers` that is not wait-free but has the property that at most one operation on the object will fail to terminate. We illustrate his lower bound proof for the case where $n = 3$. Suppose there is a wait-free consensus algorithm for three processes that uses a single `weak-sticky` object and `registers`. One could replace the weak-sticky object by the implementation from `registers`. Then, if at most one process can fail and at most one process's operation on the weak-sticky object fails to terminate, at least one non-faulty process can complete the algorithm and write its output in shared memory. All non-faulty processes can then return that value. This yields a 1-resilient consensus algorithm for three processes using only `registers`, which is impossible (see Sect. 5.2). The use of "imperfect" implementations to prove impossibility results is also discussed in Sect. 6.1.

The robustness question is somewhat more complicated for the standard definition of consensus numbers: although the hierarchy is not robust for all object types, the robustness property does hold for some classes of objects.

A number of objects have been constructed that violate the robustness property [85,86,245,263]. The objects used in these constructions allow the response to an operation to depend on the identity of the process that invoked the operation.

Schenk [282,283] proved that the consensus hierarchy is not robust by considering a type with unbounded non-determinism, i.e. an operation may cause an object to choose non-deterministically from an infinite number of possible state transitions. In this case, he said an algorithm is wait-free if the number of steps taken by a process must be bounded, where the bound may depend on the input to the algorithm. For objects with bounded non-determinism, this definition of wait-freedom is equivalent to the usual requirement that every execution is finite. Lo and Hadzilacos [227] improved Schenk's result by showing that the hierarchy is not robust even when restricted to objects with bounded non-determinism.

Schenk defined two types, called `lock` and `key`. The `key` object is a simple non-deterministic object that can easily be used to solve the **weak agreement** problem: All processes must agree on a common output value and, if all processes have the same input value, the output value must differ from it. He used a combinatorial argument to show that, for any consensus algorithm using `keys` and `registers`, there exists a fixed output value for each `key` which is consistent with every execution. This allows all the `key` objects to be eliminated, which is impossible, unless $\text{cons}(\texttt{key}) = 1$.

The `lock` object was specially constructed to provide processes with a solution to the consensus problem if and only if processes can "convince" the object that they can solve weak agreement. The `lock` object non-deterministically chooses an instance of the weak agreement problem and gives this instance to the processes as a challenge. It then reveals the solution to the original consensus problem if and only if processes provide the `lock` object with a correct solution to the challenge. (The idea of defining an object that only provides useful results to operations when it is accessed properly, in combination with another type of object, was originated by Jayanti [194] and is common to many of the non-robustness proofs.) If processes have access to both a `lock` and `key` object, they can use the `key` to solve the `lock`'s challenge and unlock the solution to consensus. Thus, $\text{cons}(\{\texttt{lock}, \texttt{key}\}) = \infty$. Schenk used a type of valency argument developed by Lo [226], to show that weak agreement and, hence, consensus for two processes cannot be solved using only `locks` and `registers`.

9.2 Robustness results

Although the consensus hierarchy is not robust in general, the practical importance of the non-robustness results is unclear, since the objects used in the proofs are rather unusual. The hierarchy has been shown to be robust for some classes of objects that include many of the objects commonly considered in the literature.

A `consensus` object is an object that is specifically designed to solve consensus. It supports one operation, propose, which has one argument (an input value to the consensus problem being solved). All propose operations to the same `consensus` object in an execution return the same value (the output value), which must be one of the values proposed. An m`-consensus` object is a restricted form that allows only m propose operations to be performed. Chandra, Hadzilacos, Jayanti and Toueg [86] showed that m`-consensus` objects cannot be combined with any objects that have consensus number n to solve consensus among more than $\max(m, n)$ processes.

Ruppert [274] showed that the consensus hierarchy is robust for the class of all RMW and readable types. The proof uses the characterization of the types that can be used to solve consensus for n processes, described in Sect. 7.2. It is easy to show, using a valency argument, that any consensus algorithm for n processes built from such objects must use one object whose type satisfies the conditions of the characterization. Therefore, n-process consensus can be solved using only that type and `registers`.

Herlihy and Ruppert [178] used the topological approach to characterize the one-shot types that can be combined with other types to solve consensus. (A one-shot object is one that can be accessed at most once by each process.) The characterization applies to non-deterministic types, provided the number of possible responses to any invocation is finite. They defined a kind of connectivity property, called **k-solo-connectivity**, of the simplicial complexes associated with the invocations that can be invoked on an object of type `T` and the responses that the object can return. They showed that there is a type `B` such that $\text{cons}(\texttt{B}) < k$, but $\text{cons}(\{\texttt{T}, \texttt{B}\}) \geq k$ if and only if `T` is not k-solo-connected. The key tool in showing the "only if" direction of the proof is a simulation technique that builds on the BG simulation (see Sect. 6.1). Assuming `T` is k-solo-connected, the simulation allows a k-process consensus algorithm built from objects of types `T`, `B` and `register` to be simulated using only objects of type `B` and `registers`. This means that if $\text{cons}(\{\texttt{T}, \texttt{B}\}) \geq k$ then $\text{cons}(\texttt{B}) \geq k$ too. The other direction is a generalization of the non-robustness results [227, 283] described above. It follows from Herlihy and Ruppert's characterization that the class of deterministic one-shot types is robust.

The robustness result for readable and RMW types used two important properties: such objects are deterministic and their state information can be accessed in some simple way by each process. The robustness result for one-shot types used a similar property: when accessing a one-shot object, a process gets, in a single operation, all of the state information that it will ever be able to obtain directly from the object. Can robustness results be extended to other natural classes of types that do not have these kinds of properties? By finding the line that separates those types that can be combined with others to violate robustness from those that cannot, we gain insight into the way that types behave when used together in complex systems.

10 Space complexity lower bounds

Some lower bounds on space complexity are concerned with the number of shared objects required to solve a problem. Others tell us how large the shared objects must be.

One early space lower bound is due to Rabin [265]. He considered choice coordination among two choices, each of which is identified with one read-modify-write object, and proved that no deterministic asynchronous algorithm can solve this problem unless the objects have at least $\sqrt[3]{n}/2$ different possible states. His proof uses combinatorial and graph-theoretic techniques to construct a bad execution.

Burns, Jackson, Lynch, Fischer, and Peterson [79] considered deterministic solutions to the mutual exclusion problem using one shared object. They proved lower bounds on the size of the object for different fairness conditions. For algorithms with lockout freedom, the object must have $\Omega(\sqrt{n})$ different states. A stronger condition, bounded bypass, requires n different states. Since they gave an algorithm with lockout freedom that uses $n/2 + O(1)$ states, this shows that bounded bypass is a strictly stronger condition. They also gave algorithms with bypass bound 2 that use $n + O(1)$ states. The lower bound shows that these algorithms for mutual exclusion with bounded bypass are optimal to within a small additive constant. However, for lockout freedom, matching upper and lower bounds for space complexity are not known.

The proof of the lower bound for bounded bypass uses the pigeonhole principle: if the object has an insufficient number of states, there are two configurations in which different sets of processes want to access the resource, yet the configurations are indistinguishable to the processes in one set. Then it is possible to construct an execution from one of these configurations in which processes from this set enter the critical section many times, but no other processes do. Hence, from the other configuration, there is an analogous execution in which some other process that wants the resource is forced to wait. Versions of this proof appear in both [234, Theorem 10.41] and [47, Theorem 4.3]. Similar ideas can be used to prove the $\Omega(\sqrt{n})$ lower bound for lockout freedom [234, Theorem 10.44].

Burns and Lynch [77,234] introduced covering arguments to prove that any mutual exclusion algorithm for $n \geq 2$ processes that communicate using `registers` uses at least n `registers`, no matter how large the `registers` are. First they consider the situation where some process P begins to want the shared resource and is given steps exclusively until it obtains the resource. They show that, during this sequence of steps, P must write to some `register` not covered by any other process; otherwise they can construct an execution in which P and some other process have the shared resource simultaneously. Next, they show how to construct executions which result in configurations that have successively more covered `registers`, yet are indistinguishable (to the other processes) from configurations in which no process has or wants the shared resource. Their lower bound is optimal, matching the number of `registers` used by known mutual exclusion algorithms [216,219].

Fich, Herlihy, and Shavit [136] considered a very weak termination condition, **non-deterministic solo termination**: at any point, if all but one process fails, there is an execution in which the remaining process terminates. In particular, wait-free and randomized wait-free algorithms satisfy non-deterministic solo termination. They proved that $\Omega(\sqrt{n})$ `registers` are needed by any asynchronous algorithm for n-process consensus that satisfies this property. They began by considering the case where processes are anonymous. They showed that, if the number of processes is sufficiently large compared to the number of `registers`, it is possible, using clones, to construct an execution in which both 0 and 1 are output. A clone of a process P starts in the same initial state as P and runs in lock step with P until it covers a certain `register` that P writes to. In their proof, they constructed multivalent configurations from which there is an execution by one set of processes that output 0 and an execution by a disjoint set of processes that output 1. These executions both begin by having processes write to the `registers` they cover. If the processes in one of these executions only write to the `registers` covered by the other set of processes, these executions can be combined to obtain an execution in which some processes output 0 and some output 1. Otherwise, one can obtain another such multivalent configuration in which more `registers` are covered. For non-anonymous processes, this idea is combined with a new method of cutting and combining executions. Because consensus is a decision task, the proof is more difficult than for mutual exclusion, where processes can repeatedly request exclusive access to the critical section. Although there are algorithms for randomized wait-free consensus among n processes that use $O(n)$ `registers` of bounded size [28], it remains open whether this is optimal.

Fich, Herlihy, and Shavit extended their lower bound to algorithms using **historyless objects**, objects whose state depends only on the last non-trivial operation applied to them. (Some examples of historyless types are `register`, `swap`, and `test&set`.) They showed that $\Omega(\sqrt{n})$ historyless objects are necessary for randomized wait-free implementations of objects such as `compare&swap`, `fetch&increment`, and bounded `counters` in an n-process system. Jayanti, Tan and Toueg [198] used a simpler covering argument to prove that $n-1$ historyless objects are necessary for randomized wait-free implementations of objects such as `compare&swap`, `fetch&increment`, and bounded `counters`. In addition to historyless objects, their lower bound applies to `resettable consensus` objects. Such an object can be used to repeatedly solve different instances of consensus through the use of a reset operation that puts the object back to its initial state.

Attiya, Gorbach and Moran [39] used the covering technique to prove lower bounds in a fault-free, asynchronous, anonymous system, where processes communicate via `registers`. They showed that $\Omega(\log n)$ shared `registers` (and $\Omega(\log n)$ rounds) are required for n processes to solve consensus in this model. Moran, Taubenfeld and Yadin [247] also used a covering argument to prove that any wait-free implementation of a `mod m counter` for n processes from objects with only two states must use at least $\min(\frac{n+1}{2}, \frac{m+1}{2})$ such objects.

The non-robustness proofs of Kleinberg and Mullainathan [208] and of Jayanti [192], discussed in Sect. 9.1, include space lower bounds which show that wait-free consensus is impossible using a small number of objects of certain types. Kleinberg and Mullainathan [208] also considered objects which can be read and pairs of which can have their contents swapped. Using graph theory together with some indistinguishability arguments, they proved that to solve consensus among n processes using these objects and `registers`, at least $n+1$ of these objects are necessary.

Saks, Zaharoglou and Cori [278] studied the problem of implementing a list that contains one element for each process and supports two operations: report, which returns the entire list, and move-to-front, which moves the element belonging to the process to the front of the list. Such a list might be used for scheduling decisions based on the order in which the processes started executing their current jobs. They proved that if the only objects available for the implementation are one `single-writer multi-reader register` per process, then the average size of each is $\Omega(\log^2 n)$ bits. The proof makes use of the fact that, in a system of n processes, if the element belonging to some process is at the end of the list and that process takes no steps, the algorithm behaves like a solution to the problem for $n-1$ processes. This observation leads to a recurrence that bounds the space needed by n processes in terms of the space needed by $n-1$ processes. They also show that there is a matching upper bound when no operations are concurrent with any move-to-front operation.

Much work remains to obtain space complexity lower bounds for other problems and in models with more powerful objects.

11 Time complexity lower bounds

Time complexity is usually measured by the number of rounds required for a system to solve a problem, or by the number of steps an individual process must take. This section surveys time complexity lower bounds for a variety of problems.

11.1 Agreement problems

The first significant lower bound on time complexity for a distributed problem was proved for terminating reliable broadcast in a synchronous message-passing model where up to f arbitrary process faults can occur. Using a chain argument, Fischer and Lynch [139] showed that at least $f+1$ rounds are required. The bound also holds in models where processes can use cryptographic primitives to authenticate messages [110, 120], although the chain arguments are more complicated. In fact, Dolev, Reischuk, and Strong [119] proved that every algorithm has executions with at least $\min(b+2, f+1)$ rounds in which $b \leq f$ arbitrary process faults actually occur.

When only crash failures occur, Hadzilacos [159] and Lamport and Fischer [222] were able to prove that $\min(f+1, n-1)$ rounds are necessary for solving f-resilient terminating reliable broadcast, even if at most one process can crash in each round. A version of the proof, which uses a chain argument, is given in [234, Sect. 6.7] and [47, Sect. 5.1.4] to prove that $\min(f+1, n-1)$ rounds are needed for consensus in a synchronous message-passing system with f crash failures. The chain is constructed implicitly and is very long. Babaoğlu, Stephenson and Drummond [54] proved similar lower bounds for solving terminating reliable broadcast using broadcast primitives that are prone to faults.

Aguilera and Toueg [14] and Moses and Rajsbaum [250] used valency arguments to obtain considerably simpler proofs of the lower bound on the number of rounds for consensus. At each round, to ensure multivalence, the adversary causes one process to crash. This can continue only so long as there remains a process to crash.

Dolev, Reischuk, and Strong [119] used a chain argument to prove that, for simultaneous terminating reliable broadcast, if up to f arbitrary process faults can occur, then at least $f+1$ rounds are necessary, even for the execution in which no faults occur. Using a knowledge-based approach, Dwork and Moses [126] studied simultaneous consensus in the synchronous message-passing model with crash failures. They gave matching upper and lower bounds on the number of rounds needed to achieve simultaneous consensus in an execution in terms of the pattern of failures that occur in the execution. Their lower bound applies even to the weak version of the problem, where validity is required only when no faults occur. Interestingly, they showed that if validity is replaced by a nontriviality condition that requires that there is an execution in which processes decide 0 and an execution in which processes decide 1, the problem can be solved in two rounds. However, this algorithm does not extend to the stronger model in which at most one process can crash in each round.

Moses and Tuttle [251] extended Dwork and Moses' bounds on the number of rounds for obtaining simultaneous consensus to a large class of related problems and to models in which faulty processes can fail to send or receive some of their messages. When all faulty processes only fail to send certain messages or when all faulty processes only fail to receive certain messages, they showed that processes can determine when to decide using a polynomial amount of local computation. However, if faulty processes can fail to send some messages and fail to receive some messages, they proved that, unless $P = NP$, there is no algorithm for these problems in which processes can decide at the earliest possible round in every execution using only a polynomial amount of local computation.

Inoue, Moriya, Masuzawa, and Fujiwara [187] considered the problem of clock synchronization in synchronous shared-memory systems with `single-writer registers`. Any number of processes may fail to take a step in a round (and so do not advance their clocks). Their requirement is that those processes which have been non-faulty for sufficiently many rounds must agree exactly on the value of their clocks. They proved that at least $n-1$ rounds are required from the time a process last failed to take a step until its clock value is guaranteed to agree. To do so, they considered two executions that are indistinguishable to some process for $n-2$ rounds, but in which that process must have different clock values to be correct. They also gave an algorithm in which all processes that have taken steps for $\Omega(n)$ consecutive rounds will agree on their clock values.

Attiya, Dwork, Lynch, and Stockmeyer [37] used valency arguments to give lower bounds on the time required to solve consensus in a partially synchronous message-passing model with crash failures, where messages are delivered within time d and there is a bound, r, on the ratio of process speeds. They proved that the worst-case running time of an f-resilient consensus algorithm is at least $(r+f-1)d$. Attiya and Djerassi-Shintel [36] used similar techniques to prove an $\Omega(\frac{rdn}{n-f})$ lower bound for the consensus problem in a partially synchronous model where up to f processes experience timing faults: they continue to execute the algorithm, but may not satisfy the timing assumptions of the model. They also proved lower bounds for set consensus and renaming in this model. Alur, Attiya and Taubenfeld [18] considered wait-free consensus in partially synchronous models where processes can experience crash failures and communicate using `registers`. They proved that each process requires $\Theta(\frac{U \log U}{\log \log U})$ time, where U is an (unknown) upper bound on the time between steps of a process. To do this, they carefully assign times, consistent with the parameter U, to the steps of any sufficiently long asynchronous execution.

Afek and Stupp [12] obtained complexity results from computability results by using a simulation. They considered the problem of electing a leader in a system of n processes using `registers` and one `compare&swap` object that can store one of v different values, and proved that some process must take $\Omega(\log_v n)$ steps. Given such a leader election algorithm in which no process ever takes more than d steps, they showed how $\lfloor n/(d+1) \rfloor$ simulators can simulate the n threads, using only `registers`, and thereby solve $(v-1)^d$-set consensus. In the simulation, different simulators may actually simulate different executions of the leader election algorithm. However, the number of different simulated executions is at most $(v-1)^d$, so the simulators will output at most $(v-1)^d$ different values. The lower bound on d follows from the fact that `registers` alone cannot solve set consensus when $(v-1)^d < \lfloor n/(d+1) \rfloor$ (see Sect. 8.2).

11.2 Approximate agreement

Attiya, Lynch, and Shavit [41] proved that any wait-free approximate agreement algorithm with convergence ratio at most $1/2$, for n processes that communicate using `single-writer registers` (of unbounded size), has a failure-free execution in which no process decides the value of its output before round $\log_2 n$. They do this by obtaining an upper bound on the number of processes that can influence the state of a particular process during the first t rounds of a round-robin execution. Then they show that if a process P is not influenced by another process P', it cannot decide; otherwise, there is another execution which is indistinguishable to P in which P' runs to completion before the other processes begin and outputs an incompatible value.

For wait-free two-process approximate agreement using `registers`, Herlihy [168] used a valency argument to prove that a process has to take $\Omega(\log \frac{1}{\rho})$ steps in the worst case, where ρ is the convergence ratio. Schenk [281] proved that any algorithm for this problem using objects of consensus number one must have a chain of at least $1/\rho$ different final configurations, where adjacent configurations are indistinguishable to one of the processes. From this result, he used a combinatorial argument to obtain a tradeoff between the number of steps taken by a single process and the number of rounds in the execution: If the worst-case number of steps is at most k, then the worst-case number of rounds is at least $\Omega(\log_{2k} \frac{1}{\rho})$. For `single-writer registers`, this tradeoff was first obtained by Attiya, Lynch, and Shavit [41]. Schenk used a different combinatorial argument to prove that any wait-free approximate agreement algorithm for n processes that uses b-bit `registers` must take $\Omega((\log \frac{1}{\rho})/b)$ rounds and use $\Omega((\log \frac{1}{\rho})/b)$ `registers`. The proof considers the amount of information a process needs to determine its output value after all the other processes have decided.

Schenk also gave an algorithm using 1-bit `registers` that matches his lower bounds for $b = 1$. Together with Attiya, Lynch, and Shavit's $\log_2 n$ lower bound for `single-writer registers`, this implies that any wait-free implementation of `registers` from `single-writer registers` has round complexity $\Omega(\log n)$. However, there is a large gap between this lower bound and the best known implementation [164]. It also remains open whether approximate agreement can be solved in fewer rounds using larger `registers`.

As described in Sect. 8.3, Hoest and Shavit [185] used the topological approach to study the complexity of solving approximate agreement in the iterated immediate snapshot model.

Dolev, Lynch, Pinter, Stark and Weihl [117] considered approximate agreement in message-passing models with f arbitrary process faults. Specifically, they obtained lower bounds for the convergence ratio of synchronous one-round algorithms and asynchronous single-phase algorithms, in which processes share their input values and then, based on this information, choose their output values. Fekete [132] gave lower bounds on the convergence ratio of any synchronous algorithm, as a function of the number of rounds, for crash failures and arbitrary process faults. Fekete [133] also gave a lower bound of $\lceil (n-f)/f \rceil^{-r}$ on the convergence ratio for r-round algorithms in an asynchronous message-passing model where up to f faulty processes could omit sending some of their messages. Note, if $f \geq n/2$, this says that the convergence ratio is at least 1, implying that approximate agreement is impossible. Plunkett and Fekete [264] gave a lower bound on the convergence ratio for single-round algorithms that tolerate a variety of faults.

11.3 The complexity of universal constructions

Herlihy's universality result [169], discussed in Sect. 3, and subsequent papers [8,90,170,200], provide **universal constructions**, which automatically give a wait-free distributed implementation of any object type, using sufficiently powerful shared-memory primitives. Jayanti [193,195] has studied some of the limitations of this approach to providing implementations. He showed that a process that performs a wait-free simulation of an operation using a universal construction requires $\Omega(n)$ steps of local computation in the worst case, where n is the number of processes [193]. This bound does not depend on the nature of the communication between processes, and even holds in an amortized setting. The key idea in the proof is the design of an object type that conspires with the scheduler to reveal as little information about the behaviour of the object as possible. This ensures that each process, simulating a single operation *op*, must do some computation for each simulated operation that precedes *op*. The bound is tight [169].

Jayanti [195] also proved a lower bound of $\Omega(\log n)$ on the number of shared-memory operations that must be performed by a universal construction in the worst case. This bound applies to a shared-memory model that has quite powerful primitive types of objects and holds (for expected complexity) even if randomization is permitted. Jayanti proved the bound by considering the wakeup problem and studying how information propagates through the system. Roughly speaking, each shared-memory operation at most doubles the size of the set of processes that are known (by some process or memory location) to have woken up. This lower bound is also tight [8].

11.4 Counting

In the **counting** problem, processes atomically increment a `counter` and get the current value of the `counter` in response. Counting is useful for sharing resources, implementing timestamps, and other applications.

Moran and Taubenfeld [246] studied counting modulo m in the context of asynchronous systems equipped with single-bit read-modify-write objects. Even for a very weak correctness requirement, they show that m must be a divisor of 2^T, where T is the worst-case running time of the increment operation. It follows that the running time is at least $\log_2 m$, and also that the problem is impossible if m is not a power of two. The key idea of the proof is to create executions where, for some $t < T$, 2^t processes are hidden: the execution cannot be distinguished by the other processes from one where the hidden processes take no steps. Whenever one hidden process flips a `bit`, another process flips it back before any non-hidden process can see it. If processes are to return the correct answer, it must be that m divides 2^t, so that processes do not need to distinguish the two runs. Similar ideas were used by Moran, Taubenfeld and Yadin [247] to obtain space lower bounds. (See Sect. 10.)

Herlihy, Shavit and Waarts [180] proved that the number of steps to execute increment is $\Omega(n/c)$, where c is the maximum number of processes that access any particular object in any execution of the algorithm. They looked at the sequences of objects accessed by different processes in certain runs and showed that these sequences must contain common objects to avoid having both processes return the same value. It follows that a sequence must contain at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ objects to ensure that it intersects the sequence of every other process. Dwork, Herlihy and Waarts [124] obtained a tradeoff for consensus using a similar proof.

11.5 Other time complexity lower bounds

Building on their earlier work [61] (see Section 7.3), which characterized the tasks that can be solved in an asynchronous message-passing model with at most one crash failure, Biran, Moran, and Zaks [63] gave tight upper and lower bounds on the number of rounds needed for tasks that are solvable in this model.

Chaudhuri, Herlihy and Tuttle [94] showed that $\Omega(\log n)$ rounds are required in a synchronous message-passing system for any wait-free comparison-based algorithm to break symmetry. As an algorithm runs, a process may get information about the identifiers of other processes in the system and store this information in its local state. For comparison-based algorithms, two processes will behave in the same way as long as their identifiers have the same relative order compared to each of the identifiers they have learned about. In each round, they showed how an adversary can kill a constant fraction of the remaining processes to ensure that all surviving processes remain in indistinguishable states. This establishes a general $\Omega(\log n)$ lower bound for problems, such as leader election, that require different processes to perform different actions.

Peleg and Rubinovich [261] showed that the time required to construct a minimum spanning tree in a synchronous message-passing system is $\Omega(\sqrt{n}/(B \log n))$, where B is the maximum number of bits that can be transmitted in a single message. Their result holds for networks of diameter $\Omega(\log n)$. Lotker, Patt-Shamir and Peleg [229] later showed that an $\Omega(\sqrt[3]{n}/B)$ bound holds even for graphs of diameter 4 using similar techniques. Both results are proved by considering a specific network of small diameter where the construction of a minimum spanning tree requires a large amount of information to be transmitted from one part of the network to another. Although there is a short path that connects the two parts (since the diameter is small), the bound on message size ensures that transmitting all the information along this path would take a long time. Transmitting the information along other paths also takes a long time, since those paths are much longer.

Several papers have studied the time complexity of wait-free implementations of `snapshot` objects, which are convenient for programmers to use, from `registers`, which are more commonly provided in real systems. Israeli and Shirazi [188] considered the implementation of a `single-writer snapshot` object from `single-writer registers` and showed that $\Omega(n)$ steps are required to update an element. Specifically, given any implementation with small enough update time, they show how an adversary can construct two indistinguishable executions that each contain a scan which should return different results. Jayanti, Tan and Toueg [198] used a covering argument to show that $\Omega(n)$ steps are needed for scans, even when the implementation only has to be non-blocking and can use `registers`. Fatourou, Fich and Ruppert [128,129] considered the problem of implementing a `snapshot` object with m elements in a wait-free manner from `multi-writer registers`. When $n > m + 1$, at least m `registers` are required. They used covering arguments to prove that, for any implementation that uses this many `registers`, the scan requires $\Omega(mn)$ steps in the worst case. This bound is tight. They also proved that $\Omega(\sqrt{mn})$ steps are required to perform a scan in an implementation that uses any number of `single-writer registers` in addition to the m `registers`.

In some applications, processes typically execute their operations when no other processes are running. Although algorithms are required to be correct regardless of how many processes run simultaneously, it may be that the important measure of performance is **contention-free** complexity, the maximum complexity over all possible solo executions. Alur and Taubenfeld [19] gave the first lower bounds for contention-free time complexity. They proved that any asynchronous algorithm solving mutual exclusion using only `registers` has a solo execution that takes $\Omega(\frac{\log n}{b+\log\log n})$ steps, where b is the number of bits per `register`. They showed, using a combinatorial argument, that if all solo executions take fewer steps, then there are two processes whose solo executions are similar enough that they can be interleaved without either process noticing the presence of the other, so that both processes enter the critical section at the same time. There do exist mutual exclusion algorithms using $O(\log n)$-bit `registers` that have $O(1)$ contention-free complexity [221], matching the lower bound for $b \in \Theta(\log n)$. Alur and Taubenfeld also showed that some solo execution must access $\Omega(\sqrt{\frac{\log n}{b+\log\log n}})$ different `registers`. In the same paper, they gave simple lower bounds on the contention-free complexity of solving the renaming problem, using different types of single-bit read-modify-write objects.

In some systems, shared objects are stored at different processes and a process can access locally stored shared objects at a significantly lower cost than remotely stored shared objects. Anderson and Yang [21] proved a necessary tradeoff for mutual exclusion between the number of accesses to remote shared objects performed by processes and the **contention** (the maximum number of processes that can simultaneously access the same object). The proof uses a combinatorial argument that allows an adversary to construct a long execution if the contention is always low. Anderson and Kim [23] have improved this result to obtain a lower bound on time complexity (that does not depend on the contention). Specifically, they prove that in a system of n processes, some process must perform $\Omega(\log n / \log\log n)$ critical events (for example, certain accesses to `registers` or `compare&swap` objects) to enter and exit its critical section. This is very close to the known $O(\log n)$ upper bound. Furthermore, there are a number of significant technical difficulties in removing the dependence on contention that make the proof very interesting.

12 Message and bit complexity lower bounds

In systems where processes communicate by sending messages to one another, the total number of messages and the total number of bits transmitted by an algorithm are useful measures of the algorithm's complexity. In this section, we describe lower bounds on the message and bit complexities of various problems.

12.1 Computation in a ring

Many of the early message complexity lower bounds were proved expressly for ring networks, where processes are arranged in a circle. Two properties of rings are used to prove strong lower bounds. Firstly, $\Omega(n)$ messages must be sent to transmit information from a process to the diametrically opposite process. Secondly, the high degree of symmetry of a ring can be exploited to show strong lower bounds on message complexity: before the symmetry is broken, whenever a process sends a message, many other processes must do likewise. Thus, although a ring is a very simple network, and hence suitable for lower bound proofs, it is not so simple that problems become easy to solve. This makes the ring a good candidate when one is looking for the worst-case network topology for a problem, and a good starting point for lower bound proofs.

There are a number of modelling issues to be considered when studying ring lower bounds: the processes may operate synchronously or asynchronously, communication may be bidirectional or unidirectional, processes may either have distinct identifiers or be anonymous, and processes may or may not initially know the size of the ring. In the case of a bidirectional ring, processes may or may not initially know which direction is clockwise. For each of these five choices, the latter option makes problems harder (or at least no easier) to solve, so lower bounds for the former automatically apply to the latter.

In a unidirectional ring, one crash failure of a process or communication channel makes it impossible for some processes to communicate information to processes further along the ring. Similarly, in a bidirectional ring, two crash failures can disconnect the network. Thus, when studying rings, one generally assumes that the system is reliable.

Burns [78] showed a worst-case lower bound of $\Omega(n \log n)$ messages for leader election among n processes in an asynchronous bidirectional ring, if processes do not initially know n, but do have distinct identifiers. See [47, Sect. 3.3.3] for a discussion of his inductive proof, which generates a costly execution for n processes by pasting together a pair of costly executions for $n/2$ processes that have no communication across one edge. Bodlaender [64] proved, using a combinatorial argument, that the $\Omega(n \log n)$ bound holds even for the average message complexity (taken over all assignments of identifiers to processes) when processes know the ring size and have distinct identifiers drawn from a set of size $(1+\epsilon)n$, for $\epsilon > 0$. He also showed how to extend this result to randomized algorithms. Pachl [258] gave a similar proof of an $\Omega(n \log n)$ lower bound on expected message complexity for randomized algorithms that allow a constant probability of error on a unidirectional ring of unknown size, where processes have distinct identifiers.

Higham and Przytycka [184] used a weighted measure of message complexity, where some edges are more expensive to use than others (cf. [52]). Using this complexity measure, they obtained a generalization of the lower bound for leader election in an asynchronous, unidirectional ring when processes have distinct identifiers.

The **solitude verification** problem is to determine whether there is exactly one process with input value 1. It is closely related to the leader election problem. It can be used to verify that a leader has been elected and, in a unidirectional ring with a leader, solitude verification can be performed with $O(n)$ bit transmissions. Hence, $\omega(n)$ lower bounds for solitude verification imply the same lower bounds for leader election.

When the ring size is only known to within a factor of 2, i.e. $N \leq n \leq 2N$, for some N, and processes have distinct identifiers chosen from a universe of size $s \geq 2N$, Abrahamson, Adler, Gelbart, Higham, and Kirkpatrick [2] proved that the average (over all possible assignments of identifiers to processes) expected bit complexity of randomized solitude verification on asynchronous unidirectional rings is in $\Omega(n \log(n/s))$. Their proof uses the pigeonhole principle combined with a scenario argument. They also provided a leader election algorithm with $O(n \log s)$ expected bit complexity for unknown ring size, matching their lower bound when $s \geq n^{1+\epsilon}$ for some constant $\epsilon > 0$. If processes are anonymous, but ring size is known to within a factor of 2, they gave another randomized leader election algorithm that uses $O(n \log n)$ expected bits of communication. They showed that this algorithm is optimal by proving a lower bound of $\Omega(n \log \Delta)$ when $N \leq n \leq N + \Delta$ for some N and each identifier can be assigned to at most two processes.

Abrahamson, Adler, Higham, and Kirkpatrick [4] obtained tight bounds for the bit complexity of solitude verification on asynchronous unidirectional rings of anonymous processes allowing randomization and a constant probability of error. Their bounds do not depend on whether the error is one-sided or two-sided or whether deadlock can occur, but do depend on the processes' knowledge of the ring size and whether the processes must know when the algorithm has terminated. For example, if the ring size n is known and all processes must output the answer, they proved that any algorithm solving solitude verification with error probability $\epsilon > 0$ has expected bit complexity $\Theta(n \min(\sqrt{\log n}, \sqrt{\log\log(1/\epsilon)}))$. They used sophisticated scenario and symmetry arguments to obtain their lower bounds on the expected number of bits transmitted. Their technique considered a configuration having exactly one process with input value 1 and for which, with high probability, only a small number of bits are transmitted. They showed that there must be a short sequence of bits that are transmitted along a particular link with reasonably high probability. Then by removing, replicating, and splicing parts of the ring, they were able to construct another configuration having more than one process with input value 1 on which the algorithm errs with unacceptably high probability.

A more general problem related to leader election is for processes to compute a function of their input values. Once a leader has been elected, that leader can initiate a message which will travel around the ring gathering all the inputs. When the message returns to the leader, it computes the function locally, and then distributes the answer to the other processes using $n-1$ additional messages. On the other hand, leader election can be solved by computing the maximum, using the (distinct) process identifiers as input values and electing the process with the largest input value. There are deterministic algorithms for computing the maximum of the input values in an asynchronous unidirectional ring using $O(n \log n)$ messages [116,262], so leader election has the same complexity. Thus having randomness or bidirectional links does not help to elect a leader in an asynchronous ring where processes have distinct identifiers.

To compute a function on a ring of anonymous processes, the function value must be the same when the inputs are cyclically permuted around the ring. Attiya, Snir and Warmuth [46] studied the problem of computing a Boolean function of input values in an anonymous asynchronous bidirectional ring of known size, where processes begin in identical states and do not know which direction is clockwise. Any function that is computable in this model can be computed using $O(n^2)$ messages: every process sends its input around the ring. They showed most functions that are computable in this setting do require $\Theta(n^2)$ messages, using a symmetry argument. Duris and Galil [123] gave an $\Omega(n \log^* n)$ bound on the number of messages needed for any non-constant function. Moran and Warmuth [248] showed this bound is tight for some functions and proved that the bit complexity must be $\Omega(n \log n)$ for any non-constant function. They also constructed a family of non-constant functions that can be computed within these bounds. Their lower bound on bit complexity was generalized by Bodlaender, Moran and Warmuth [65] to apply to networks where processes have distinct identifiers. For randomized computation on a unidirectional anonymous ring, Abrahamson, Adler, Higham, and Kirkpartrick [3] proved that $\Omega(n\sqrt{\log n})$ bits must be transmitted for computing any non-constant function and showed that this bound is tight for some functions. Their lower bound holds even when algorithms may become deadlocked or fail to terminate with high probability: They show that an erroneous computation can be obtained by cutting and splicing parts of an accepting computation in which too few bits are transmitted. Attiya and Snir [45] gave an $\Omega(n \log n)$ bound on the number of messages for computing, in an anonymous ring, a function (such as exclusive or) whose output value is not determined by a short subsequence of the input values. Their bound applies to the average, over all inputs, of the number of messages sent in the worst-case execution for that input. They also give related lower bounds for the expected complexity of randomized algorithms. Similar results are known for non-anonymous rings [64].

If the function to be computed has the range $\{0, 1\}$, we can think of the problem of computing the function as a problem of **recognizing strings in a language**, namely those that produce the output value 1. Each process begins with one character of the input string (in order around the ring). Mansour and Zaks [238] studied this problem in a ring of unknown size with a distinguished leader. The obvious algorithm which has the leader send one message all the way around the ring to gather the entire string uses $\Theta(n)$ messages, which is clearly optimal for non-trivial languages. However this algorithm transmits messages containing a total of $\Theta(n^2)$ bits. Mansour and Zaks studied whether the bit complexity can be improved. They gave an elegant proof that the bit complexity of the problem is $O(n)$ if and only if the language is regular. If the language is not regular, $\Omega(n \log n)$ bits are necessary, which establishes an interesting gap.

Lower bounds for synchronous rings generally rely on symmetry arguments. A highly symmetric configuration of an algorithm is one that contains many processes in the same state. The first step of the proof is to carefully design one or more initial configurations that have a high degree of symmetry (for example, by choosing the identifiers so that comparison-based algorithms cannot tell apart sections of the ring). One then shows that, in order to solve a problem, processes must reduce the level of symmetry by communicating with one another. The message complexity lower bounds take advantage of the fact that, whenever a message is sent by one process, it is also sent by all other processes in the same state.

Frederickson and Lynch [143] showed that any comparison-based leader election algorithm in a bidirectional synchronous ring requires $\Omega(n \log n)$ messages. For simplicity, consider the case where n is a power of 2. They designed an assignment of identifiers to processes so that, for any segment of the ring of length 2^i, there are $n/2^i$ segments that look identical to any comparison-based algorithm. Consider a process in one such segment and another process in the corresponding location in another segment. These two processes will be in identical states until information is propagated to one of them from outside its segment. If a process wishes to propagate information to its neighbour that will be useful in distinguishing the two segments, it must either send a message to its neighbour, or it can remain silent while the corresponding process in the other segment sends a message to its neighbour. Frederickson and Lynch keep track of the longest distances across which information has been propagated in this way. Doubling this quantity from 2^i to 2^{i+1} requires messages to be sent in 2^i rounds. However, each time one message is sent, it will be sent simultaneously by $n/2^{i+1}$ other processes embedded in indistinguishable segments. This gives a total of $n/2$ messages to double the maximum distance that information has been propagated. The $\Omega(n \log n)$ bound follows from the fact that information must eventually be propagated at least halfway around the ring to ensure symmetry is broken and exactly one leader is elected.

Using Ramsey theory, Frederickson and Lynch extend their $\Omega(n \log n)$ lower bound from comparison-based algorithms to arbitrary time-bounded algorithms, provided the process identifiers are chosen from a sufficiently large set. They also give a leader election algorithm for a synchronous unidirectional ring that allows process identifiers to be arbitrary integers and sends only $O(n)$ messages, but can take a very large number of rounds. There is also a randomized leader election algorithm for anonymous synchronous rings with expected message complexity $O(n)$, provided processes know the ring size n [189].

Attiya, Snir and Warmuth [46] used similar ideas to prove that computing most functions in a synchronous, bidirectional, anonymous ring requires $\Omega(n \log n)$ messages. Attiya and Mansour [42] built on these results to prove lower bounds in this model on the number of messages required for recognizing strings in a language.

This section describes many $\Omega(n \log n)$ lower bounds for computation in rings. However, the lower bounds do not subsume one another, because of differences in models or in the measures of complexity used. The problem of unifying these results remains open: perhaps one strong lower bound for ring computation would imply all the others. Such a result would improve our understanding of the lower bounds.

12.2 Other network topologies

Consider the terminating reliable broadcast problem in a fault-free message-passing model, where n processes are arranged in a connected network with m message channels. Every process except the sender must receive at least one message in a fault-free execution, so at least $n - 1$ messages are necessary. If processes do not have any information about the topology of the graph, every edge must be traversed to ensure that there is no node "hiding" in the middle of the edge, so at least m messages are necessary. A simple flooding algorithm shows that m messages are sufficient. If each process knows the identities of its neighbours, depth first search uses $\Theta(n)$ messages. However, the messages transmitted may become very large: In order to ensure that the algorithm sends only one message to each node, a list of previously reached nodes is added to the message so that the algorithm can tell when it is unnecessary to forward the message to some of its neighbours. Awerbuch, Goldreich, Peleg and Vainish [53] analysed the message complexity of this problem for the case where messages must be of bounded length. Their lower bounds apply to synchronous systems, but they also give matching upper bounds for asynchronous networks, thereby establishing tight upper and lower bounds for both models. They showed that if each process knows the identities and arrangement of processes within a radius of ρ, then $\Theta(\min(m, n^{1+\Theta(1)/\rho}))$ messages are required. When $\rho = 1$, $\Theta(m)$ messages of bounded-size are required. Their lower bound is for comparison-based algorithms, but this restriction can be removed using Ramsey theory [233].

Korach, Moran and Zaks [210,211] considered complete networks of synchronous processes with no faults, where processes do not know which of their channels is connected to which process. They proved that $\Omega(n^2)$ messages are required to find a minimum spanning tree of the network, even if only edge weights 0 and 1 can occur. If an algorithm uses too few messages, they use a combinatorial argument to show that the weights on some the edges not used by the algorithm can be changed to obtain a network with a different minimum spanning tree. However, the algorithm cannot distinguish the new network from the original one. They also gave an $\Omega(n^2/k)$ bound for finding a spanning tree of degree at most k. Korach, Moran and Zaks defined the class of global problems and generalized their work to obtain lower bounds on message complexity for any problem in this class. A problem is **global** if, for any algorithm that solves the problem, each execution of the algorithm must use a set of edges that spans the network. Leader election, broadcast and the construction of a spanning tree are examples of global problems. Their proof is an adversary argument, where the adversary chooses the round at which each process begins execution and chooses which edge a process obtains when it tries to send a message across an unused incident edge. As the adversary constructs an execution, it keeps track of the size of the largest connected component in the subgraph induced by edges that have been used so far. Suppose at least $m(k)$ messages must be sent to obtain a component of size k. How many messages are required to get a component of size $2k + 1$? The adversary first has the algorithm construct two components of size k, using at least $2m(k)$ messages. If any process in one of the two large components then starts sending messages across unused edges, the adversary can direct the first k of them into the other large component. So at least $k+1$ messages must be sent before any vertex outside the two large components is discovered. Thus, a total of $2m(k) + k + 1$ messages are needed to construct a component of size $2k + 1$. The lower bound of $\Omega(n \log n)$ messages follows from the fact that all n processes must belong to the same component at the end of the execution. Korach, Moran and Zaks also provide a similar proof of an $\Omega(n^2)$ lower bound for certain matching problems.

Afek and Gafni [9] gave a tradeoff between time and message complexity for leader election. Singh [286] showed a stronger tradeoff between time and message complexity for asynchronous networks. Reischuk and Koshors [270] looked at global problems in arbitrary networks. They showed that if processes initially know the identifiers of only their immediate neighbours, $\Omega(m)$ bounded-size messages are required, where m is the number of edges in the network graph.

Goldreich and Sneh [151] studied the problem of computing a function of the processes' input values in an asynchronous system. The processes communicate using unidirectional channels which can experience crash failures, but it is assumed that the network remains strongly connected. They construct a graph with n vertices and m edges for which the problem of computing any function that depends on all of its inputs requires $\Omega(mn/\text{polylog}(n))$ messages. This means that the naive $O(mn)$ algorithm in which every process sends its input value to every other process by flooding the network is close to optimal. A combinatorial argument is used to show that, if every message that a process P can send causes fewer than $m/\text{polylog}(n)$ other messages to be sent around the network, the adversary could crash some edges of the graph to ensure that no new information ever gets back to P. Their proof continues by showing how an adversary can construct a bad schedule. The adversary first schedules one message from P that causes at least $m/\text{polylog}(n)$ other messages to be sent. When all of these messages have been delivered, the adversary allows P to send another message, which again must trigger $m/\text{polylog}(n)$ further messages. They show that this can be repeated $n/\text{polylog}(n)$ times.

Wattenhofer and Widmayer [295] studied the counting problem in an asynchronous message-passing environment. They consider executions where each of the n processes executes one increment operation. A centralized solution would have a single process keep track of the value of the counter. However, this requires one process to send and receive n messages during the execution. A solution where the work performed is more evenly distributed is desirable. They showed that it is not possible to create an algorithm where every process sends and receives $O(1)$ messages: in fact, even if no two increments are allowed to run concurrently, they show how to construct an execution in which one process will send or receive a total of $\Omega(\log n/ \log\log n)$ messages. The key observation used to obtain the lower bound is that each increment must find out about the previous increment via some chain of messages. The adversary greedily chooses which increment to schedule next so that this chain is long.

Dwork and Skeen [127] considered the commit problem in the synchronous network model where processes may experience crash failures. They showed that there must be a chain of messages from each process P to each other process Q in the failure-free execution where every initial value is 1. Otherwise, one can construct an execution where P's initial value is 0, but Q still decides 1, by killing any process as soon as it hears (directly or indirectly) from P. It follows that at least $2(n - 1)$ messages are required in the failure-free execution. This result was reproved using formal notions of knowledge by Hadzilacos [161]. Segall and Wolfson [284] generalized Dwork and Skeen's argument to give a lower bound on the number of message hops needed for solving the commit problem among a subset of processes in a network.

Amdur, Weber and Hadzilacos [20] studied the terminating reliable broadcast problem in a synchronous system where up to f of the processes may experience crash failures. In each round, each process may send a message to any set of processes. They showed that at least $\lceil (n + f - 1)/2 \rceil$ messages must be used in one of the failure-free executions, and this result is tight. Hadzilacos and Halpern [162] proved similar tight results for other kinds of faults. For the case of arbitrary process faults, they proved $\Theta(nf)$ messages are used in some failure-free execution. Dolev and Reischuk [118] gave a similar proof of this result. Both papers also derived bounds for the situation where processes may use authenticated messages to combat arbitrary process faults. The lower bounds also apply to the consensus problem because of the simple reduction mentioned in Sect. 3.3.

We illustrate the technique used for these lower bounds by proving the a similar bound for consensus in a system with arbitrary process faults [118,162]. Consider an algorithm for n processes that tolerates f faulty processes. Let E_i be the execution where all processes are correct and begin with input i. Suppose the total number of processes with which some process P communicates directly either in E_0 or E_1 is at most f. Consider an execution where those processes are faulty and P has input 0 while all other processes have input 1. If the faulty processes behave towards P as they do in E_0, and towards the rest of the processes as they do in E_1, P must decide 0, and the other correct processes must decide 1, violating agreement. Thus, for each process P, the set of processes with which P communicates either in E_0 or in E_1 must contain at least $f + 1$ processes. It follows that one of the two executions uses at least $n(f + 1)/4$ messages.

13 Randomized computation

Adding randomness to a model can increase its computational power. For example, it is possible to break symmetry by having processes flip coins. Thus an unsolvability result for a deterministic model that was proved using a symmetry argument might not be valid for a randomized model. Note that there will still be some execution where symmetry is never broken, namely the execution where all processes generate exactly the same infinite sequence of coin flips. However, the probability of this happening is 0. Thus, one might only require a randomized algorithm to terminate with probability 1 instead of requiring termination in all executions. A stronger condition requires that the expected time until termination is finite. Randomized consensus differs from consensus in exactly this way, making it easier: As discussed below, in asynchronous message-passing systems, randomized consensus can be solved by a randomized algorithm that tolerates up to $\lceil n/2 \rceil - 1$ crash failures, but consensus is unsolvable with just one crash failure. When considering randomized algorithms, correctness conditions can also be modified, allowing incorrect outputs, but requiring that the probability of such errors is very small. Gupta, Smolka and Bhaskar's survey [158] is a good introduction to the use of randomization in the design of distributed algorithms.

More powerful models combined with weakened problem specifications make proving impossibility results more difficult than for deterministic models. However, a variety of unsolvability results and lower bounds for randomized algorithms do exist. These are discussed in the remainder of this section.

13.1 Unsolvability results

An impossibility result for the terminating reliable broadcast problem in a randomized synchronous message-passing system was presented by Graham and Yao [153]. They considered algorithms for three processes, one of which may behave arbitrarily. In particular, the action of the faulty process at each round can depend on the messages sent by the other processes during that round. They showed that no algorithm can achieve agreement and validity with probability greater than $(\sqrt{5} - 1)/2$, where the probability is taken over the random choices made by the processes during executions chosen by an adversary. The proof is a detailed scenario argument, where the views of a process in two scenarios are indistinguishable in the sense that they are identical random variables. They also obtain an algorithm that achieves their bound.

Bracha and Toueg [75] considered the problem of solving consensus in an asynchronous message-passing system. As described in Sect. 5.1, this cannot be done using a deterministic algorithm that is resilient to even one crash failure. They showed an f-resilient randomized algorithm (which never errs and terminates with probability 1) exists if and only if $f < n/2$. We sketch a simpler proof of the unsolvability result. First consider the case of two processes, P and Q, one of which may fail. The standard type of valency argument shows the existence of an infinite execution, but this is insufficient to show that no randomized consensus algorithm can exist, since there may be infinite executions that occur with probability 0. Thus, modified definitions of valency are used for this proof: a configuration is solo-univalent if all terminating *solo* executions from C lead to the same decision value; otherwise, the configuration is solo-multivalent. Any initial configuration where processes have different inputs is solo-multivalent. To derive a contradiction, we now prove that no configuration of a consensus algorithm can be solo-multivalent. Suppose C is a solo-multivalent configuration. Then there exist solo executions σ_P by P and σ_Q by Q starting from C that lead to different decision values d_P and d_Q. Now consider an execution from C where P first executes σ_P (and decides d_P), and then Q executes σ_Q (and decides d_Q). By indefinitely delaying any messages sent during σ_P, one can ensure that this execution is legal. But this execution violates the agreement property of consensus algorithms. The result for n processes can be obtained by a reduction: if an $\lceil n/2 \rceil$-resilient algorithm for n processes exists, then a 1-resilient algorithm for two processes could be constructed by having P simulate $\lceil n/2 \rceil$ processes and Q simulate the other $\lfloor n/2 \rfloor$ processes. Bracha and Toueg [75] also showed that n-process consensus can be solved in this model in a way that tolerates f arbitrary process faults if and only if $f < n/3$.

Coan and Welch [105] proved there are no f-resilient algorithms for the commit problem in partially synchronous systems of $n \leq 2f$ processes, where non-faulty processes must eventually terminate with probability 1. They do this by explicitly constructing bad executions. They also prove that the expected number of steps taken by each process cannot be bounded. This proof uses a valency argument that allows the construction of many long executions leading to multivalent configurations.

Angluin [26] showed that if the ring size is known only within a factor of two, then even randomized algorithms cannot elect a leader in a synchronous, anonymous ring. Suppose there exists such an algorithm. Consider some terminating execution for a ring of n processes that occurs with probability $p > 0$. We construct an execution in a ring of $2n$ processes formed by cutting the original ring and pasting two copies together. With probability p^2, each pair of diametrically opposite processes will take exactly the same steps as the corresponding process in the n-process ring did. Since one process declared itself the leader in the n-process ring, two processes will do so at the end of the execution in the larger ring, which means the algorithm errs with non-zero probability. As mentioned in Sect. 5.4, there are randomized algorithms to elect a leader in an anonymous ring of known size. It follows that no randomized algorithm can determine the number of processes in an anonymous ring of unknown size. Cidon and Shavitt [102] used similar arguments to prove the impossibility of computing a large class of functions (including ring size and exclusive or) in an anonymous, synchronous ring of unknown size. Their proof applies even to randomized algorithms that terminate correctly with probability 1, provided they have bounded average message complexity.

Chor and Moscovici [100] characterized the tasks that are solvable by f-resilient randomized algorithms. (See Sect. 7.3.)

13.2 Complexity lower bounds

Randomized consensus is solvable in the asynchronous shared-memory model where processes communicate using `registers` and can flip coins. There are wait-free algorithms for randomized consensus among n processes that perform $O(n^2 \log n)$ expected work [74] and wait-free algorithms where the expected number of operations performed by each process is $O(n \log^2 n)$ [32]. Most algorithms for randomized consensus are based on collective coin flipping, which is a way of combining many local coin flips into a single global coin flip. However, there is a complication: a malicious adversary can destroy some of the local coins after they are tossed but before they are used. The goal of a collective coin flip algorithm is to limit the degree to which the adversary can influence the outcome of the global coin flip.

Aspnes proved that any f-resilient algorithm for randomized consensus performs $\Omega(f^2/\log^2 f)$ local coin flips (and, hence, work) with high probability [29]. His result applies to asynchronous shared-memory systems where processes communicate using only `registers`, as well as to all models that can be deterministically simulated by such a model, including asynchronous message-passing systems and asynchronous shared-memory systems with `snapshot` objects. The proof of his lower bound has two parts. One is a lower bound on the number of local coin flips needed to prevent an adversary from having too much influence on the outcome of a collective coin flip. The other is an extension of the valency argument to the randomized setting to show that an algorithm either performs a collective coin flip with small bias or spends lots of local coin flips to avoid doing so. Aspnes introduced the notion of an a-**univalent configuration**, a configuration from which an adversary scheduler can cause the algorithm to produce the output value a with sufficiently high probability. A **bivalent configuration** is both 0-univalent and 1-univalent and a **nullvalent configuration** is neither. He showed that, with high probability, an adversary scheduler can force any algorithm into a bivalent or nullvalent configuration from its initial configuration or whenever a local coin flip is performed. He also proved that a bivalent configuration always leads to a nullvalent configuration or to a configuration in which a local coin flip can be scheduled next. Finally, in nullvalent configurations, he showed that the coin flipping lower bound applies. A polylogarithmic gap remains between the upper and lower bounds for the amount of work to solve randomized consensus in asynchronous models.

Bar-Joseph and Ben-Or [56] extended Aspnes's result to synchronous message-passing systems, obtaining a lower bound of $\Omega(f/\sqrt{n \log n})$ rounds (with high probability) for f-resilient randomized consensus among n processes. They also gave a matching upper bound in this model. In contrast, for deterministic algorithms, $f + 1$ rounds are needed. (See Sect. 11.1.)

If the power of the adversary scheduler is restricted so that its choices can only depend on the actions of the processes (and cannot depend directly on the outcome of coin flips), then faster algorithms are possible: there is a randomized consensus algorithm using `registers` with expected $O(\log n)$ operations performed per process [49]. For message passing, even expected constant time randomized consensus algorithms have been achieved against weak adversaries, for asynchronous systems with omission faults [99] and synchronous systems with arbitrary process faults [134], tolerating a constant fraction of faulty processes. Chor, Merritt, and Shmoys [99] also obtained an upper bound on the probability of early termination in a synchronous message-passing system with crash failures. They did this by proving that, if the probability is too high, there is some input for which the chain argument used in the deterministic time lower bound applies.

Russell, Saks, and Zuckerman [276] proved lower bounds on the time to generate a shared coin flip that has a positive constant probability of being heads and a positive constant probability of being tails, even when arbitrary process faults can occur. The probability is taken over the random choices made by non-faulty processes. Specifically, they considered n-process synchronous algorithms with $r(n)$ rounds, where, at each round, each non-faulty process flips a fair coin and broadcasts the outcome to the other processes. They showed that, if $r(n) \leq (\frac{1}{2} - \epsilon) \log^* n$, for some constant $\epsilon > 0$, then the algorithm is not $\Omega(n)$-resilient. They also provided a tradeoff between the number of faulty processes and the number of rounds. These results are obtained by bounding the influence of random sets of variables on the value of Boolean functions. Collective coin flipping can be reduced to leader election using one additional round in which the chosen leader, if non-faulty, flips a fair coin and broadcasts it to all other processes. Thus their lower bounds also apply to leader election, where the probability of causing a faulty process to be leader must be bounded by some constant less than 1.

Coan and Dwork [104] obtained a lower bound on the number of rounds necessary for any randomized synchronous message-passing algorithm to solve simultaneous consensus when up to f processes can crash. More specifically, they proved that $f+1$ rounds are required in any execution in which at most r processes crash by round r, for all $r \leq f$. They did this by showing how to transform any randomized algorithm for this problem into a deterministic algorithm for the same problem and then applying results discussed in Sect. 11.1. They also obtained a similar lower bound for the distributed firing squad problem by reducing it to simultaneous consensus.

The **write-all** problem is to set n `registers`, all initially 0, to 1. It has been used for the construction of efficient wait-free algorithms [24, 156] and as the basis of step-by-step simulations of fault-free synchronous shared-memory systems by shared-memory systems with crash failures or asynchrony [24, 205, 239, 206]. Buss, Kanellakis, Ragde, and Shvartsman [81] proved that any randomized asynchronous algorithm for this problem that uses n processes, at most half of which can crash, must perform $\Omega(n \log n)$ writes to these `registers`. The idea of their proof is that an adversary schedules the processes to run until each covers one of the n `registers`. Among the `registers` with value 0, the adversary chooses the half which have the fewest processes covering them and schedules the $n/2$ or more processes which cover other `registers` to perform their writes. This can be repeated $\log_2 n$ times, each time reducing the number of `registers` with value 0 by at most a factor of 2. They provide a matching deterministic upper bound when processes can communicate using `snapshot` objects. Against restricted adversaries whose choices cannot depend directly on the outcome of coin flips, there is a randomized algorithm using `registers` that performs $O(n(\log n)^3)$ work with high probability [31]. For any

$\epsilon > 0$, there is also a deterministic algorithm using only registers that performs $O(n^{1+\epsilon})$ work [25]. However, it is an open question whether there is a deterministic algorithm for the write-all problem that uses only registers and performs $n(\log n)^{O(1)}$ total operations.

The write-all problem has also been considered in the synchronous shared-memory model with faulty processes. Kanellakis and Shvartsman [205] showed that the work complexity of the problem is $\Theta(n \log n / \log \log n)$ using snapshot objects and give an algorithm that does $O(n(\log n)^2 / \log \log n)$ work using only registers. Kedem, Palem, Raghunathan, and Spirakis [207] improved the lower bound using registers to $\Omega(n \log n)$ expected work for randomized algorithms. Martel and Subramonian [239] showed that this expected lower bound on work could be obtained using a restricted adversary. These lower bounds are obtained by reduction from the problem of computing the or of n bits. To prove a lower bound for computing or, they derived an upper bound (via a recurrence) on the number of processes at time t that are affected when some input bit is flipped, starting from the input configuration where all inputs are 0. Kanellakis and Shvartsman [206] give a more detailed discussion of the write-all problem and other related problems.

The randomized space complexity for the mutual exclusion problem was considered by Kushilevitz, Mansour, Rabin, and Zuckerman [214] when a single read-modify-write object is used to provide a fair randomized solution. They proved a lower bound on the size of the object. Specifically, $\Omega(\log \log n)$ bits are necessary. They also proved that their lower bound is tight, in contrast to the deterministic case, where $\Theta(\log n)$ are necessary, as discussed in Sect. 10. For weaker fairness conditions, they obtain smaller lower bounds that depend on the number of processes accessing the critical section in a mutually exclusive manner. Their proofs are based on an analysis of Markov chains.

Allenberg-Navony, Itai, and Moran [17] and Bodlaender [64] discuss circumstances under which lower bounds on the average-case complexity of deterministic distributed algorithms (where the average is taken over a probability distribution on the inputs) give rise to lower bounds on the expected complexity of randomized distributed algorithms. Their work extends Yao's minimax principle [299], which applies to sequential and parallel computation. Using this technique, they obtain lower bounds on the expected message complexity of finding the maximum process identifier in bidirectional rings of processes.

Several of the message complexity lower bounds for rings discussed in Sect. 12.1 apply to randomized algorithms.

14 Conclusions

This survey has presented many impossibility results in distributed computing and different techniques for proving them. This research topic has proved to be an interesting and fruitful one over the past two decades. Stronger impossibility results can be proved in distributed computing than in most other areas of computer science, because of the limitations imposed by local knowledge. We hope this survey will inspire and enable more people to contribute new impossibility results for distributed computing problems and new techniques for obtaining them.

Acknowledgements. We are indebted to Nancy Lynch for allowing us to use ideas and material from her survey paper [233]. We thank Maurice Herlihy and Vassos Hadzilacos for enlightening discussions. The anonymous referees provided many very helpful comments. Our work was supported by the Natural Sciences and Engineering Research Council of Canada and by Communications and Information Technology Ontario. A part of this work was done while Eric Ruppert was visiting Brown University.

References

1. K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 291–302, 1988
2. K. Abrahamson, A. Adler, R. Gelbart, L. Higham, D. Kirkpatrick. The bit complexity of randomized leader election on a ring. *SIAM J. Comput.*, 18(1): 12–29, Feb. 1989
3. K. Abrahamson, A. Adler, L. Higham, D. Kirkpatrick. Randomized function evaluation on a ring. *Distributed Computing*, 3(3): 107–117, July 1989
4. K. Abrahamson, A. Adler, L. Higham, D. Kirkpatrick. Tight lower bounds for probabilistic solitude verification on anonymous rings. *J. ACM*, 41(2): 277–310, Mar. 1994
5. M. Adler, F. Fich. The complexity of end-to-end communication in memoryless networks. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 239–248, 1999
6. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4): 873–890, Sept. 1993
7. Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D.-W. Wang, L. Zuck. Reliable communication over unreliable channels. *J. ACM*, 41(6): 1267–1297, Nov. 1994
8. Y. Afek, D. Dauber, D. Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995
9. Y. Afek, E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM J. Comput.*, 20(2): 376–394, Apr. 1991
10. Y. Afek, D. S. Greenberg, M. Merritt, G. Taubenfeld. Computing with faulty shared objects. *J. ACM*, 42(6): 1231–1274, Nov. 1995
11. Y. Afek, M. Merritt, G. Taubenfeld. The power of multi-objects. *Information and Computation*, 153(1): 117–138, Aug. 1999
12. Y. Afek, G. Stupp. Optimal time-space tradeoff for shared memory leader election. *J. Algorithms*, 25(1): 95–117, Oct. 1997
13. D. Agrawal, M. Choy, H. V. Leong, A. K. Singh. Mixed consistency: A model for parallel programming. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 101–110, 1994
14. M. K. Aguilera, S. Toueg. A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. *Inf. Process. Lett.*, 71(3-4): 155–158, Aug. 1999
15. M. Ahamad, R. A. Bazzi, R. John, P. Kohli, G. Neiger. The power of processor consistency. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 251–260, 1993
16. M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1): 37–49, 1995

17. N. Allenberg-Navony, A. Itai, S. Moran. Average and randomized complexity of distributed problems. *SIAM J. Comput.*, 25(6): 1254–1267, Dec. 1996
18. R. Alur, H. Attiya, G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM J. Comput.*, 26(2): 539–556, 1997
19. R. Alur, G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation*, 126(1): 62–73, Apr. 1996
20. E. S. Amdur, S. M. Weber, V. Hadzilacos. On the message complexity of binary Byzantine agreement under crash failures. *Distributed Computing*, 5(4): 175–186, Apr. 1992
21. J. Anderson, J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1): 68–84, Jan. 1996
22. J. H. Anderson. Composite registers. *Distributed Computing*, 6(3): 141–154, Apr. 1993
23. J. H. Anderson, Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, 2001
24. R. Anderson, H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 370–380, 1991
25. R. Anderson, H. Woll. Algorithms for the certified write-all problem. *SIAM J. Comput.*, 26(5): 1277–1283, 1997
26. D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980
27. E. Arjomandi, M. J. Fischer, N. A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *J. ACM*, 30(3): 449–456, July 1983
28. J. Aspnes. Time- and space-efficient randomized consensus. *J. Algorithms*, 14(3): 414–431, May 1993
29. J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. ACM*, 45(3): 415–450, May 1998
30. J. Aspnes, M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990
31. J. Aspnes, W. Hurwood. Spreading rumors rapidly despite an adversary. *J. Algorithms*, 26(2): 386–411, Feb. 1998
32. J. Aspnes, O. Waarts. Randomized consensus in $O(n \log^2 n)$ operations per processor. *SIAM J. Comput.*, 25(5): 1024–1044, Oct. 1996
33. C. Attiya, D. Dolev, J. Gil. Asynchronous Byzantine consensus. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 119–133, 1984
34. H. Attiya. A direct proof of the asynchronous lower bound for k-set consensus. Technical Report 0930, Computer Science Department, Technion, 1998. Available from `www.cs.technion.ac.il/Reports`
35. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3): 524–548, July 1990
36. H. Attiya, T. Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *Journal of Parallel and Distributed Computing*, 61(8): 1096–1109, Aug. 2001
37. H. Attiya, C. Dwork, N. Lynch, L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1): 122–152, Jan. 1994
38. H. Attiya, R. Friedman. A correctness condition for high-performance multiprocessors. *SIAM J. Comput.*, 27(6): 1637–1670, Dec. 1998
39. H. Attiya, A. Gorbach, S. Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2): 162–183, Mar. 2002
40. H. Attiya, A. Herzberg, S. Rajsbaum. Optimal clock synchronization under different delay assumptions. *SIAM J. Comput.*, 25(2): 369–389, Apr. 1996
41. H. Attiya, N. Lynch, N. Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4): 725–763, July 1994
42. H. Attiya, Y. Mansour. Language complexity on the synchronous anonymous ring. *Theoretical Comput. Sci.*, 53(2–3): 169–185, 1987
43. H. Attiya, M. Mavronicolas. Efficiency of semisynchronous versus asynchronous networks. *Mathematical Systems Theory*, 27(6): 547–571, Nov./Dec. 1994
44. H. Attiya, S. Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4): 1286–1313, 2002
45. H. Attiya, M. Snir. Better computing on the anonymous ring. *J. Algorithms*, 12(2): 204–238, June 1991
46. H. Attiya, M. Snir, M. K. Warmuth. Computing on an anonymous ring. *J. ACM*, 35(4): 845–875, Oct. 1988
47. H. Attiya, J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998
48. H. Attiya, J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2): 91–122, May 1994
49. Y. Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 1997
50. B. Awerbuch. Communication-time trade-offs in network synchronization. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pages 272–276, 1985
51. B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4): 804–823, Oct. 1985
52. B. Awerbuch, A. Baratz, D. Peleg. Cost-sensitive analysis of communication protocols. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–188, 1990
53. B. Awerbuch, O. Goldreich, D. Peleg, R. Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2): 238–256, Apr. 1990
54. Ö. Babaoğlu, P. Stephenson, R. Drummond. Reliable broadcasts and communication models: tradeoffs and lower bounds. *Distributed Computing*, 2(4): 177–189, Aug. 1988
55. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9): 1053–1065, Sept. 1989
56. Z. Bar-Joseph, M. Ben-Or. A tight lower bound for randomized synchronous consensus. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 193–199, 1998
57. K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson. A note on reliable, full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5): 260–261, May 1969
58. R. A. Bazzi, G. Neiger. Simplifying fault-tolerance: Providing the abstraction of crash failures. *J. ACM*, 48(3): 499–554, May 2001
59. Y. Ben-Asher, D. G. Feitelson, L. Rudolph. ParC — an extension of C for shared memory parallel processing. *Software Practice and Experience*, 26(5): 581–612, 1996
60. S. Biaz, J. L. Welch. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Inf. Process. Lett.*, 80(3): 151–157, Nov. 2001

61. O. Biran, S. Moran, S. Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *J. Algorithms*, 11(3): 420–440, Sept. 1990
62. O. Biran, S. Moran, S. Zaks. Deciding 1-solvability of distributed task is NP-hard. In *Graph-Theoretic Concepts in Computer Science. 16th International Workshop*, volume 484 of *LNCS*, pages 206–220, 1990
63. O. Biran, S. Moran, S. Zaks. Tight bounds on the round complexity of distributed 1-solvable tasks. *Theoretical Comput. Sci.*, 145(1–2): 271–290, July 1995
64. H. L. Bodlaender. New lower bound techniques for distributed leader finding and other problems on rings of processors. *Theoretical Comput. Sci.*, 81(2): 237–256, Apr. 1991
65. H. L. Bodlaender, S. Moran, M. K. Warmuth. The distributed bit complexity of the ring: From the anonymous to the non-anonymous case. *Information and Computation*, 108(1): 34–50, Jan. 1994
66. P. Boldi, S. Vigna. Computing anonymously with arbitrary knowledge. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 181–188, 1999
67. P. Boldi, S. Vigna. An effective characterization of computability in anonymous networks. In *Distributed Computing, 15th International Symposium*, pages 33–47, 2001
68. E. Borowsky. *Capturing the power of resiliency and set consensus in distributed systems*. PhD thesis, University of California, Los Angeles, Oct. 1995
69. E. Borowsky, E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, 1993
70. E. Borowsky, E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 41–52, 1993
71. E. Borowsky, E. Gafni. A simple algorithmically reasoned characterization of wait-free computations. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 189–198, 1997
72. E. Borowsky, E. Gafni, N. Lynch, S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3): 127–146, July 2001
73. L. Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Inf.*, 25(2): 179–201, Feb. 1988
74. G. Bracha, O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Distributed Algorithms, 5th International Workshop*, volume 579 of *LNCS*, pages 143–150, 1991
75. G. Bracha, S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4): 824–840, Oct. 1985
76. M. F. Bridgland, R. J. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 52–63, 1987
77. J. Burns, N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2): 171–184, Dec. 1993
78. J. E. Burns. A formal model for message passing systems. Technical Report 91, Indiana University Computer Science Department, 1980
79. J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, G. L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *J. ACM*, 29(1): 183–205, Jan. 1982
80. J. E. Burns, G. L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–157, 1989
81. J. F. Buss, P. C. Kanellakis, P. L. Ragde, A. A. Shvartsman. Parallel algorithms with processor failures and delays. *J. Algorithms*, 20(1): 45–86, Jan. 1996
82. J. Cage. An autobiographical statement. *Southwest Review*, 76(1): 59, 1991. A description of advice from his father, an inventor
83. L. Carroll. *Through the Looking Glass, and What Alice Found There*, chapter IX. Macmillan, 1872
84. M. Castro, B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186, 1999
85. T. Chandra, V. Hadzilacos, P. Jayanti, S. Toueg. The h_m^r hierarchy is not robust. Manuscript, 1994
86. T. Chandra, V. Hadzilacos, P. Jayanti, S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 334–343, 1994
87. T. Chandra, V. Hadzilacos, S. Toueg, B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 322–330, 1996
88. T. Chandra, S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2): 225–267, Mar. 1996
89. T. D. Chandra, V. Hadzilacos, S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4): 685–722, July 1996
90. T. D. Chandra, P. Jayanti, K. Tan. A polylog time wait-free construction for closed objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998
91. K. M. Chandy, J. Misra. *Parallel Program Design*. Addison-Wesley, 1988
92. S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1): 132–158, July 1993
93. S. Chaudhuri, M. Herlihy, N. A. Lynch, M. R. Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5): 912–943, Sept. 2000
94. S. Chaudhuri, M. Herlihy, M. R. Tuttle. Wait-free implementations in message-passing systems. *Theoretical Comput. Sci.*, 220(1): 211–245, June 1999
95. S. Chaudhuri, P. Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. In *Distributed Algorithms, 10th International Workshop*, volume 1151 of *LNCS*, 1996
96. S. Chaudhuri, J. L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2): 335–354, Apr. 1994
97. G. V. Chockler, I. Keidar, R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4): 427–469, Dec. 2001
98. B. Chor, A. Israeli, M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987
99. B. Chor, M. Merritt, D. B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *J. ACM*, 36(3): 591–614, July 1989
100. B. Chor, L. Moscovici. Solvability in asynchronous environments. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 422–427, 1989

101. B. Chor, L.-B. Nelson. Solvability in asynchronous environments II: Finite interactive tasks. *SIAM J. Comput.*, 29(2): 351–377, Apr. 2000
102. I. Cidon, Y. Shavitt. Message terminating algorithms for anonymous rings of unknown size. *Inf. Process. Lett.*, 54(2): 111-119, Apr. 1995
103. B. A. Coan, D. Dolev, C. Dwork, L. Stockmeyer. The distributed firing squad problem. *SIAM J. Comput.*, 18(5): 990–1012, Oct. 1989
104. B. A. Coan, C. Dwork. Simultaneity is harder than agreement. *Information and Computation*, 91(2): 205–231, Apr. 1991
105. B. A. Coan, J. L. Welch. Transaction commit in a realistic timing model. *Distributed Computing*, 4(2): 87–103, June 1990
106. S. Cook, C. Dwork, R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1): 87–97, Feb. 1986
107. R. Cori, S. Moran. Exotic behaviour of consensus numbers. In *Distributed Algorithms, 8th International Workshop*, volume 857 of *LNCS*, pages 101–115, 1994
108. A. Cremers, T. N. Hibbard. An algebraic approach to concurrent programming control and related complexity problems. In *Algorithms and Complexity: New Directions and Recent Results*, page 446, New York, 1976. Academic Press. Abstract only
109. R. De Prisco, D. Malkhi, M. K. Reiter. On k-set consensus in asynchronous systems. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 257–265, 1999
110. R. A. DeMillo, N. A. Lynch, M. J. Merritt. Cryptographic protocols. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 383–400, 1982
111. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9): 569, Sept. 1965
112. E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1(2): 115–138, 1971
113. D. Dolev. The Byzantine generals strike again. *J. Algorithms*, 3(1): 14–30, Mar. 1982
114. D. Dolev, C. Dwork, L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1): 77–97, Jan. 1987
115. D. Dolev, J. Y. Halpern, H. R. Strong. On the possibility and impossibility of achieving clock synchronization. *J. Comput. Syst. Sci.*, 32(2): 230–250, Apr. 1986
116. D. Dolev, M. Klawe, M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *J. Algorithms*, 3: 245–260, 1982
117. D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, W. E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3): 499–516, July 1986
118. D. Dolev, R. Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1): 191–204, Jan. 1985
119. D. Dolev, R. Reischuk, H. R. Strong. Early stopping in Byzantine agreement. *J. ACM*, 37(4): 720–741, 1990
120. D. Dolev, H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.*, 12(4): 656–666, Nov. 1983
121. S. Dolev. *Self-stabilization*. MIT Press, 2000
122. S. Dolev, J. Welch. Crash resilient communication in dynamic networks. *IEEE Transactions on Computers*, 46(1): 14–26, Jan. 1997
123. P. Duris, Z. Galil. Two lower bounds in asynchronous distributed computation. *J. Comput. Syst. Sci.*, 42(3): 254–266, June 1991
124. C. Dwork, M. Herlihy, O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44(6): 779–805, Nov. 1997
125. C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2): 288–323, Apr. 1988
126. C. Dwork, Y. Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2): 156–186, Oct. 1990
127. C. Dwork, D. Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, 1983
128. P. Fatourou, F. Fich, E. Ruppert. Space-optimal multi-writer snapshot objects are slow. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 13–20, 2002
129. P. Fatourou, F. Fich, E. Ruppert. A tight time lower bound for space-optimal implementations of multi-writer snapshots. In *Proceedings of the 35th ACM Symposium on Theory of Computing*, 2003. To appear
130. A. Fekete, N. Lynch. The need for headers: An impossibility result for communication over unreliable channels. In *CONCUR'90 Theories of Concurrency: Unification and Extensions*, volume 458 of *LNCS*, pages 199–215, 1990
131. A. Fekete, N. Lynch, Y. Mansour, J. Spinelli. The impossibility of implementing reliable communication in the face of crashes. *J. ACM*, 40(5): 1087–1107, Nov. 1993
132. A. D. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1): 9–29, Jan. 1990
133. A. D. Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1): 95–124, Nov. 1994
134. P. Feldman, S. Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM J. Comput.*, 26(4): 873–933, Aug. 1997
135. C. Fetzer, F. Cristian. Lower bounds for convergence function based clock synchronization. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–143, 1995
136. F. Fich, M. Herlihy, N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5): 843–862, Sept. 1998
137. F. Fich, E. Ruppert. Lower bounds in distributed computing. In *Distributed Computing, 14th International Symposium*, volume 1914 of *LNCS*, pages 1–28, 2000
138. M. J. Fischer. The consensus problem in unreliable distributed systems. In *Proc. International Conference on Foundations of Computation Theory*, pages 127–140, 1983. Also available as Yale Technical Report, Department of Computer Science, YALEU/DCS/RR-273, June 1983
139. M. J. Fischer, N. A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4): 183–186, June 1982
140. M. J. Fischer, N. A. Lynch, M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1): 26–39, Jan. 1986
141. M. J. Fischer, N. A. Lynch, M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2): 374–382, Apr. 1985
142. M. J. Fischer, S. Moran, S. Rudich, G. Taubenfeld. The wakeup problem. *SIAM J. Comput.*, 25(6): 1332–1357, 1996
143. G. N. Frederickson, N. A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1): 98–115, Jan. 1987
144. R. Friedman. Implementing hybrid consistency with high-level synchronization operations. *Distributed Computing*, 9(3): 119-129, Dec. 1995
145. E. Gafni. Distributed computing. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 48. CRC Press, 1998
146. E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proceedings of the 17th Annual ACM Sym-*

posium on Principles of Distributed Computing, pages 143–152, 1998
147. E. Gafni, E. Koutsoupias. Three-processor tasks are undecidable. *SIAM J. Comput.*, 28(3): 970–983, Jan. 1999
148. J. A. Garay, K. J. Perry. A continuum of failure models for distributed computing. In *Distributed Algorithms, 6th International Workshop*, volume 647 of *LNCS*, pages 153–165, 1992
149. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990. Proceedings published in *Computer Architecture News*, 18(2)
150. O. Goldreich. Cryptography and cryptographic protocols. *Distributed Computing*, in this issue
151. O. Goldreich, D. Sneh. On the complexity of global computation in the presence of link failures: The case of uni-directional faults. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 103–111, 1992
152. E. Goubault, T. P. Jensen. Homology of higher dimensional automata. In *Proc. 3rd International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 254–268, 1992
153. R. L. Graham, A. C. Yao. On the improbability of reaching Byzantine agreements. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 467–478, 1989
154. J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1979
155. D. S. Greenberg, G. Taubenfeld, D.-W. Wang. Choice coordination with multiple alternatives. In *Distributed Algorithms, 6th International Workshop*, volume 647 of *LNCS*, pages 54–68, 1992
156. J. F. Groote, W. Hesselink, S. Mauw, R. Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14(2): 75–81, Apr. 2001
157. J. Gunawardena. Homotopy, concurrency. *Bulletin of the EATCS*, 54: 184–193, Oct. 1994
158. R. Gupta, S. A. Smolka, S. Bhaskar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1): 7–86, Mar. 1994
159. V. Hadzilacos. A lower bound for Byzantine agreement with fail-stop processors. Technical Report 21-83, Harvard University, July 1983
160. V. Hadzilacos. Connectivity requirements for Byzantine agreement under restricted types of failures. *Distributed Computing*, 2(2): 95–103, Aug. 1987
161. V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 129–134, 1987
162. V. Hadzilacos, J. Y. Halpern. Message-optimal protocols for Byzantine agreement. *Mathematical Systems Theory*, 26(1): 41–102, 1993
163. V. Hadzilacos, S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, May 1994
164. S. Haldar, K. Vidyasankar. Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *Acta Informatica*, 33(2): 177–202, 1996
165. J. Y. Halpern, N. Megiddo, A. A. Munshi. Optimal precision in the presence of uncertainty. *Journal of Complexity*, 1(2): 170–196, Dec. 1985
166. J. Y. Halpern, Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3): 549–587, July 1990
167. J. Havlicek. Computable obstructions to wait-free computability. *Distributed Computing*, 13(2): 59–83, 2000
168. M. Herlihy. Impossibility results for asynchronous PRAM. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 327–336, 1991
169. M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1): 124–149, Jan. 1991
170. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Syst.*, 15(5): 745–770, Nov. 1993
171. M. Herlihy, S. Rajsbaum. Set consensus using arbitrary objects. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 324–333, 1994
172. M. Herlihy, S. Rajsbaum. Algebraic spans. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, 1995
173. M. Herlihy, S. Rajsbaum. Algebraic topology and distributed computing: A primer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 203–217. Springer, 1995
174. M. Herlihy, S. Rajsbaum. The decidability of distributed decision tasks. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 589–598, 1997. Full version available from `www.cs.brown.edu/people/mph/decide.html`
175. M. Herlihy, S. Rajsbaum. A wait-free classification of loop agreement tasks. In *Distributed Computing, 12th International Symposium*, volume 1499 of *LNCS*, pages 175–185, 1998
176. M. Herlihy, S. Rajsbaum. New perspectives in distributed computing. In *Mathematical Foundations of Computer Science: 24th International Symposium*, volume 1672 of *LNCS*, pages 170–186, 1999
177. M. Herlihy, S. Rajsbaum, M. R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–142, 1998. Full version available from `www.cs.brown.edu/people/mph/hrt.html`
178. M. Herlihy, E. Ruppert. On the existence of booster types. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 653–663, 2000
179. M. Herlihy, N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6): 858–923, Nov. 1999
180. M. Herlihy, N. Shavit, O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4): 193–203, 1996
181. M. P. Herlihy, J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3): 463–492, July 1990
182. L. Higham, J. Kawash. Bounds for mutual exclusion with only processor consistency. In *Distributed Computing, 14th International Symposium*, volume 1914 of *LNCS*, pages 44–58, 2000
183. L. Higham, J. Kawash, N. Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th International Conference on Parallel and Distributed Computing Systems*, pages 349–356, 1997
184. L. Higham, T. Przytycka. Asymptotically optimal election on weighted rings. *SIAM J. Comput.*, 28(2): 720–732, 1998
185. G. Hoest, N. Shavit. Towards a topological characterization of asynchronous complexity. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 199–208, 1997
186. P. W. Hutto, M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *The 10th International Conference on Distributed Computing Systems*, pages 302–309, 1990
187. M. Inoue, S. Moriya, T. Masuzawa, H. Fujiwara. Optimal wait-free clock synchronization protocol on a shared-memory

multi-processor system. In *Distributed Algorithms, 11th International Workshop*, volume 1320 of *LNCS*, pages 290–304, 1997

188. A. Israeli, A. Shirazi. The time complexity of updating snapshot memories. *Inf. Process. Lett.*, 65(1): 33–40, Jan. 1998
189. A. Itai, M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1): 60–87, 1990
190. J. James, A. K. Singh. Fault tolerance bounds for memory consistency. In *Distributed Algorithms, 11th International Workshop*, volume 1320 of *LNCS*, pages 200–214, 1997
191. P. Jayanti. Wait-free computing. In *Distributed Algorithms, 9th International Workshop*, volume 972 of *LNCS*, pages 19–50, 1995
192. P. Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4): 592–614, July 1997
193. P. Jayanti. A lower bound on the local time complexity of universal constructions. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192, 1998
194. P. Jayanti. Solvability of consensus: Composition breaks down for nondeterministic types. *SIAM J. Comput.*, 28(3): 782–797, Sept. 1998
195. P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998
196. P. Jayanti, T. D. Chandra, S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3): 451–500, 1998
197. P. Jayanti, A. S. Sethi, E. L. Lloyd. Minimal shared information for concurrent reading and writing. In *Distributed Algorithms, 5th International Workshop*, volume 579 of *LNCS*, pages 212–228, 1991
198. P. Jayanti, K. Tan, S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2): 438–456, 2000
199. P. Jayanti, S. Toueg. Wakeup under read/write atomicity. In *Distributed Algorithms, 4th International Workshop*, volume 486 of *LNCS*, pages 277–288, 1990
200. P. Jayanti, S. Toueg. Some results on the impossibility, universality and decidability of consensus. In *Distributed Algorithms, 6th International Workshop*, volume 647 of *LNCS*, pages 69–84, 1992
201. M. Jayaram, G. Varghese. Crash failures can drive protocols to arbitrary states. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 247–256, 1996
202. R. E. Johnson, F. B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, 1985
203. Y.-J. Joung. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Comput. Sci.*, 243(1-2): 307–338, July 2000
204. C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 285–296, 1990
205. P. C. Kanellakis, A. A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4): 201–217, Apr. 1992
206. P. C. Kanellakis, A. A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, Boston, 1997
207. Z. Kedem, K. Palem, A. Raghunathan, P. Spirakis. Combining tentative and definite executions for very fast dependable computing. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 381–390, 1991
208. J. M. Kleinberg, S. Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–144, 1993
209. R. Koo, S. Toueg. Effects of message loss on the termination of distributed protocols. *Inf. Process. Lett.*, 27(4): 181–188, Apr. 1988
210. E. Korach, S. Moran, S. Zaks. The optimality of distributive constructions of minimum weight and degree restricted spanning trees in a complete network of processors. *SIAM J. Comput.*, 16(2): 231–236, Apr. 1987
211. E. Korach, S. Moran, S. Zaks. Optimal lower bounds for some distributed algorithms for a complete network of processors. *Theoretical Comput. Sci.*, 64(1): 125–132, Apr. 1989
212. M. J. Kosa. Time bounds for strong and hybrid consistency for arbitrary abstract data types. *Chicago Journal of Theoretical Computer Science*, Aug. 1999. Available from `cjtcs.cs.uchicago.edu`
213. C. P. Kruskal, L. Rudolph, M. Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Trans. Prog. Lang. Syst.*, 10(4): 579–601, Oct. 1988
214. E. Kushilevitz, Y. Mansour, M. O. Rabin, D. Zuckerman. Lower bounds for randomized mutual exclusion. *SIAM J. Comput.*, 27(6): 1550–1563, Dec. 1998
215. R. E. Ladner, A. LaMarca, E. Tempero. Counting protocols for reliable end-to-end transmission. *J. Comput. Syst. Sci.*, 56(1): 96–111, Feb. 1998
216. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8): 453–455, Aug. 1974
217. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9): 690–691, Sept. 1979
218. L. Lamport. The weak Byzantine generals problem. *J. ACM*, 30(3): 668–676, July 1983
219. L. Lamport. The mutual exclusion problem. Part II: Statement and solutions. *J. ACM*, 33(2): 327–348, Apr. 1986
220. L. Lamport. On interprocess communication, part II: Algorithms. *Distributed Computing*, 1(2): 86–101, Apr. 1986
221. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1): 1–11, Feb. 1987
222. L. Lamport, M. Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, Menlo Park, Apr. 1982
223. L. Lamport, N. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*, volume B, chapter 18. Elsevier, 1990
224. L. Lamport, P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1): 52–78, 1985
225. L. Lamport, R. Shostak, M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.*, 4(3): 382–401, July 1982
226. W.-K. Lo. More on t-resilience vs. wait-freedom. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 110–119, 1995
227. W.-K. Lo, V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3): 689–728, 2000
228. W.-K. Lo, V. Hadzilacos. On the power of shared object types to implement one-resilient consensus. *Distributed Computing*, 13(4): 219–238, 2000

229. Z. Lotker, B. Patt-Shamir, D. Peleg. Distributed MST for constant diameter graphs. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 63–71, 2001
230. M. C. Loui, H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987
231. R. Lubitch, S. Moran. Closed schedulers: A novel technique for analyzing asynchronous protocols. *Distributed Computing*, 8(4): 203–210, 1995
232. J. Lundelius, N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3): 190–204, Aug./Sept. 1984
233. N. A. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–27, 1989
234. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996
235. N. A. Lynch, M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Comput. Sci.*, 13(1): 17–43, Jan. 1981
236. D. Malkhi, M. Merritt, M. Reiter, G. Taubenfeld. Objects shared by Byzantine processes. In *Distributed Computing, 14th International Symposium*, volume 1914 of *LNCS*, pages 345–359, 2000
237. Y. Mansour, B. Schieber. The intractability of bounded protocols for on-line sequence transmission over non-FIFO channels. *J. ACM*, 39(4): 783–799, Oct. 1992
238. Y. Mansour, S. Zaks. On the bit complexity of distributed computations in a ring with a leader. *Information and Computation*, 75(2): 162–177, Nov. 1987
239. C. Martel, R. Subramonian. On the complexity of certified write-all algorithms. *J. Algorithms*, 16: 361–387, 1994
240. M. Mavronicolas, D. Roth. Linearizable read/write objects. *Theoretical Comput. Sci.*, 220(1): 267–319, June 1999
241. M. Merritt, O. Reingold, G. Taubenfeld, R. Wright. Tight bounds for shared memory systems accessed by Byzantine processes. In *Distributed Computing, 16th International Symposium*, pages 222–236, 2002
242. M. Merritt, G. Taubenfeld. Atomic m-register operations. *Distributed Computing*, 7(4): 213–221, May 1994
243. M. Merritt, G. Taubenfeld. Fairness of shared objects. In *Distributed Computing, 12th International Symposium*, volume 1499 of *LNCS*, pages 303–316, 1998
244. L. Mies van der Rohe. On restraint in design. *New York Herald Tribune*, 28 June 1959
245. S. Moran, L. Rappoport. On the robustness of h^r_m. In *Distributed Algorithms, 10th International Workshop*, volume 1151 of *LNCS*, pages 344–361, 1996
246. S. Moran, G. Taubenfeld. A lower bound on wait-free counting. *J. Algorithms*, 24(1): 1–19, July 1997
247. S. Moran, G. Taubenfeld, I. Yadin. Concurrent counting. *J. Comput. Syst. Sci.*, 53(1): 61–78, Aug. 1996
248. S. Moran, M. K. Warmuth. Gap theorems for distributed computation. *SIAM J. Comput.*, 22(2): 379–394, Apr. 1993
249. S. Moran, Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Inf. Process. Lett.*, 26(3): 145–151, Nov. 1987
250. Y. Moses, S. Rajsbaum. A layered analysis of consensus. *SIAM J. Comput.*, 31(4): 989–1021, 2002
251. Y. Moses, M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1): 121-169, 1988
252. A. Mostefaoui, S. Rajsbaum, M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. In *Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 153–162, 2001
253. M. Naor, L. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6): 1259–1277, 1995
254. G. Neiger. Distributed consensus revisited. *Inf. Process. Lett.*, 49: 195–201, Feb. 1994
255. G. Neiger. Set-linearizability. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, page 396, 1994
256. G. Neiger, S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *J. Algorithms*, 11(3): 374–419, Sept. 1990
257. N. Nishimura. A model for asynchronous shared memory parallel computation. *SIAM J. Comput.*, 23(6): 1231–1252, 1994
258. J. K. Pachl. A lower bound for probabilistic distributed algorithms. *J. Algorithms*, 8(1): 53–65, Mar. 1987
259. B. Patt-Shamir, S. Rajsbaum. A theory of clock synchronization. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 810–819, 1994
260. M. Pease, R. Shostak, L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2): 228–234, Apr. 1980
261. D. Peleg, V. Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5): 1427–1442, 2000
262. G. L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Prog. Lang. Syst.*, 4(4): 758–762, Oct. 1982
263. G. L. Peterson. Properties of a family of booster types. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, page 281, 1999
264. R. Plunkett, A. Fekete. Approximate agreement with mixed mode faults: Algorithm and lower bound. In *Distributed Computing, 12th International Symposium*, volume 1499 of *LNCS*, pages 333–346, 1998
265. M. O. Rabin. The choice coordination problem. *Acta Inf.*, 17(2): 121–134, June 1982
266. M. O. Rabin, D. Lehmann. The advantages of free choice: A symmetric and fully distributed solution for the dining philosophers problem. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, chapter 20. Prentice Hall, 1994
267. O. Rachman. Anomalies in the wait-free hierarchy. In *Distributed Algorithms, 8th International Workshop*, volume 857 of *LNCS*, pages 156–163, 1994
268. G. M. Reed, A. W. Roscoe, R. F. Wachter, editors. *Topology and Category Theory in Computer Science*. Clarendon Press, Oxford, 1991
269. J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, California, 1993
270. R. Reischuk, M. Koshors. Lower bounds for synchronous networks and the advantage of local information. In *Distributed Algorithms, 2nd International Workshop*, volume 312 of *LNCS*, pages 374–387, 1987
271. I. Rhee, J. L. Welch. The impact of time on the session problem. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 191–202, 1992
272. P. Rosenstiehl, J. R. Fiksel, A. Holliger. Intelligent graphs: Networks of finite automata capable of solving graph problems. In R. C. Read, editor, *Graph Theory and Computing*, pages 219–265. Academic Press, New York, 1972
273. E. Ruppert. Consensus numbers of multi-objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 211–217, 1998
274. E. Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4): 1156–1168, 2000

275. J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5): 651–650, 1999
276. A. Russell, M. Saks, D. Zuckerman. Lower bounds for leader election and collective coin-flipping in the perfect information model. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 339–347, 1999
277. N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 173–179, 1999
278. M. Saks, F. Zaharoglou. Optimal space distributed order-preserving lists. *J. Algorithms*, 31(2): 320–334, May 1999. The lower bound in this paper is joint work with R. Cori
279. M. Saks, F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5): 1449–1483, Mar. 2000
280. N. Santoro, P. Widmayer. Time is not a healer. In *6th Annual Symposium on Theoretical Aspects of Computer Science*, volume 349 of *LNCS*, pages 304–313, 1989
281. E. Schenk. Faster approximate agreement with multi-writer registers. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1995
282. E. Schenk. *Computability and Complexity Results for Agreement Problems in Shared Memory Distributed Systems*. PhD thesis, Department of Computer Science, University of Toronto, Sept. 1996
283. E. Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 279, 1997
284. A. Segall, O. Wolfson. Transaction commitment at minimal communication cost. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 112–118, 1987
285. B. Simons, J. L. Welch, N. Lynch. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 84–96, 1990
286. G. Singh. Leader election in complete networks. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 179–190, 1992
287. E. Sperner. Neuer Beweis für die Invarianz der Dimensionszahl und des Gebietes. *Abhandlungen aus dem Mathematischen Seminar der Hamburgischen Universität*, 6(3/4): 265–272, Sept. 1928
288. T. K. Srikanth, S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3): 626–645, July 1987
289. M. Stenning. A data transfer protocol. *Computer Net.*, 1: 99–110, 1976
290. F. Stomp, G. Taubenfeld. Constructing a reliable test&set bit. *IEEE Transactions on Parallel and Distributed Systems*, 10(3): 252–265, Mar. 1999
291. G. Taubenfeld, S. Katz, S. Moran. Initial failures in distributed computations. *International Journal of Parallel Programming*, 18(4): 255–276, Aug. 1989
292. G. Taubenfeld, S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Inf.*, 33(1): 1–20, Feb. 1996
293. J. Valois. Space bounds for transactional synchronization. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, page 243, 1996
294. D. Wang, L. Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 73–83, 1989
295. R. Wattenhofer, P. Widmayer. An inherent bottleneck in distributed counting. *Journal of Parallel and Distributed Computing*, 49(1): 135–145, Feb. 1998
296. J. L. Welch. Simulating synchronous processors. *Information and Computation*, 74(2): 159–171, Aug. 1987
297. J. L. Welch, N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1): 1–36, 1988
298. M. Yamashita, T. Kameda. Computing functions on asynchronous anonymous networks. *Mathematical Systems Theory*, 29(4): 331–356, July/Aug. 1996. See Erratum in *Theory of Computing Systems*, 31(1): 109, 1998
299. A. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977

Index

This index contains entries for the problems and proof techniques mentioned in this survey. For definitions related to models of distributed systems, see Sect. 2.

adversary argument, 4.1, 5.1, 5.2, 5.3, 5.6, 6.4, 7.3, 11.1, 11.3, 11.5, 12.2, 13.1, 13.2
approximate agreement, 3.2, 5.3, 6.1, 6.5, 8.3, 11.2
Byzantine agreement, *see* terminating reliable broadcast
chain argument, 4.1, 5, 5.1, 11.1, 12.2, 13.1
characterization, 4.1, 5.1, 5.2, 5.4, 5.6, 7.2, 7.3, 8.3, 8.4, 9.2, 13.1
choice coordination, 3.3, 5.6, 10
clock synchronization, 3.2, 5.5, 11.1
clone, 4.1, 5.4, 10
combinatorial argument, 4.1, 9.1, 10, 11.2, 11.5, 12.1, 12.2, 13.2
commit, 3.3, 5.1, 12.2, 13.1
consensus, 3.1, 5.1, 5.2, 5.3, 6, 6.1, 6.4, 7.1, 7.2, 8.1, 9, 9.1, 9.2, 11.1, 11.4, 12.2
  binary, 3.1, 5.1, 8.1
  randomized, 3.1, 10, 13.1, 13.2
constructive argument, 4.1, 6.1, 6.2, 9.1, 11.3, 11.5, 12.2
coordinated attack, *see* simultaneous terminating reliable broadcast
counter-example algorithm, 4.1, 5.1, 5.3, 5.5, 10, 11.1, 12.1
counting, 11.4, 12.2
covering argument, 4.1, 10, 11.4, 11.5, 13.2
decidability, 7.2, 8.3
dining philosophers, 3.4, 4.1, 5.4
distance argument, 4.1, 6.2, 11.5, 12.1
distributed firing squad, 3.3, 5.6, 13.2
end-to-end communication, 6.4
exclusive or, 12.1, 13.1
full-information algorithm, 4, 8.2
function computation, 12.1, 12.2, 13.1
game theory, 7.3
graph theory, 6.2
group membership, 3.3, 5.6
implementation, 2
  `counter` from small objects, 10
  from historyless objects, 10
  `load-linked store-conditional register` from RMW objects, 6.5
  message passing from `registers`, 6
  `move-to-front list` from `registers`, 10

objects from message passing, 6.3
objects from objects of lower consensus number, 9, 9.1
objects from `registers`, 7.1
`register` from message passing, 6.3
`register` from `single-bit registers`, 6.5
`register` from `single-writer registers`, 6.3, 11.2
universal construction, 11.3
`snapshot` from `registers`, 11.5
indistinguishability, 4.1, 5.2, 5.3, 6.3, 6.4, 10, 11.1, 11.5
information-theoretic argument, 4.1, 6.2, 10, 11.2, 11.3, 11.5, 12.2, 13.2
k-assignment, 3.4, 5.6
knowledge-based argument, 4.1, 5.1, 11.1, 12.2
k-set consensus, *see* set consensus
leader election, 3.3, 4.1, 5.4, 5.4, 5.6, 11.1, 12.1, 12.2, 13.2
loop agreement, 8.4
maximum, 12.1, 13.2
minimum spanning tree, 11.5
mutual exclusion, 3.4, 5.6, 6.3, 10, 11.5, 13.2
pigeonhole principle, 4.1, 5.6, 10, 11.4, 12.1
Ramsey theory, 4.1, 12.1
recognizing strings in a language, 12.1
reduction, 4.1, 5.3, 5.6, 7.1, 8.3, 12.1, 13.2
renaming, 3.4, 5.6, 8.3, 11.1, 11.5
order-preserving, 3.4, 5.6
robustness, 9, 9.1, 9.2
safe agreement, 6.1
scenario argument, 4.1, 5.3, 5.5, 5.6, 9.1, 12.1, 12.2, 13.1
separation, 6.2, 11.2
sequence transmission, 6.4
session problem, 6.2
set consensus, 3.1, 3.3, 5.6, 6.1, 8.2, 8.3, 8.4, 11.1
shifting argument, 4.1, 5.5 6.3 6.3
simulation, 2, 4.1, 5.1, 5.3, 6.1, 6.2, 6.4, 9.1, 11.1, 13.2
BG, 6.1, 8.2, 9.2
simultaneous consensus, 3.3, 5.1, 11.1
randomized, 13.2
simultaneous terminating reliable broadcast, 3.3, 11.1
solitude verification, 12.1
spanning tree, 12.2
see also minimum spanning tree
stretching argument, 4.1, 5.5
symmetry argument, 4.1, 5.4, 5.6, 12.1
task assignment, 3.4, 5.6
team consensus, 7.2
terminating reliable broadcast, 3.3, 5.1, 5.3, 11.1, 12.2, 13.1
topological argument, 4.1, 5.6, 7.2, 8, 8.1, 8.2, 8.3, 8.4, 11.2
tradeoff, 6.2, 12.2, 13.2
two generals, 5
undecidability, 7.1, 8.3
valency argument, 4.1, 5.1, 5.2, 5.3, 5.6, 6.1, 7.2, 7.3, 8.3, 9.1, 9.2, 10, 11.1, 11.2, 13.1, 13.2
wakeup, 3.3, 11.3
weak agreement, 9.1
write-all, 3.4, 13.2